Fairness, Correctness, and Automation

by

Samuel E. P. Drews

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2020

Date of final oral examination: 12/10/2020

The dissertation is approved by the following members of the Final Oral Committee:

Aws Albarghouthi, Assistant Professor, Computer Sciences
Loris D'Antoni, Assistant Professor, Computer Sciences
Sharon Li, Assistant Professor, Computer Sciences
Vikas Singh, Professor, Biostatistics and Medical Informatics

To Nancy, Dennis, Will, and Lulu.

ACKNOWLEDGMENTS

This dissertation concludes a five–and–a–half–year period of graduate study at UW–Madison. While the denouement has been marred by the impact of COVID–19, the journey has indeed reached its end; along the way I had the privilege of learning from, working with, and being supported by countless individuals who enriched my life and without whom this work would not have been possible.

I am twice as fortunate as many of my peers in that I have been graced with not one, but *two* fantastic academic advisors, Aws Albarghouthi and Loris D'Antoni. I owe so much to both of them for their mentorship, encouragement, patience, and for believing in me even when I did not. I benefited greatly from their presence, as well as that of everyone else in the MadPL group, faculty and students alike. A special mention goes out to Calvin Smith, who was a better officemate than one could ask for, and also a dear friend.

Of course, I could not have made it to where I am today without the love and support of my family: my parents, Nancy and Dennis, have always been supportive of my endeavors and challenged me to achieve greater heights; my siblings, Will and Lulu, have always been there to cheer me on. And last but not least, I cannot begin to spout enough appreciation towards my partner, David, who was there for me every step of the way.

Thank you, everyone, from the bottom of my heart.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

As software permeates our world in every imaginable way, from healthcare to policing to autonomous vehicles, algorithms play an expanding role in shaping our everyday lives. Indeed, automated decision-making, particularly in the form of machine learning, has profound transformative potential on our society; this power is accompanied by increasing concerns about the safety and fairness of the machine-learned models involved. We argue that such impactful or safety-critical settings necessitate *exact guarantees* about the quality of the systems involved. Accordingly, this dissertation tackles these problems using classic ideas from *formal methods*, since the artifacts involved—the learning algorithms and the models—are ultimately still programs.

First, we attend to the interface between humans and machine-learned models: The means by which practitioners create models is to provide curated training examples to a learning algorithm; however, it is often unclear how subtleties in the training data translate into program behavior. As a cursory step towards formally, automatically reasoning about the relation between training data input and model output, we focus specifically on *data-poisoning attacks*, a problem in *adversarial machine learning*. We present Antidote, a tool that verifies if the composition of decision-tree learning and classification is robust to small perturbations in the training data. Antidote uses *abstract interpretation* to symbolically train all possible trees for an intractably large space of possible datasets, and in turn, to determine all possible classifications for a given input.

Next, we turn our attention to the notion of *algorithmic fairness*. The research community has debated a wide variety of fairness definitions; we propose to express such definitions formally as probabilistic program properties. With the goal of enabling rigorous reasoning about fairness, we design a novel technique for verifying probabilistic properties that

admits a wide class of decision-making programs, implemented in a tool we call FairSquare. FairSquare is the first verification tool for automatically certifying that a program meets a given fairness property. Our evaluation demonstrates FairSquare's ability to verify fairness for a range of different machine-learned programs.

Finally, in the event that a program is provably unfair (or more generally, that a program does not meet some probabilistic correctness property), we turn our attention to the problem of *program synthesis* for probabilistic programs: automatically constructing a program to meet a probabilistic specification of its desired behavior. We propose *distribution-guided inductive synthesis* (DIGITS), a novel technique that (*i*) iteratively calls a synthesizer on finite sets of samples from a given distribution and (*ii*) verifies that the resulting program is both correct and minimally different from a target program (e.g. an unfair model to be repaired). DIGITS has strong convergence guarantees rooted in *computational learning theory*; our evaluation shows that DIGITS is indeed able to synthesize a range of programs, including repairs to those that FairSquare verified were unfair.

1 INTRODUCTION

The modern era has instilled a great deal of responsibility in "algorithms." Particularly driven by advances in machine learning and big-data technology, we are entrusting automated decision-making processes with tasks that impact human lives in substantial ways, including criminal sentencing (Angwin et al., 2016), health care (Mazurowski et al., 2019), controlling autonomous vehicles (Grigorescu et al., 2020), and much more. We may find it appealing to believe that automating these tasks provides efficiency, objectivity, and uniformity exceeding that of manual human effort, but we must nonetheless be careful when embracing this new wave of technology.

The problem stems from the fact that, frankly, many of these tasks are hard. We delegate them to *machine learning* algorithms because the traditional programming workflow does not work: instead of finding a methodical solution and having developers engineer an implementation, training data examples are handed to a learning algorithm which creates some *model* of the training data. This model, in turn, can be used as a program that operates on new, unseen inputs. The end result is that this program may seem to make decisions well, but we do not necessarily understand how it is doing so. How can we be sure it works correctly? (What does "correctly" mean?)

Traditional "Software 1.0"[1] *logic* programming admits a common pattern for correcting mistakes: users often file bug reports to the developers, describing inputs to the system that result in unexpected outputs (or crashes). The developers can fix this bug by manually tracing the input through the program to identify some line of code whose logic is incorrect. Test cases can be added to ensure that the bug no longer occurs; an

[1] This distinction of software paradigms appears to have been coined in a blog post by Karpathy (2017): "Software 1.0 is code we write. Software 2.0 is code written by the optimization based on an evaluation criterion (such as 'classify this training data correctly')."

ambitious development team could even go so far as to *formally verify* that the code behaves exactly as according to some mathematical specification (see Ringer et al. (2019); Klein et al. (2018) for overviews of real-world, formally verified software and the techniques therein).

However, the development and use of "Software 2.0," *data* programming, is frequently uncooperative with this process. As an example, consider the 2015 incident in which frustrated users reported that Google Photos's machine-learning-based search functionality was tagging images of black people as "gorilla" (BBC, 2015). This bug is not easily fixable, and for years, Google's "solution" has been to avoid tagging images as "gorilla" altogether (Hern, 2018). After all, one could feed the misclassified images in question through the model to try to identify where things go *wrong*, but the models in question are frequently incomprehensible to humans (Knight, 2017), and it is not possible to isolate what "logic" should change. It is likely the case that this Google Photos bug is due (at least in part) to the under-representation of black people in the training dataset. But even if we were to amend the dataset accordingly, would we *know* this racist tagging behavior would be completely eliminated? Could we be certain?

Let us distill two takeaways from this example: First, in "Software 2.0," the only real mechanism by which developers create programs is through feeding a large corpus of training data into a learning algorithm, and it is not always obvious to practitioners how subtleties in the data translate into model behavior. Second, while it is expected that machine-learned models will not have 100% accuracy, they all-too-often disproportionately mistreat minority and vulnerable groups; fixing this disparity is highly non-trivial (Hao, 2019).

The work within this thesis takes modest steps towards equipping developers with the tools they need to automate *responsibly*; specifically, we tackle problems related to *algorithmic fairness* and *adversarial machine learning* from a *formal methods* perspective. Software 2.0 practitioners should

become invested in certifying the reliability, safety, and fairness of their systems, and accordingly, this thesis establishes prototypical techniques to do so. Ideally, the development environments of the future will assist not only with producing models, but also, e.g., with understanding the relation between the training data and those models, or with *guaranteeing* that those models do not exhibit undesirable behavior. Ultimately, this thesis optimistically envisions a future in which our society pursues equity while embracing new technologies.

1.1 Context and Challenges

While the goals we've just discussed are vast, in this thesis we examine only a small number of specific problems, but to great depth, as cursory steps towards that future. Here, we provide a brief overview of relevant background information on program verification and synthesis, on adversarial machine learning, and on algorithmic fairness.

Program Verification and Synthesis. Our motivation is to analyze Software 2.0 artifacts (models and learning algorithms) for *correctness properties*. These artifacts are ultimately still programs; formal methods research has developed a number of techniques to analyze programs, and we will apply them here.

Throughout this thesis, we will occasionally borrow notation from Hoare logic (Hoare, 1969), a classic formal system for analyzing the correctness of programs. Specifically, correctness is stated as a "Hoare triple":

$$\{P\} \quad C \quad \{Q\}$$

where P and Q are logical statements (referred to as the "precondition" and "postcondition," respectively), and C is a code snippet. A Hoare triple asserts that if P is true about the program state, and the code C is

evaluated, then Q will be true afterwards.

For example, the repeated application of bitwise-exclusive-or operations can be used somewhat non-intuitively to perform an in-place swap. We would state the correctness of this operations as follows:

$$\{x = c_1 \wedge y = c_2\} \quad x := x \hat{} y;\ y := x \hat{} y;\ x := x \hat{} y \quad \{x = c_2 \wedge y = c_1\}$$

Hoare logic provides a proof system for establishing the validity of such triples (though we elide those details here).

The field of *program verification* deals more generally with establishing the correctness properites of programs; on the other hand, *program synthesis* is concerned with the creation of programs that are correct by construction. Together, verification and synthesis form the conceptual core of the work within this thesis; however, we will be concerned with novel notions of program correctness derived from the profound way that software has begun to interact with society. Indeed, our correctness properties will look something like:

{An attacker has bounded power}
I use machine learning
{The model is unaffected}

or

{Demographic information about society}
Hiring decisions are made by machine-learning
{The decisions are fair}

which will require substantial conceptual and technical development to formalize, certify, and enforce.

Adversarial Machine Learning. Artificial intelligence, in the form of machine learning, is rapidly transforming the world as we know it. Today, machine learning is responsible for an ever-growing spectrum of sensitive

decisions—from loan decisions, to diagnosing diseases, to autonomous driving. Many recent works have shown how machine learning models are brittle (Szegedy et al., 2014; Wang et al., 2018b; Chen et al., 2017; Biggio et al., 2012; Steinhardt et al., 2017), and with ML spreading across many industries, the issue of robustness in ML models has taken center stage.

The research field that deals with studying robustness of ML models is referred to as *adversarial machine learning*. In this field, researchers have proposed many definitions that try to capture robustness to different *adversaries*. The majority of these works have focused on verifying or improving the model's robustness to *test-time attacks* (Gehr et al., 2018; Singh et al., 2019; Anderson et al., 2019; Katz et al., 2017; Wang et al., 2018a), where an adversary can craft small perturbations to input examples that fool the ML model into changing its prediction, e.g., making small changes to the picture of a cat that causes the model to classify it as a zebra (Carlini and Wagner, 2017).

Another important problem concerns a different threat model: In *data-poisoning attacks*, an adversary can produce slight modifications of the training set, e.g., by supplying a small amount of malicious training points, to influence the produced model and its predictions. This attack model is possible when data is curated, for example, via crowdsourcing or from online repositories, where attackers can try to add malicious elements to the training data. For instance, Xiao et al. (2015a) consider adding malicious training points to affect a malware detection model; similarly, Chen et al. (2017) consider adding a small number of images to bypass a facial recognition model.

Data-poisoning robustness has been studied extensively (Biggio et al., 2012; Xiao et al., 2012, 2015b; Newell et al., 2014; Mei and Zhu, 2015). This body of work has demonstrated data-poisoning attacks—i.e., modifications to training sets—that can degrade classifier accuracy, sometimes dramatically, or force certain predictions on specific inputs. While some

defenses have been proposed against specific attacks (Laishram and Phoha, 2016; Steinhardt et al., 2017), we are not aware of any technique that can formally verify that a given learning algorithm is robust to perturbations to a given training set. Verifying data-poisoning robustness of a given learner requires solving a number of challenges:

- The datasets over which the learner operates are typically large (thousands of elements). Even when considering simple poisoning attacks, the number of modified training sets we need to consider can be intractably large to represent and explore explicitly.
- Because learners are complicated programs that employ complex metrics (e.g., entropy and loss functions), their verification requires new specialized techniques.

Algorithmic Fairness. Software has become a powerful arbitrator of a range of significant decisions with far-reaching societal impact—hiring (Miller, 2015; Kobie, 2016), welfare allocation (Eubanks, 2015), prison sentencing (Angwin et al., 2016), policing (Berg, 2014; Perry, 2013), amongst many others. With the range and sensitivity of algorithmic decisions expanding by the day, the problem of understanding the nature of program bias is a pressing one: Indeed, the notion of *algorithmic fairness* has recently captured the attention of a broad spectrum of experts, within computer science and without (Dwork et al., 2012; Zemel et al., 2013; Feldman et al., 2015; Calders and Verwer, 2010; Datta et al., 2015; Angwin et al., 2016; Valentino-Devries et al., 2012; Sweeney, 2013; Tutt, 2016; Ajunwa et al., 2016; Barocas and Selbst, 2014; Rudin et al., 2020; Sánchez-Monedero et al., 2020; Wieringa, 2020).

Fairness and justice have always been ripe topics for philosophical debate (Rawls, 2009), and, of course, there are no established rigorous definitions. Nonetheless, the rise of automated decision-making has prompted

the introduction of a number of formal definitions of fairness, and their utility within different contexts is being actively studied and contested (Ruggieri, 2014; Friedler et al., 2016; Kleinberg et al., 2017). To survey a few:

- *Group Fairness* dictates that, for some subset of the population $A \subset \mathcal{X}$, the outcomes given to A must, in an aggregate sense, match those given to all of $\mathcal{X}$ (or $\mathcal{X} \setminus A$). This is also referred to as "statistical parity" or "disparate impact" (Feldman et al., 2015) and is a standard that has been considered in United States legal guidelines (EEOC, 2014). There are variants of group fairness that focus on false-positive rates, "equalized odds," etc. (Hardt et al., 2016)

- Often juxtaposed against group fairness, *Individual Fairness* instead asserts that *similar* individuals must be given *similar* outcomes. The classic formalization (Dwork et al., 2012) assumes we are given some metric space of individuals $(\mathcal{X}, d_{\mathcal{X}})$ and some metric space of outcomes $(\mathcal{Y}, d_{\mathcal{Y}})$: any two individuals x_1, x_2 must have outcomes y_1, y_2 such that $d_{\mathcal{Y}}(y_1, y_2) \leqslant d_{\mathcal{X}}(x_1, x_2)$.

- Some definitions (Datta et al., 2016; Kilbertus et al., 2017; Kusner et al., 2017) rooted in counterfactual reasoning, attempt to tease out a notion of which features *cause* a decision to be made. Fairness is ensured when the decision is caused by features that, a priori, we all agree are suitable for making non-discriminatory decisions.

These definitions introduce a number of challenges: first, it has been shown that the different definitions can be contradictory (Friedler et al., 2016), which contributes, in part, to the ongoing search for *good* definitions of fairness. Second, it is *not* sufficient to ensure fairness (whatever the exact definition is) with respect to some sensitive feature by simply omitting that feature from the inputs to the decision-making program—in practice, there are often strong correlations with other "proxy" features to which

the program has access. In other words, checking that a program is fair and creating programs guaranteed to be fair—either by hand or through machine learning—are important, but very non-trivial problems.[2]

1.2 Contributions and Organization

In Chapter 2 we foray into formally reasoning about learning algorithms themselves to better understand the relation between the training data and the model. We focus on decision-tree learning and present Antidote, a tool that verifies whether test-time classifications are robust to data-poisoning attacks. Antidote leverages the classic program analysis technique of *abstract interpretation*, devising a specialized abstract domain to reason about how a program (in this case, the decision–tree learner) manipulates high–dimensional data tensors. Antidote is able to *prove* some cases where, for example, adding additional training data would not change a particular test instance classification. The content of this chapter is based off work previously presented at PLDI (Drews et al., 2020).

The remainder focuses on algorithmic fairness, which we encode as analysis problems for probabilistic programs. In Chapter 3 we formalize a language for specifying decision-making programs and probabilistic definitions of fairness. We then present FairSquare, a verifier for such probabilistic properties, based upon work presented at OOPSLA (Albarghouthi et al., 2017b). FairSquare evaluates probabilistic definitions of fairness through a novel *symbolic-volume-computation* algorithm that monotonically converges to the *exact* probabilities in the limit, Thus resulting in a *sound and complete fairness verification procedure*. To our knowledge, this is the

[2]There has been an effort within the machine–learning community to adapt loss functions with fairness regularizers, etc, as a means to enforce fairness; some recent works include those by Sagawa et al. (2019); Goel et al. (2020); Mandal et al. (2020). Interestingly, there is also evidence that learning algorithms that attempt to enforce fairness may be more prone to data-poisoning attacks (Chang et al., 2020).

first probabilistic-inference algorithm for arithmetic SMT theories with this expressivity and guarantees.

Chapter 4 then focuses on the dual of this problem: probabilistic synthesis. We present DIGITS, which is able to synthesize a program that satisfies some probabilistic postcondition while also satisfying a quantitative objective: DIGITS searches for a program that is minimally different from some functional specification (e.g., a machine-learned program that does *not* satisfy the postcondition). DIGITS reduces complicated reasoning about how programs manipulate probability distributions over their inputs to reasoning about how programs behave on finite sets of sampled inputs. Remarkably, DIGITS *provably* converges to a near-optimal solution; we provide an analysis of its complexity that draws strong connections with *computational learning theory*. Our evaluation shows that DIGITS is able to automatically synthesize fairness repairs to machine-learned programs that FairSquare verified were unfair, thus providing a mechanism for practitioners to enforce fairness of their models through a general post-processing step. This work is based on a pair of papers presented at CAV (Albarghouthi et al., 2017a; Drews et al., 2019).

Finally, we conclude in Chapter 5 with some closing remarks and by sketching a few directions for future work.

2 VERIFYING DATA-POISONING ROBUSTNESS

In this chapter, we formally reason about the relation between training sets and learned model behavior in the interest of verifying a particular security property: We focus on the problem of verifying data-poisoning robustness for *decision-tree learners*. We choose decision trees because (*i*) they are widely used interpretable models; (*ii*) they are used in industrial models like random forests and XGBoost (Chen and Guestrin, 2016); (*iii*) decision-tree-learning has been shown to be unstable to training-set perturbation (Dwyer and Holte, 2007; Turney, 1995; Li and Belford, 2002; Pérez et al., 2005); and (*iv*) decision-tree-learning algorithms are typically deterministic—e.g., they do not employ stochastic optimization techniques—making them amenable to verification.

The data-poisoning robustness property we consider in this chapter is as follows: Let a *perturbed set* $\Delta(T)$ define a set of neighboring datasets the adversary could have attacked to yield the training set T, which we ultimately use during learning. To determine whether the training algorithm L is robust, we need to measure how the model learned by L varies when modifying the training set. Let us say we have an input example x—e.g., a test example—and its classification label is $M(x) = y$, where $M = L(T)$ is the model learned from the training set T. We say that x is robust to poisoning if and only if for all $T' \in \Delta(T)$, we have $L(T')(x) = y$; in other words, no matter what dataset $T' \in \Delta(T)$ we use to construct the model $M' = L(T')$, we want M' to always return the same classification y on the input x.

We present Antidote, a tool for verifying data-poisoning robustness of decision-tree learners based on *abstract interpretation* (Cousot and Cousot, 1977). At a high level, Antidote takes as input a training set T and an input x, symbolically constructs every tree built by a particular decision-tree learner L on every possible variation of T in $\Delta(T)$, and applies all those

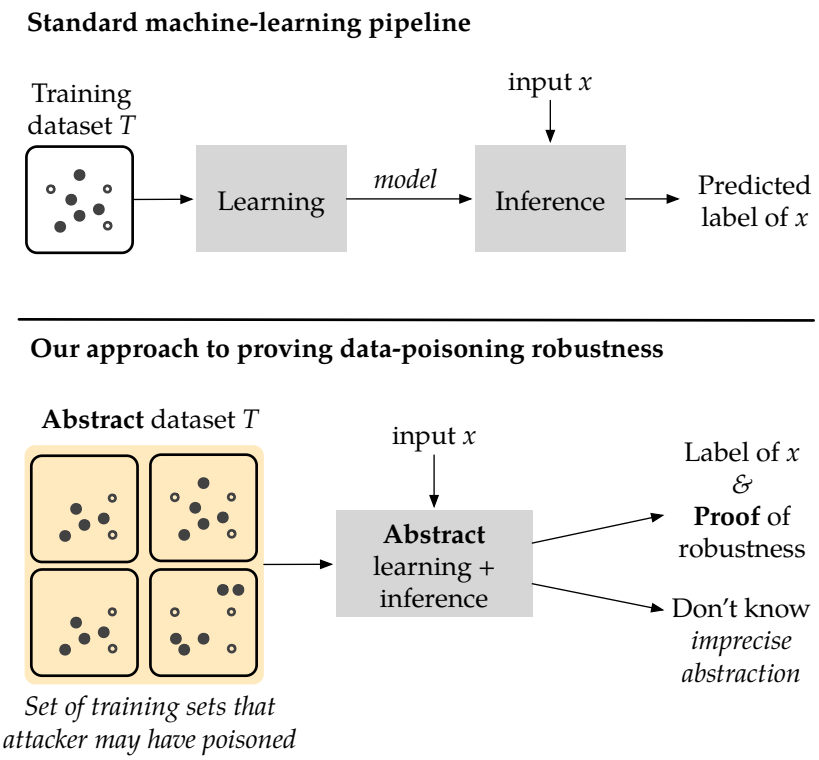


Figure 2.1: High-level overview of our approach (Antidote)

trees to x. If all the trees agree on the label of x, then we know that x is robust to poisoning T. (See Figure 2.1 for an overview.) The layout of this chapter is as follows:

- In Section 2.1 we provide a high-level overview of Antidote.
- In Section 2.2 we formally define the poisoning robustness problem and describe a *trace-based view* of decision-tree learning as a stand-alone algorithm (facilitating a simpler analysis).
- In Sections 2.3 and 2.4 we describe an *abstract domain that concisely encodes sets of perturbed datasets* and the abstract transformers necessary to verify robustness of the decision-tree learner.

- In Section 2.5 we present an evaluation of Antidote on various datasets, including a fragment of MNIST (LeCun et al.), showing that Antidote can prove poisoning robustness for all datasets in cases where an enumeration approach would be doomed to fail.

Proofs of theorems stated throughout this chapter can be found in Appendix B.1. This chapter is based on the work of Drews et al. (2020).

2.1 Overview

In this section, we give an overview of decision-tree learning, the poisoning-robustness problem, and motivate our abstraction-based proof technique.

Decision-Tree Learning. Consider the dataset T_{bw} at the top of Figure 2.2. It is comprised of 13 elements with a single numerical feature. Each element is labeled as a white (empty) or black (solid) circle. We use x to denote the feature value of each element. Our goal is to construct a decision tree that classifies a given number into white or black.

For simplicity, we assume that we can only build trees of depth 1, like the one shown at the bottom Figure 2.2. At each step of building a decision tree, the learning algorithm is looking for a predicate φ with the best score, with the goal of splitting the dataset into two pieces with *least diversity*, i.e., most elements have the same class (formally defined usually using a notion of entropy). This is what we see in our example: using the predicate $x \leqslant 10$, we split the dataset into two sets, one that is mostly white (left) and one that is completely black (right). This is the best split we can have for our data, assuming we can only pick predicates of the form $x \leqslant c$, for an integer c.[1]

[1] Note that, while the set of predicates $x \leqslant c$ is infinite, for this dataset (and in general for any dataset), there exists only finitely many inequivalent predicates—e.g., $x \leqslant 4$ and $x \leqslant 5$ split the dataset into the same two sets.

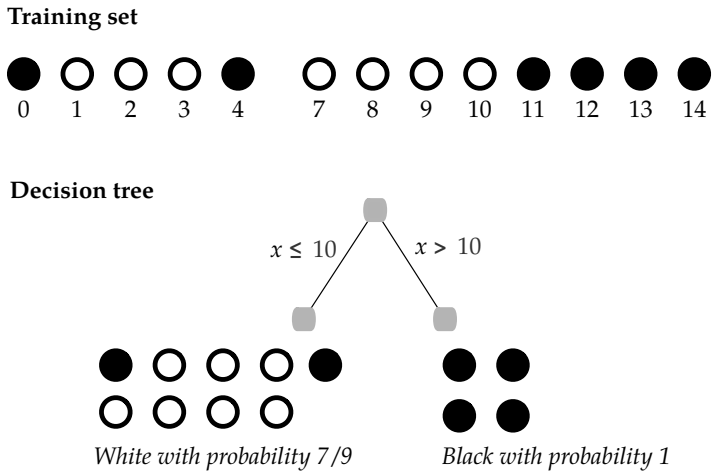


Figure 2.2: Illustrative decision-tree learning example

Given a new element for a classification, we check if it is $\leqslant 10$, in which case we say it is white with probability $7/9$—i.e., the fraction of white elements such that $\leqslant 10$. Otherwise, if the element is > 10, we say it is black with probability 1.

Data-Poisoning Robustness. Imagine we want to classify an input x but want to make sure the classification would not have changed had the training data been slightly different. For example, maybe some percentage of the data was maliciously added by an attacker to sway the learning algorithm, a problem known as *data poisoning*. Our goal is to check whether the classification of x is robust to data poisoning.

A Naïve Approach. Consider our running example and imagine we want to classify the number 5. Additionally, we want to prove that *removing up to two elements* from the training set would not change the classification of 5—i.e., we assume that up to ~15% (or 2/13) of the dataset is contributed maliciously. The naïve way to do this is to consider every possible training dataset with up to two elements removed and retrain the decision tree.

If all trees classify the input 5 as white, the classification is robust to this level of poisoning.

Unfortunately, this approach is intractable. Even for our tiny example, we have to train 92 trees $(\binom{13}{2} + \binom{13}{1} + 1)$. For a dataset of 1000 elements and a poisoning of up to 10 elements, we have $\sim 10^{23}$ possibilities.

An Abstract Approach. Our approach to efficiently proving poisoning robustness exploits a number of insights. First, we can perform decision-tree learning *abstractly* on a *symbolic set of training sets*, without having to deal with a combinatorial explosion. The idea is that the operations in decision-tree learning, e.g., selecting a predicate and splitting the dataset, do not need to look at every concrete element of a dataset, but at aggregate statistics (counts).

Recall our running example in Figure 2.2. Let us say that up to two elements have been removed. No matter what two elements you choose, the predicate $x \leqslant 10$ remains one that gives *a* best split for the dataset. In cases of ties between predicates, our algorithm abstractly represents all possible splits. For each predicate, we can symbolically compute best- and worst-case scores in the presence of poisoning as an *interval*. Similarly, we can also compute an interval that overapproximates the set of possible classification probabilities. For instance, in the left branch of the decision-tree, the probability will be $[0.71, 1]$ instead of 0.78 (or $7/9$). The best case probability of 1 is when we drop the black points 0 and 4; the worst-case probability of 0.71 (or $5/7$) is when we drop any two white points.

The next insight that enables our approach is that we *do not need to explicitly build the tree*. Since our goal is to prove robustness of a single input point, which effectively takes a single trace through the tree, we mainly need to keep track of the abstract training sets as they propagate along those traces. This insight drastically simplifies our approach; otherwise, we would need to somehow abstractly represent sets of elements of a tree data structure, a non-trivial problem in program analysis.

Abstraction and Imprecision. We note that our approach is sound but necessarily incomplete; that is, when our approach returns "robust" the answer is correct, but there are robust instances for which our approach will not be able to prove robustness. The are numerous sources of imprecision due to overapproximation, for example, we use the *intervals domain* (or disjunctive intervals) to capture real-valued entropy calculations of different training set splits, as well as the final probability of classification.

2.2 Poisoning and Decision Tree Learning

In this section, we begin by formally defining the *data-poisoning-robustness problem*. Then, we present a *trace-based* view of decision-tree learning, which will pave the way for a poisoning-robustness proof technique.

The Poisoning Robustness Problem

In a typical supervised learning setting, we are given a learning algorithm L and a training set $T \subseteq \mathcal{X} \times \mathcal{Y}$ comprised of elements of some set $\mathcal{X}$, each with its classification label from a finite set of classes $\mathcal{Y}$. Applying L to T results in a classifier (or model): $M : \mathcal{X} \to \mathcal{Y}$. For now, we assume that both the learning algorithm L and the models it learns are deterministic functions.[2]

A *perturbed set* $\Delta(T) \subseteq 2^{\mathcal{X} \times \mathcal{Y}}$ defines a set of possible *neighboring* datasets of T. Our robustness definitions are relative to some given perturbation Δ. (In Section 2.3, we define a specific perturbed set that captures a particular form of data poisoning.)

Definition 2.1 (Poisoning Robustness)**.** *Fix a learning algorithm* L*, a training set* T*, and let* $\Delta(T)$ *be a perturbed set. Given an element* $x \in \mathcal{X}$*, we say that* x *is*

[2]Our approach, however, needs to handle non-determinism in decision-tree learning, which arises when breaking ties for choosing predicates with equal scores and choosing labels for classes with equal probabilities.

robust to poisoning T *if and only if*

$$\forall T' \in \Delta(T).\ L(T')(x) = L(T)(x)$$

When T *and* Δ *are clear from context, we will simply say that* x *is robust.*

In other words, no matter what dataset $T' \in \Delta(T)$ we use to construct the model $M = L(T')$, we want M always to return the same classification for x.

Example 2.2. *Imagine we suspect that an attacker has contributed 10 training points to* T*, but we do not know which ones. We can define* $\Delta(T)$ *to be* T *as well as every subset of* T *of size* $|T| - 10$*. If an input* x *is robust for this definition of* $\Delta(T)$*, then no matter whether the attacker has contributed 10 training items or not, the classification of* x *does not change.*

Decision Trees: A Trace-Based View

We now formally define decision trees. We will formalize a tree as the *set of traces* from the root to each of the leaves. As we will see, this trace-based view will help enable our proof technique. The idea of representing an already-learned decision tree as a set of traces is not new and has often been explored in the context of extracting interpretable rules from decision trees (Quinlan, 1987).

A decision tree R is a finite set of traces, where each trace is a tuple (σ, y) such that σ is a sequence of Boolean predicates and $y \in \mathcal{Y}$ is the classification.

Semantically, a tree R is a function in $\mathcal{X} \to \mathcal{Y}$. Given an input $x \in \mathcal{X}$, applying $R(x)$ results in a classification y from the trace $(\sigma, y) \in R$ where x satisfies all the predicates in the sequence $\sigma = [\varphi_1, \ldots, \varphi_n]$, that is, $\bigwedge_{i=1}^{n} x \models \varphi_i$ is true. We say a tree R is *well-formed* if for every $x \in \mathcal{X}$ there

exists exactly one trace $(\sigma, y) \in R$ such that x satisfies all predicates in σ. In the following we assume all trees are well-formed.

Example 2.3 (Decision tree traces)**.** *Consider the decision-tree in Figure 2.2. It contains two traces, each with a sequence of predicates containing a single predicate:* $([x \leqslant 10], \textit{white})$ *and* $([x > 10], \textit{black})$.

Decision-Tree Learning: A Trace-Based View

We now present a simple decision-tree learning algorithm, DTrace. Then, in Section 2.3, we abstractly interpret DTrace with the goal of proving poisoning robustness.

One of our key insights is that we do not need to explicitly represent the learned trees (i.e., the set of all traces), since our goal is to prove robustness of a *single input* point, which effectively takes a *single trace* through the tree. Therefore, in this section, we will define a *trace-based decision-tree learning algorithm*. This is inspired by standard algorithms—like CART (Breiman, 2017), ID3 (Quinlan, 1986), and C4.5 (Quinlan, 1993)—but *it is input-directed, in the sense that it only builds the trace of the tree that a given input* x *will actually traverse.*

A Trace-Based Learner. Our trace-based learner DTrace is shown in Figure 2.3. It takes a training set T and an input x and computes the trace traversed by x in the tree learned on T. Intuitively, if we compute the set of all traces $\mathsf{DTrace}(T, x)$ for each $x \in T$, we get the full tree, the one that we would have traditionally learned for T.

The learner DTrace repeats two core operations: (*i*) selecting a predicate φ with which to split the dataset (using bestSplit) and (*ii*) removing elements of the training set based on whether they satisfy the predicate φ (depending on x, using filter).[3] The number of times the loop is repeated

[3]Note that (*ii*) is what distinguishes our trace-based learning from conventional learning of a full tree. (*i*) selects predicates that would comprise the tree, while (*ii*) directs

Input: training set T and input $x \in \mathcal{X}$
Initialize: $\varphi \leftarrow \diamond$, $\sigma \leftarrow$ empty trace
repeat d times
  if $\mathsf{ent}(\mathsf{T}) = 0$ **then** return
  $\varphi \leftarrow \mathsf{bestSplit}(\mathsf{T})$
  if $\varphi = \diamond$ **then** return
  $\mathsf{T} \leftarrow \mathsf{filter}(\mathsf{T}, \varphi, x)$
  if $x \models \varphi$ **then** $\sigma \leftarrow \sigma\varphi$ **else** $\sigma \leftarrow \sigma\neg\varphi$
Output: $\mathrm{argmax}_{i \in [1,k]}\, p_i$, where $\mathsf{cprob}(\mathsf{T}) = \langle p_1, \ldots, p_k \rangle$

Figure 2.3: Trace-based decision-tree learner DTrace

(d) is the maximum depth of the trace that is constructed. Throughout, we assume a fixed set of classes $\mathcal{Y} = \{1, \ldots, k\}$.

The mutable state of DTrace is the triple $(\mathsf{T}, \varphi, \sigma)$:

- T is the training set, which will keep getting refined (by dropping elements) as the trace is constructed.
- φ is the most recent predicate along the trace, which is initially undefined (denoted by $\diamond$).
- σ is the sequence of predicates along the trace, which is initially empty.

Predicate Selection. We assume that DTrace is equipped with a finite set of predicates Φ with which it can construct a decision-tree classifier; each predicate in Φ is a Boolean function in $\mathcal{X} \to \mathbb{B}$.

$\mathsf{bestSplit}(\mathsf{T})$ computes a predicate $\varphi^\star \in \Phi$ that splits the current dataset T—usually minimizing a notion of entropy. Ideally, the learning algorithm would consider every possible sequence of predicates to partition a dataset in order to arrive at an optimal classifier. For efficiency, a decision-tree-learning algorithms does this greedily: it selects the best predicate it can

us to recurse *only* along the path that the specific x would take, as opposed to recursing down both (and without affecting how predicates in the tree are selected).

find for a single split and moves on to the next split. To perform this greedy choice, it measures how diverse the two datasets resulting from the split are. We formalize this below:

We use $\mathsf{T}{\downarrow_\varphi}$ to denote the subset of T that satisfies φ, i.e.,

$$\mathsf{T}{\downarrow_\varphi} = \{(x, y) \in \mathsf{T} \mid x \models \varphi\}$$

Let Φ' be the set of all predicates that do not trivially split the dataset: $\Phi' = \{\varphi \in \Phi \mid \mathsf{T}{\downarrow_\varphi} \neq \emptyset \wedge \mathsf{T}{\downarrow_\varphi} \neq \mathsf{T}\}$. Finally, $\mathsf{bestSplit}(\mathsf{T})$ is defined as follows:

$$\mathsf{bestSplit}(\mathsf{T}) = \operatorname*{argmin}_{\varphi \in \Phi'} \mathsf{score}(\mathsf{T}, \varphi)$$

where $\mathsf{score}(\mathsf{T}, \varphi) = |\mathsf{T}{\downarrow_\varphi}| \cdot \mathsf{ent}(\mathsf{T}{\downarrow_\varphi}) + |\mathsf{T}{\downarrow_{\neg\varphi}}| \cdot \mathsf{ent}(\mathsf{T}{\downarrow_{\neg\varphi}})$. Informally, $\mathsf{bestSplit}(\mathsf{T})$ is the predicate that splits T into two sets with the lowest entropy, as defined by the function ent shown in Figure 2.4. Formally, ent computes *Gini impurity*, which is used, for instance, in the CART algorithm (Breiman, 2017). Note that if $\Phi' = \emptyset$, we assume $\mathsf{bestSplit}(\mathsf{T})$ is undefined (returns $\diamond$). Further, if multiple predicates are possible values of $\mathsf{bestSplit}(\mathsf{T})$, we assume one is returned nondeterministically. Later, in Section 2.3, our abstract interpretation of DTrace will actually capture all possible predicates in the case of a tie.

Example 2.4. *Recall our example from Section 2.1 and Figure 2.2. For readability, we use* T *instead of* T_{bw} *for the name of the dataset. Let us compute* $\mathsf{score}(\mathsf{T}, \varphi)$*, where* φ *is* $x \leqslant 10$*. We have* $|\mathsf{T}{\downarrow_\varphi}| = 9$ *and* $|\mathsf{T}{\downarrow_{\neg\varphi}}| = 4$*. For the classification probabilities, defined by* cprob *(Figure 2.4), we have* $\mathsf{cprob}(\mathsf{T}{\downarrow_\varphi}) = \langle 7/9, 2/9\rangle$ *and* $\mathsf{cprob}(\mathsf{T}{\downarrow_{\neg\varphi}}) = \langle 0, 1\rangle$ *assuming the first element represents white classification; e.g., in* $\mathsf{T}{\downarrow_\varphi}$*, there's a* $7/9$ *chance of being classified as white. For* ent*, we have* $\mathsf{ent}(\mathsf{T}{\downarrow_\varphi}) \approx 0.35$ *and* $\mathsf{ent}(\mathsf{T}{\downarrow_{\neg\varphi}}) = 0$*. Since* $\mathsf{T}{\downarrow_\varphi}$ *is solely composed of black points, its Gini impurity is 0.*

The score of $x \leqslant 10$ *is therefore* ~3.1*. For the predicate* $x \leqslant 11$*, we get the higher (worse) score of* ~3.2*, as it generates a more diverse split.*

$$\mathsf{ent}(T) = \sum_{i=1}^{k} p_i(1-p_i), \quad \text{where } \mathsf{cprob}(T) = \langle p_1, \ldots, p_k \rangle$$
$$\mathsf{cprob}(T) = \left\langle \frac{|\{(x,y) \in T \mid y = i\}|}{|T|} \right\rangle_{i \in [1,k]}$$

Figure 2.4: Auxiliary operator definitions. ent is Gini impurity; cprob returns a vector of classification probabilities, one element for each class $i \in [1, k]$.

Filtering the Dataset. The operator filter removes elements of T that evaluate differently than x on φ. Formally,

$$\mathsf{filter}(T, \varphi, x) = \begin{cases} T{\downarrow_{\varphi}} & \text{if } x \models \varphi \\ T{\downarrow_{\neg\varphi}} & \text{otherwise} \end{cases}$$

Learner Result. When DTrace terminates in a state $(T_r, \varphi_r, \sigma_r)$, we can read the classification of x as the class i with the highest number of training elements in T_r.

Using cprob, in Figure 2.4, we compute the probability of each class i for a training set T as a vector of probabilities. Finally, DTrace returns the class with the highest probability:

$$\operatorname*{argmax}_{i \in [1,k]} p_i \qquad \text{where } \mathsf{cprob}(T_r) = \langle p_1, \ldots, p_k \rangle$$

As before, in case of a tie in probabilities, we assume a nondeterministic choice.

Example 2.5. *Following the computation from Example 2.4, DTrace*$(T, 18)$ *terminates in state* $(T{\downarrow_{x>10}},\ x \leqslant 10,\ [x > 10])$. *Point 18 is associated with the trace* $[x > 10]$ *and is classified as black because* $\mathsf{cprob}(T{\downarrow_{x>10}}) = \langle 0, 1 \rangle$.

2.3 Abstractions of Poisoned Semantics

In this section, we begin by defining a data-poisoning model in which an attacker contributes a number of malicious training items. Then, we demonstrate how to apply the trace-based learner DTrace to *abstract sets of training sets*, allowing us to efficiently prove poisoning-robustness.

The n-Poisoning Model

For our purposes, we will consider a poisoning model where the attacker has contributed up to n elements of the training set—we call it n-*poisoning*. Formally, given a training set T and a natural number $n \leqslant |T|$, we define the following perturbed set:

$$\Delta_n(T) = \{T' \subseteq T \ : \ |T \setminus T'| \leqslant n\}$$

In other words, $\Delta_n(T)$ captures every training set the attacker could have possibly started from to arrive at T.

This definition of dataset poisoning matches many settings studied in the literature (Chen et al., 2017; Steinhardt et al., 2017; Xiao et al., 2015a). The idea is that an attacker has contributed a number of malicious data points into the training set to influence the resulting classifier. For example, Chen et al. (2017) consider poisoning a facial recognition model to enable bypassing authentication, and Xiao et al. (2015a) consider poisoning a malware detector to allow the attacker to install malware.

We do not know which n points in T are the malicious ones, or if there are malicious points at all. Thus, the set $\Delta_n(T)$ captures every possible subset of T where we have removed up to n (potentially malicious) elements. Our goal is to prove that our classification is robust to up to n possible poisoned points added by the attacker. So if we try every possible dataset in $\Delta_n(T)$ and they all result in the same classification on x, then x

is robust regardless of the attacker's potential contribution.

Observe that $|\Delta_n(T)| = \sum_{i=1}^{n} \binom{|T|}{i}$. So even for relatively small datasets and number n, the set of possibilities is massive, e.g., for MNIST-1-7 dataset (Section 2.5), for $n = 50$, we have about 10^{141} possible training sets in $\Delta_n(T)$.

Abstract Domains for Verifying n-Poisoning

Our goal is to efficiently evaluate DTrace on an input x for all possible training datasets in $\Delta_n(T)$. If all of them yield the same classification y, then we know that x is a robust input. Our insight is that we can abstractly interpret DTrace on a symbolic set of training sets without having to fully expand it into all of its possible concrete instantiations. This allows us to train on an enormous number of datasets, which would be impossible via enumeration.

Recall that the state of DTrace is (T, φ, σ); for our purposes, we do not have to consider the sequence of predicates σ, as we are only interested in the final classification, which is a function of T. In this section, we present the *abstract domains* for each component of the learner's state.

Abstract Training Sets. Abstracting training sets is the main novelty of our technique. We use the abstract element $\langle T', n' \rangle$ to denote a set of training sets and it captures the definition of $\Delta_{n'}(T')$: For every training set T' and number n', the concretization function is $\gamma(\langle T', n' \rangle) = \Delta_{n'}(T')$. Therefore, we have that initially the *abstraction* function $\alpha(\Delta_n(T)) = \langle T, n \rangle$ is precise. Note that an abstract element $\langle T', n' \rangle$ succinctly captures a large number of concrete sets, $\Delta_{n'}(T')$. Further, all operations we perform on $\langle T', n' \rangle$ will only modify T' and n', without resorting to concretization.

We can define an *efficient* join operation on two elements in the abstract domain[4] as follows:

[4]Elements in the domain are ordered so that $\langle T_1, n_1 \rangle \sqsubseteq \langle T_2, n_2 \rangle$ if and only if $T_1 \subseteq T_2 \wedge n_1 \leqslant n_2 - |T_2 \setminus T_1|$. In the text, we define the concretization function, a special case

Definition 2.6 (Joins). *Given two training sets* T_1, T_2 *and* $n_1, n_2 \in \mathbb{N}$, $\langle T_1, n_1\rangle \sqcup \langle T_2, n_2\rangle \coloneqq \langle T', n'\rangle$ *where* $T' = T_1 \cup T_2$ *and* $n' = \max(|T_1 \setminus T_2| + n_2, |T_2 \setminus T_1| + n_1)$.

Notice that the join of two sets is an overapproximation of the union of the two sets. The following proposition formalizes the soundness of this operation:

Proposition 2.7. *For any* T_1, T_2, n_1, n_2, *the following holds:*

$$\gamma(\langle T_1, n_1\rangle) \cup \gamma(\langle T_2, n_2\rangle) \subseteq \gamma(\langle T_1, n_1\rangle \sqcup \langle T_2, n_2\rangle).$$

Example 2.8. *For any training set* T_1, *if we consider the abstract sets* $\langle T_1, 2\rangle$ *and* $\langle T_1, 3\rangle$, *because the second set represents strictly more concrete training sets, we have*

$$\langle T_1, 2\rangle \sqcup \langle T_1, 3\rangle = \langle T_1, 3\rangle$$

Now consider the training set $T_2 = \{x_1, x_2\}$. *We have*

$$\langle T_2, 2\rangle \sqcup \langle T_2 \cup \{x_3\}, 2\rangle = \langle T_2 \cup \{x_3\}, 3\rangle$$

Notice how the join increased the poisoned elements from 2 to 3 to accommodate for the additional element x_3.

Abstract Predicates and Numeric Values. When abstractly interpreting what predicates the learner might choose for different training sets, we will need to abstractly represent sets of possible predicates. Simply, a set of predicates is abstracted *precisely* as the corresponding set of predicates Ψ—i.e., for every set Ψ, we have $\alpha(\Psi) = \Psi$ and $\gamma(\Psi) = \Psi$. Moreover,

of the abstraction function, and the join operation; note that we do not require an explicit meet operation for the purposes of this chapter—although one is well-defined:

$$\begin{aligned}\langle T_1, n_1\rangle \sqcap \langle T_2, n_2\rangle \coloneqq \text{if } |T_1 \setminus T_2| > n_1 \vee |T_2 \setminus T_1| > n_2 \text{ then } \bot \\ \text{else } \langle T_1 \cap T_2, \min(n_1 - |T_1 \setminus T_2|, n_2 - |T_2 \setminus T_1|)\rangle\end{aligned}$$

$\Psi_1 \sqcup \Psi_2 = \Psi_1 \cup \Psi_2$. For certain operations, it will be handy for Ψ to contain a special null predicate $\diamond$.

When abstractly interpreting numerical operations, like cprob and ent, we will need to abstract sets of numerical values. We do so using the standard *intervals* abstract domain (denoted $[l, u]$). For instance, $\alpha(\{0.2, 0.4, 0.6\}) = [0.2, 0.6]$ and $\gamma([0.2, 0.6]) = \{x \mid 0.2 \leqslant x \leqslant 0.6\}$. The join of two intervals is defined as $[l_1, u_1] \sqcup [l_2, u_2] = [\min(l_1, l_2), \max(r_1, r_2)]$. Interval arithmetic follows the standard definitions and we thus elide it here.[5]

Abstract Learner DTrace$^\#$

We are now ready to define an abstract interpretation of the semantics of our decision-tree learner, denoted DTrace$^\#$.

Abstract Domain. Recall that the state of DTrace is (T, φ, σ); for our purposes, we do not have to consider the sequence of predicates σ, as we are only interested in the final classification, which is a function of T. Using the domains described prior, at each point in the learner, our abstract state is a pair $(\langle T', n' \rangle, \Psi')$ (i.e., in the product abstract domain) that tracks the current set of training sets and the current set of possible most recent predicates the algorithm has split on (for all considered training sets).

When verifying n-poisoning for a training set T, the initial abstract state of the learner will be the pair $(\langle T, n \rangle, \{\diamond\})$. In the rest of the section, we define the abstract semantics (i.e., our abstract transformers) for all the operations performed by DTrace$^\#$. For operations that only affect one element of the state, we assume that the other component is left unchanged.

[5] While we choose intervals as our numerical abstract domain in this chapter, any numerical abstract domain could be used.

Abstract Semantics of Auxiliary Operators

We will begin by defining the abstract semantics of the auxiliary operations in the algorithm before proceeding to the core operations, filter and bestSplit. This is because the auxiliary operators are simpler and highlight the nuances of our abstraction.

Let us begin by considering $\langle T, n\rangle\downarrow^{\#}_{\varphi}$, which is the abstract analog of $T\downarrow_{\varphi}$.

$$\langle T, n\rangle\downarrow^{\#}_{\varphi} \coloneqq \langle T\downarrow_{\varphi}, \min(n, |T\downarrow_{\varphi}|)\rangle \tag{2.1}$$

Simply, it removes elements not satisfying φ from T; since the size of $T\downarrow_{\varphi}$ can go below n, we take the minimum of the two.

Proposition 2.9. *Let* $T' \in \gamma(\langle T, n\rangle)$. *For any predicate* φ, *we have* $T'\downarrow_{\varphi} \in \gamma(\langle T, n\rangle\downarrow^{\#}_{\varphi})$.

Now consider $\mathsf{cprob}(T)$, which returns a vector of probabilities for different classes. Its abstract version returns an interval for each probability, denoting the lower and upper bounds based on the training sets in the abstract set:[6]

$$\mathsf{cprob}^{\#}(\langle T, n\rangle) \coloneqq \left\langle \frac{[\max(0, c_i - n), c_i]}{[|T| - n, |T|]} \right\rangle_{i\in[1,k]}$$

[6]Note that this transformer can be more precise: for example, the interval division as written is not guaranteed to be a subset of $[0, 1]$, despite the fact that all concrete values would be. Throughout this section, many of the transformers are simply the "natural" lifting of numerical arithmetic to interval arithmetic; while this may not be optimal, we do so to make it easier to see the correctness of the approach (and to make proofs and implementation straightforward).

In the case of $\mathsf{cprob}^{\#}$, we can compute the optimal transformer inexpensively: it is equivalent to write that $\mathsf{cprob}(T)$ computes, for each class $i \in [1, k]$, the *average* of the multiset $S_i = [\text{if } y = i \text{ then } 1 \text{ else } 0 \mid (x, y) \in T]$. We can then have $\mathsf{cprob}^{\#}(\langle T, n\rangle)$ perform a similar computation for each component: let L_i denote the m-many least elements of S_i, and let U_i denote the m-many greatest elements of S_i, where $m = |T| - n$. These L_i and U_i exhibit extremal behavior of averaging, so we can directly compute the endpoints of the interval assigned to each class as $[\frac{1}{m}\sum_{b\in L_i} b, \frac{1}{m}\sum_{b\in U_i} b]$.

Note that our implementation used for the evaluation (Section 2.5) *does* employ this optimal transformer for $\mathsf{cprob}^{\#}$, while the other transformers match what is presented.

where $c_i = |\{(x, i) \in T\}|$. In other words, for each class i, we need to consider the best- and worst-case probability based on removing n elements from the training set, as denoted by the denominator and the numerator. Note that in the corner case where $n = |T|$, we set $\mathsf{cprob}^{\#}(\langle T, n\rangle) = \langle [0,1]\rangle_{i \in [1,k]}$.

Proposition 2.10. *Let* $T' \in \gamma(\langle T, n\rangle)$. *Then,*

$$\mathsf{cprob}(T') \in \gamma\big(\mathsf{cprob}^{\#}(\langle T, n\rangle)\big)$$

where $\gamma\big(\mathsf{cprob}^{\#}(\langle T, n\rangle)\big)$ *is the set of all possible probability vectors in the vector of intervals.*

Example 2.11. *Consider the training set on the left side of the tree in Figure 2.2; call it* T_ℓ. *It has 7 white elements and 2 black elements.* $\mathsf{cprob}(T_\ell) = \langle 7/9, 2/9\rangle$, *where the first element is the white probability.* $\mathsf{cprob}^{\#}(\langle T_\ell, 2\rangle)$ *produces the vector* $\langle [5/9, 1], [0, 2/7]\rangle$. *Notice the loss of precision in the lower bound of the first element. If we remove two white elements, we should get a probability of* $5/7$, *but the interval domain cannot capture the relation between the numerator and denominator in the definition of* $\mathsf{cprob}^{\#}$.

The abstract version of the Gini impurity is identical to the concrete one, except that it performs interval arithmetic:

$$\mathsf{ent}^{\#}(T) = \sum_{i=1}^{k} \iota_i([1,1] - \iota_i), \quad \text{where } \mathsf{cprob}^{\#}(T) = \langle \iota_1, \ldots, \iota_k\rangle$$

Each term ι_i denotes an interval.

Abstract Semantics of filter

We are now ready to define the abstract version of filter. Since we are dealing with abstract training sets, as well as a set of predicates Ψ, we need

to consider for each $\varphi \in \Psi$ all cases where $x \models \varphi$ or $x \models \neg\varphi$, and take the join of all the resulting training sets (Definition 2.6). Let

$$\Psi_x = \{\varphi \in \Psi \mid x \models \varphi\} \text{ and } \Psi_{\neg x} = \{\varphi \in \Psi \mid x \models \neg\varphi\}$$

Then,

$$\mathsf{filter}^\#(\langle T, n\rangle, \Psi, x) \coloneqq \left(\bigsqcup_{\varphi\in\Psi_x} \langle T, n\rangle\downarrow^\#_\varphi\right) \sqcup \left(\bigsqcup_{\varphi\in\Psi_{\neg x}} \langle T, n\rangle\downarrow^\#_{\neg\varphi}\right)$$

Proposition 2.12. *Let* $T' \in \gamma(\langle T, n\rangle)$ *and* $\varphi' \in \Psi$*. Then,*

$$\mathsf{filter}(T', \varphi', x) \in \gamma\big(\mathsf{filter}^\#(\langle T, n\rangle, \Psi, x)\big)$$

Example 2.13. *Consider the full dataset* T_{bw} *from Figure 2.2. For readability, we write* T *instead of* T_{bw} *in the example. Let* x *denote the input with numerical feature 4, and let* $\Psi = \{x \leqslant 10\}$*. First, note that because* $\Psi_{\neg x}$ *is the empty set, the right-hand side of the result of applying the* $\mathsf{filter}^\#$ *operator will be the bottom element* $\langle\emptyset, 0\rangle$ *(i.e., the identity element for* $\sqcup$*). Then,*

$$\begin{aligned}
\mathsf{filter}^\#(\langle T, 2\rangle, \Psi, x) &= \langle T, 2\rangle\downarrow^\#_{x\leqslant 10} \sqcup \langle\emptyset, 0\rangle && (\textit{def. of } \mathsf{filter}^\#)\\
&= \langle T\downarrow_{x\leqslant 10}, 2\rangle \sqcup \langle\emptyset, 0\rangle && (\textit{def. of } \langle T, n\rangle\downarrow^\#_\varphi)\\
&= \langle T\downarrow_{x\leqslant 10}, 2\rangle && (\textit{def. of } \sqcup).
\end{aligned}$$

Abstract Semantics of bestSplit

We are now ready to define the abstract version of bestSplit. We begin by defining $\mathsf{bestSplit}^\#$ without handling trivial predicates, then we refine our definition.

Minimal Intervals. Recall that in the concrete case, bestSplit returns a predicate that minimizes the function $\mathsf{score}(T, \varphi)$. To lift bestSplit to the abstract semantics, we define $\mathsf{score}^\#$, which returns an interval, and what

it means to be a *minimal* interval—i.e., the interval corresponding to the abstract minimal value of the objective function $\mathsf{score}^{\#}(\mathsf{T}, \varphi)$.

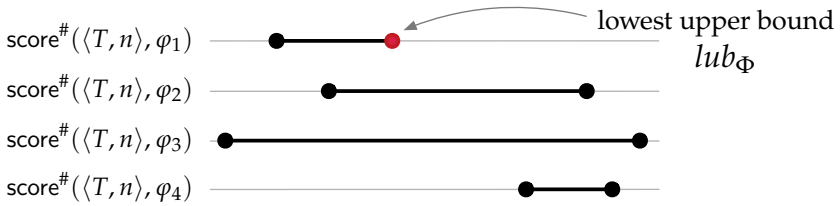
Lifting $\mathsf{score}(\mathsf{T}, \varphi)$ to $\mathsf{score}^{\#}(\langle \mathsf{T}, n \rangle, \varphi)$ can be done using the sound transformers for the intermediary computations:

$$\begin{aligned}\mathsf{score}^{\#}(\langle \mathsf{T}, n \rangle, \varphi) \coloneqq\ & |\langle \mathsf{T}, n \rangle \downarrow^{\#}_{\varphi}| \cdot \mathsf{ent}^{\#}(\langle \mathsf{T}, n \rangle \downarrow^{\#}_{\varphi}) \\ & + |\langle \mathsf{T}, n \rangle \downarrow^{\#}_{\neg\varphi}| \cdot \mathsf{ent}^{\#}(\langle \mathsf{T}, n \rangle \downarrow^{\#}_{\neg\varphi})\end{aligned}$$

where $|\langle \mathsf{T}, n \rangle| \coloneqq [|\mathsf{T}| - n, |\mathsf{T}|]$.

However, given a set of predicates Φ, $\mathsf{bestSplit}^{\#}$ must return the ones with the minimal scores. Before providing the formal definition, we illustrate the idea with an example.

Example 2.14. *Imagine a set of predicates* $\Phi = \{\varphi_1, \varphi_2, \varphi_3, \varphi_4\}$ *with the following intervals for* $\mathsf{score}^{\#}(\langle \mathsf{T}, n \rangle, \varphi_i)$.



Notice that φ_1 *has the lowest upper bound for score (denoted in red and named* lub_Φ*). Therefore, we call* $\mathsf{score}^{\#}(\langle \mathsf{T}, n \rangle, \varphi_1)$ *the* minimal interval *with respect to* Φ. $\mathsf{bestSplit}^{\#}$ *returns all the predicates whose scores* overlap *with the minimal interval* $\mathsf{score}^{\#}(\langle \mathsf{T}, n \rangle, \varphi_1)$*, which in this case are* φ_1*,* φ_2*, and* φ_3*. This is because there is a chance that* φ_1*,* φ_2 *and* φ_3 *are indeed the predicates with the best score, but our abstraction has lost too much precision for us to tell conclusively.*

Let lb/ub be functions that return the lower/upper bound of an interval. First, we define the lowest upper bound among the abstract scores of

the predicates in Φ as

$$\mathrm{lub}_\Phi = \min_{\varphi \in \Phi} \mathrm{ub}(\mathsf{score}^\#(\langle T, n\rangle, \varphi))$$

We can now define the set of predicates whose score overlaps with the minimal interval as:

$$\{\varphi \in \Phi \mid \mathrm{lb}(\mathsf{score}^\#(\langle T, n\rangle, \varphi)) \leqslant \mathrm{lub}_\Phi\}$$

Dealing with Trivial Predicates. Our formulation above considers the full set of predicates, Φ. To be more faithful to the concrete semantics, $\mathsf{bestSplit}^\#$ needs to eliminate trivial predicates from this set. In the concrete case, we only considered φ as a possible best split if φ performed a non-trivial split on T, which we denoted $\varphi \in \Phi'$. (Recall that a trivial split of T is one that returns $\emptyset$ or T.)

This is a little tricky to lift to our abstract case, since a predicate φ could non-trivially split *some* of the concrete datasets but not others. We lift the set Φ' in two ways:

- *Universal predicates*: the predicates that are non-trivial splits *for all* concrete training sets in $\gamma(\langle T, n\rangle)$[7]

$$\Phi_\forall = \{\varphi \in \Phi \mid \emptyset \notin \gamma(\langle T, n\rangle\!\downarrow^\#_\varphi) \wedge \emptyset \notin \gamma(\langle T, n\rangle\!\downarrow^\#_{\neg\varphi})\}$$

- *Existential predicates*: the predicates that are non-trivial splits *for at least one* concrete training set in $\gamma(\langle T, n\rangle)$

$$\Phi_\exists = \{\varphi \in \Phi \mid \langle\emptyset, \cdot\rangle \neq \langle T, n\rangle\!\downarrow^\#_\varphi \wedge \langle\emptyset, \cdot\rangle \neq \langle T, n\rangle\!\downarrow^\#_{\neg\varphi}\}$$

[7] Note that checking $\emptyset \notin \gamma(\langle T, n\rangle)$ is equivalent to checking $n \neq |T|$.

Finally, the definition of $\mathsf{bestSplit}^\#$ considers two cases:

$$\mathsf{bestSplit}^\#(\langle T, n\rangle) \coloneqq \text{if } \Phi_\forall = \emptyset \text{ then } \Phi_\exists \cup \{\diamond\} \text{ else} \\ \{\varphi \in \Phi_\exists : \mathrm{lb}(\mathsf{score}^\#(\langle T, n\rangle, \varphi)) \leqslant \mathrm{lub}_{\Phi_\forall}\}$$

The first case captures when no single predicate is non-trivial for all sets: we then return all predicates that succeed on at least one training set in $\langle T, n\rangle$, since we cannot be sure one is strictly better than another. To be sound, we also assume the cause of $\Phi_\forall$ being empty is a particular concrete training set for which every predicate forms a trivial split, hence we include $\diamond$ as a possibility. The second case corresponds to returning the predicates with minimal scores.

Lemma 2.15. *Let* $T' \in \gamma(\langle T, n\rangle)$. *Then,*

$$\mathsf{bestSplit}(T') \in \gamma(\mathsf{bestSplit}^\#(\langle T, n\rangle))$$

Abstracting Conditionals

We abstractly interpret conditionals in DTrace, as is standard, by taking the join of all abstract states from the feasible *then* and *else* paths. In DTrace, there are two branching statements of interest for our purposes, one with the condition $\mathsf{ent}(T) = 0$ and one with $\varphi = \diamond$.

Let us consider the condition $\varphi = \diamond$. Given an abstract state $(\langle T, n\rangle, \Psi)$, we simply set $\Psi = \{\diamond\}$ and propagate the state to the *then* branch (unless, of course, $\diamond \notin \Psi$, in which case we omit this branch). For $\varphi \neq \diamond$, we remove $\diamond$ from Ψ and propagate the resulting state through the *else* branch.

Next, consider the conditional $\mathsf{ent}(T) = 0$. For the *then* branch, we need to *restrict* an abstract state $(\langle T, n\rangle, \Psi)$ to training sets with 0 entropy: intuitively, this occurs when all elements have the same classification. We ask: *are there any concretizations composed of elements of the same class?*, and we proceed through the *then* branch with the following training set

abstraction:

$$\bigsqcup_{i \in [1,k]} \mathit{pure}(\langle T, n \rangle, i)$$

where

$$\begin{aligned} \mathit{pure}(\langle T, n \rangle, i) \coloneqq \text{Let } T' = \{(x, y) \in T \mid y = i\} \text{ in} \\ \text{if } |T \setminus T'| \leqslant n \text{ then } \langle T', n - |T \setminus T'| \rangle \\ \text{else } \bot \end{aligned}$$

The idea is as follows: the set T' defines a subset of T containing only elements of class i. But if we have to remove more than n elements from T to arrive at T', then the conditional is not realizable by a concrete training set of class i, and so we return the empty abstract state.

In the case of $\mathsf{ent}(T) \neq 0$ (the *else* branch), we soundly (imprecisely) propagate the original state without restriction.

Soundness of Abstract Learner

Finally, $\mathsf{DTrace}^\#$ soundly overapproximates the results of DTrace and can therefore be used to prove robustness to n-poisoning.

Theorem 2.16. *Let* $T' \in \gamma(\langle T, n \rangle)$*, let* $(T'_f, \cdot, \cdot)$ *be the final state of* $\mathit{DTrace}(T', x)$*, and let* $(\langle T''_f, n_f \rangle, \cdot)$ *be the final abstract state of* $\mathit{DTrace}^\#(\langle T, n \rangle, x)$*. Then* $T'_f \in \gamma(\langle T''_f, n_f \rangle)$*.*

It follows from the soundness of $\mathsf{DTrace}^\#$ that we can use it to prove n-poisoning robustness. Let $I = ([l_1, u_2], \ldots, [l_k, u_k])$ be a set of intervals. We say that interval $[l_i, u_i]$ dominates I if and only if $l_i > u_j$ for every $j \neq i$.

Corollary 2.17. *Let* $\langle T', n' \rangle$ *be the final abstract state of* $\mathit{DTrace}^\#(\langle T, n \rangle, x)$*. If* $I = \mathsf{cprob}^\#(\langle T', n' \rangle))$ *and there exists an interval in* I *that dominates* I *(i.e., same class is selected for every* $T \in \gamma\langle T, n \rangle$*), then* x *is robust to* n*-poisoning of* T*.*

2.4 Extensions

In this section, we present two extensions that make our abstract interpretation framework more practical. First, we show how our abstract domain can be modified to accommodate real-valued features. Second, we present a disjunctive abstract domain that is more precise than the one we discussed, but more computationally inefficient.

Real-Valued Features

Thus far, we have assumed that DTrace and DTrace$^\#$ operate on a finite set of predicates Φ. In real-world decision-tree implementations, this is not quite accurate: for real-valued features, there are infinitely many possible predicates of the form $\lambda x_i.\, x_i \leqslant \tau$ (where $\tau \in \mathbb{R}$), and the learner chooses a finite set of possible τ values dynamically, based on the training set T. We will use the subscript $\mathbb{R}$ to denote the real-valued versions of existing operations.

From DTrace to DTrace$_\mathbb{R}$. The new learner DTrace$_\mathbb{R}$ is almost identical to DTrace. However, each invocation of bestSplit$_\mathbb{R}$ first computes a finite set of predicates $\Phi_\mathbb{R}$. Consider all of the values appearing in T for the ith feature in $\mathcal{X}$, sorted in ascending order. For each pair of adjacent values (a, b) (i.e., such that there exists no c in T such that $a < c < b$), we include in $\Phi_\mathbb{R}$ the predicate $\varphi = \lambda x_i.\, x_i \leqslant \frac{a+b}{2}$.

Example 2.18. *In our running example from Figure 2.2, we have training set elements in T_{bw} whose features take the numeric values $\{0, 1, 2, 3, 4, 7, \ldots, 14\}$.* bestSplit$_\mathbb{R}(T_{bw})$ *would pick a predicate from the set $\Phi_\mathbb{R} = \{\lambda x.\, x \leqslant \tau \mid \tau \in \{\frac{1}{2}, \frac{3}{2}, \frac{5}{2}, \frac{7}{2}, \frac{11}{2}, \frac{15}{2}, \ldots, \frac{27}{2}\}\}$.*

From DTrace$^\#$ to DTrace$^\#_\mathbb{R}$. To apply the abstract learner in the real-valued setting, we can follow the idea above and construct a finite set $\Phi_\mathbb{R}$. Because our poisoning model assumes dropping up to n elements of the training

set, this results in roughly $(n+1) \cdot |T|$ predicates in the worst case—i.e., we need to account for every pair (a, b) of adjacent feature values or that are adjacent after removing up to n elements between them.

Example 2.19. *Continuing Example 2.18. Say we want to compute $\Phi_{\mathbb{R}}$ for $\langle T_{bw}, 1\rangle$. Then, for every pair of values that are 1 apart we will need to add a predicate to accommodate the possibility that we drop the value between them. E.g., in $T_{bw} = \{\ldots, 3, 4, 7, \ldots\}$, we will additionally need the predicate $\lambda x.\, x \leqslant (3+7)/2$, for the case where we drop the element with value 4 from the dataset.*

To avoid a potential explosion in the size of the predicate set and maintain efficiency, we compactly represent sets of similar predicates symbolically. We describe this detail in Appendix A.1.

Disjunctive Abstraction

The state abstraction used by $\mathsf{DTrace}^{\#}$ can be *imprecise*, mainly due to the join operations that take place, e.g., during $\mathsf{filter}^{\#}$. The primary concern is that we are forced to perform a very imprecise join between possibly quite dissimilar training set fragments. Consider the following example:

Example 2.20. *Let us return to T_{bw} from Figure 2.2, but imagine we have continued the computation after filtering using $x \leqslant 10$ and have selected some best predicates. Specifically, consider a case in which we have $x = 4$ and*

- $\langle T, 1\rangle$, *where* $T = \{0, 1, 2, 3, 4, 7, 8, 9, 10\}$
- $\Psi = \{x \leqslant 3, x \leqslant 4\}$ *(ignoring whether this is correct)*

Let us evaluate $\mathsf{filter}^{\#}(\langle T, 1\rangle, \Psi, x)$. *Following the definition of* $\mathsf{filter}^{\#}$, *we will compute*

$$\langle T', n'\rangle = \langle T_{\leqslant 4}, 1\rangle \sqcup \langle T_{>3}, 1\rangle$$

where

$$\mathsf{T}_{\leqslant 4} = \{(4, \mathsf{b}), (3, w), (2, w), (1, w), (0, \mathsf{b})\}$$
$$\mathsf{T}_{>3} = \{(4, \mathsf{b}), (7, w), (8, w), (9, w), (10, w)\}$$

thus giving us $\mathsf{T}' = \mathsf{T}$ *(the set we began with) and* $\mathfrak{n}' = 5$ *(much larger than what we began with). This is a large loss in precision.*

To address this imprecision, we will consider a *disjunctive* version of our abstract domain, consisting of unboundedly many disjuncts of this previous domain, which we represent as a set $\{(\langle \mathsf{T}, \mathfrak{n} \rangle_i, \Psi_i)\}_i$. Our join operation becomes very simple: it is the union of the two sets of disjuncts.

Definition 2.21 (Disjunctive Joins)**.** *Given two disjunctive abstractions* $\mathsf{D}_I = \{(\langle \mathsf{T}, \mathfrak{n} \rangle_i, \Psi_i)\}_{i \in I}$ *and* $\mathsf{D}_J = \{(\langle \mathsf{T}, \mathfrak{n} \rangle_j, \Psi_j)\}_{j \in J}$*, we define*

$$\mathsf{D}_I \sqcup \mathsf{D}_J \coloneqq \mathsf{D}_I \cup \mathsf{D}_J$$

Adapting DTrace$^\#$ to operate on this domain is immediate: each of the transformers described in the previous section is applied to each disjunct.

Because our disjunctive domain eschews memory- and time-efficiency for precision, we are able to prove more things, but at a cost (we explore this in our evaluation, Section 2.5). Note that, by construction, the disjunctive abstract domain is at least as precise as our standard abstract domain.

2.5 Implementation and Evaluation

We implemented our algorithms DTrace and DTrace$^\#$ in C++ in a (single-threaded) prototype we call Antidote. Our evaluation[8] aims to answer the following research questions:

[8]We use a machine with a 2.3GHz processor and 160GB of RAM throughout.

RQ1 Can Antidote prove data-poisoning robustness for real-world datasets?

RQ2 How does the performance of Antidote vary with respect to the scale of the problem and the choice of abstract domain?

Benchmarks and Experimental Setup

We experiment on 5 datasets (Table 2.1). We obtained the first three datasets from the UCI Machine Learning Repository (Dua and Graff, 2017). Iris is a small dataset that categorizes three related flower species; Mammographic Masses and Wisconsin Diagnostic Breast Cancer are two datasets of differing complexities related to classifying whether tumors are cancerous. We also evaluate on the widely-studied MNIST dataset of handwritten digits (LeCun et al.), which consists of 70,000 grayscale images (60,000 training, 10,000 test) of the digits zero through nine. We consider a form of MNIST that has been used in the poisoning literature and create another variant for evaluation:

- We make the same simplification as in other work on data poisoning (Biggio et al., 2012; Steinhardt et al., 2017) and restrict ourselves to the classification of ones versus sevens (13,007 training instances and 2,163 test instances), which we denote MNIST-1-7-Real. Steinhardt et al. (2017), for example, recently used this to study poisoning in support vector machines.

- Each MNIST-1-7-Real image's pixels are 8-bit integers (which we treat as real-valued); to create a variant of the problem with reduced scale, we *also* consider MNIST-1-7-Binary, a black-and-white version that uses each pixel's most significant bit (i.e. our predicates are Boolean).

For each dataset, we consider a decision-tree learner with a maximum tree depth (i.e. number of calls to bestSplit) ranging from 1 to 4. Table 2.1

Data Set	Size		$\mathcal{X}$	$\lvert\mathcal{Y}\rvert$	DT Test-Set Accuracy (%)			
	Train	Test			d 1	2	3	4
Iris	120	30	$\mathbb{R}^4$	3	20.0	90.0	90.0	90.0
Mammographic Masses	664	166	$\mathbb{R}^5$	2	80.7	83.1	81.9	80.7
Wisconsin Diagnostic Breast Cancer	456	113	$\mathbb{R}^{30}$	2	91.2	92.0	92.9	94.7
MNIST-1-7-Binary	13,007	100*	$\{0,1\}^{784}$	2	95.7	97.4	97.8	98.3
MNIST-1-7-Real	13,007	100*	$\mathbb{R}^{784}$	2	95.6	97.6	98.3	98.7

* Test set accuracy for MNIST is computed on the full 2,163 instances; robustness experiments are performed on 100 randomly chosen test set elements.

Table 2.1: Detailed metrics for the benchmark datasets considered in our evaluation of Antidote.

shows that test set[9] accuracies of the decision trees learned by DTrace are reasonably high—affirmation that when we prove the robustness of its results, we are proving something worthwhile.

Experimental Setup. For each test element, we explore the amount of poisoning (i.e. how large of a n from our Δ_n model) for which we can prove the robustness property as follows.

1. For each combination of dataset T and tree depth d, we begin with a poisoning amount $n = 1$, i.e. a single element could be missing from the training set.

2. For each test set element x, we attempt to prove that x is robust to poisoning T using any set in $\Delta_n(T)$. Let S_n be the test subset for which we do prove robustness for poisoning amount n. If S_n is non-empty, we double n and again attempt to verify the property for each element in S_n.

[9]The UCI datasets come as a single training set. We selected a random 80%-20% split of the data, saving the 20% as the test set to use in our experiments. The scale of the MNIST dataset is large; for pragmatic reasons, we fix a random subset of 100 of the original 2,163 test set elements for robustness proving, and we run our DTrace$^\#$ experiments only on this subset.

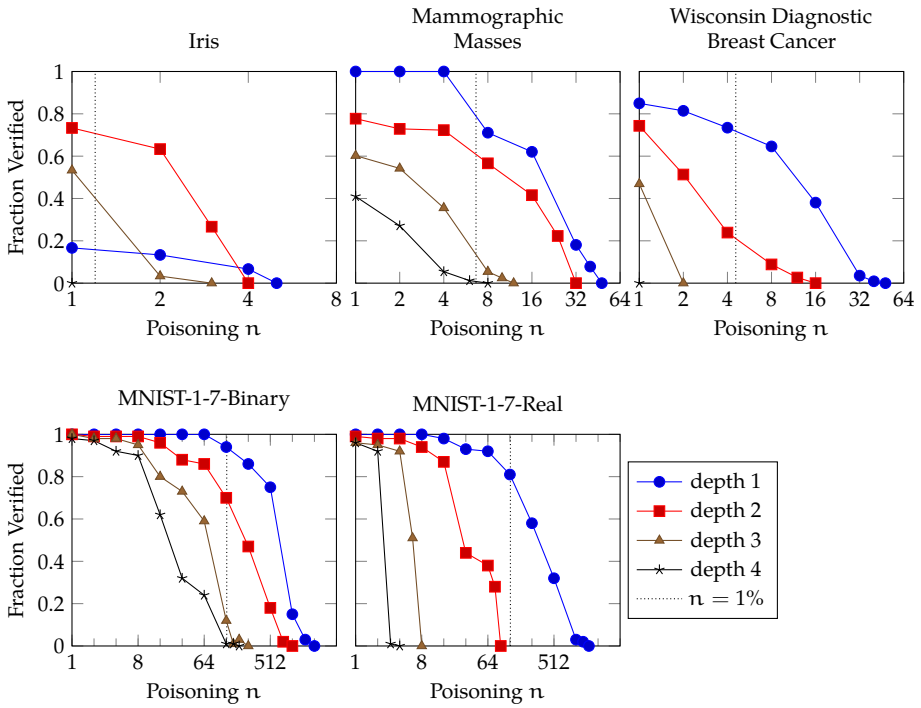


Figure 2.5: Fraction of test instances proven robust versus poisoning parameter n (log scale). The dotted line is a visual aid, indicating n is 1% of the training set size.

3. If at a depth n all instances fail, we binary search between n and $n/2$ to find an $n/2 < n' < n$ at which some instances terminate. This approach allows us to better illustrate the experiment trends in our plots.

Failure occurs due to any of three cases: (*i*) the computed over-approximation does not conclusively prove robustness, (*ii*) the computation runs out of memory, or (*iii*) the computation exceeds a one-hour timeout. We run the entire procedure for the non-disjunctive and disjunctive abstract domains.

Effectiveness of Antidote

We evaluate how effective Antidote is at proving data-poisoning robustness. In this experiment, we consider a run of the algorithm on a single

test element successful if either the non-disjunctive or disjunctive abstract domain succeeds at proving robustness (mimicking a setting in which two instances of DTrace$^\#$, one for each abstract domain, are run in parallel)—we will contrast the results for the different domains in the next subsection. Figure 2.5 shows these results.

To exemplify the power of Antidote, draw your attention to the depth-2 instance of DTrace$^\#$ invoked on MNIST-1-7-Real. For 38 of the 100 test instances, we are able to verify that even if the training set had been poisoned by an attacker who contributed up to 64 poisoned elements ($\approx \frac{1}{2}\%$), the attacker would not have had any power to change the resulting classification. Conventional machine learning wisdom says that, in decision tree learning, small changes to the training set can cause the model to behave quite differently. Our results verify nuance—sometimes, there is some stability.[10] These 38 verified instances average ~800s run time. $\Delta_{64}(\mathsf{T})$ consists of over 10^{174} concrete training sets; This is staggeringly efficient compared to a naïve enumeration baseline, which would be unable to verify robustness at this scale.

To answer **RQ1**, *Antidote can verify robustness for real-world datasets with extremely large perturbed sets and decision-tree learners with high accuracies.*

Performance of Antidote

We evaluate how the performance of Antidote is affected by the complexity of the problem, e.g., the size of the training set and its number of features, the number of poisoned elements, and the depth of the learned decision tree. Due to the large number of parameters involved in our evaluation, this section only provides a number of representative statistics. In particular,

[10] The Iris dataset has an interesting quirk—we're unable to prove much at depth 1 because in the concrete case, one of the leaves is a 50/50 split between two classes, thus changing one element could make the difference for any of the test set instances taking that path. At depth 2, a predicate is allowed to split that leaf further, making decision-tree learning more stable.

although the reader can find plots describing all the metrics evaluated on each dataset in Appendix C.1, most of our analysis will focus on MNIST-1-7-Binary (see Figure 2.6), since it exhibits the most illustrative behavior.

Box vs Disjuncts. In this section we use Disjuncts to refer to the disjunctive abstract domain and Box to refer to the non-disjunctive one. Disjuncts is more precise than Box and, as expected, it can verify more instances. However, Disjuncts is slower and more memory-intensive. Consider the MNIST-1-7-Binary dataset (see Figure 2.6). For depth 3 and $n = 64$ (approximately 0.5% of the dataset), Disjuncts can verify roughly three times as many instances as Box. However, on average, the amount of time and memory required for analyses using Disjuncts to complete each are at least an order of magnitude greater than those using Box. It is worth noting that Box can verify certain instances that Disjunct cannot verify due to timeouts. For example, at depth 4 and $n = 128$, Box is able to verify 1 problem instance,[11] while Disjuncts always times out. To summarize, Disjuncts can in general verify more instances than Box because it is more precise. However, due to Box's performance, there are instances that Box can verify and Disjuncts cannot. An interesting direction for future research would be to consider strategies that capitalize on the precision of tracking many disjuncts while incorporating the efficiency of allowing some to be joined.

Number of Poisoned Elements. It is clear from the plots that the number of poisoned elements greatly affects the performance and efficacy of Antidote. We do not focus on particular numbers, since the trends are clear from the plots (including the ones in Appendix C.1): The memory consumption and running times of Disjuncts grow exponentially with n, but are still practical and Disjuncts is effective up to high depths. The memory consumption and running times of Box grow more slowly: 95% of all experiments we ran using Box finished within 20 seconds, and none

[11] The other instances that succeeded at $n = 64$ terminated after a similar time period, but their final state did not prove robustness.

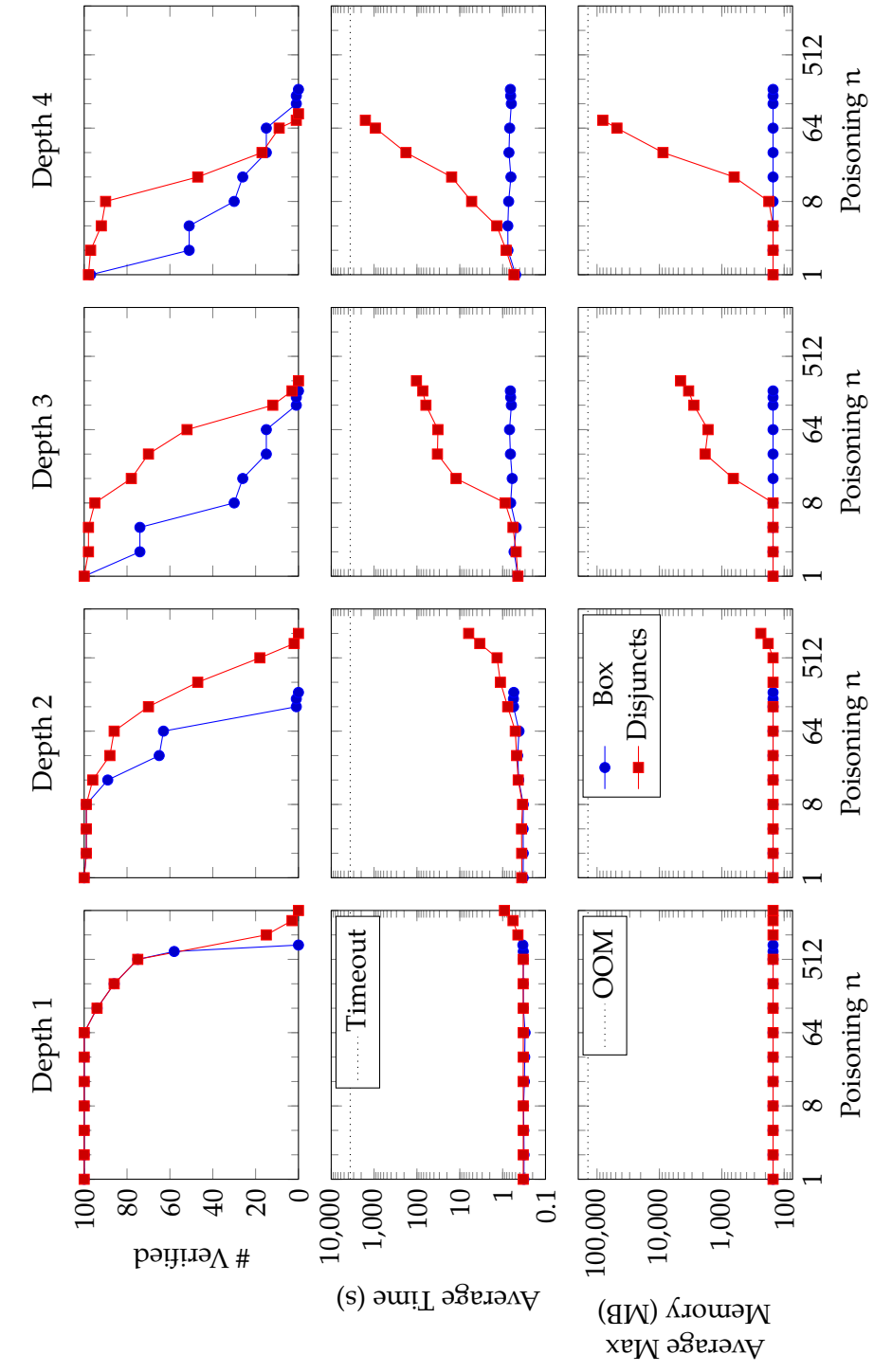


Figure 2.6: Detailed results of Antidote on MNIST-1-7-Binary

timed out (the longest took 232 seconds).[12] However, Box is less effective than Disjuncts as the depths increase; this is expected, as the loss of precision with more operations is more severe for Box.

Size of Dataset and Number of Features. We measure whether the size of the dataset (which in our benchmarks is quite correlated with the number of features) affects the performance. Consider the case of verifying a decision-tree learner of depth 3 using the disjunctive domain and a perturbed set where 0.5% of the points[13] are removed from the dataset (similar trends are observed when varying these parameters). The average running time of Antidote is 0.1s for Iris, 0.2s for Mammographic Masses, 26s for Wisconsin Diagnostic Breast Cancer, and 32s for MNIST-1-7-Binary. For MNIST-1-7-Real, 100% of the benchmarks TO at 0.05% poisoning. As expected, the size of the dataset and the number of features have an effect on the verification time. However, it is hard to exactly quantify this effect, given how differently each dataset behaves; an obvious comparison we can make is the difference between MNIST-1-7-Binary and MNIST-1-7-Real. These two datasets have identical sizes, but the former uses binary features and the latter uses real features. As we can see, handling real features results in a massive slowdown and in proving fewer instances robust. This is not surprising since real features can result in more predicates, which affect both running time and the discrimination power of individual nodes in the decision tree.

Depth of the Tree. Consider the case of verifying a decision-tree learner for MNIST-1-7-Binary using the disjunctive domain, and a perturbed set where up to 64 of the points have been added maliciously to the dataset (similar trends are observed when varying these parameters and for other datasets). The average running time of Antidote as the depth varies, while

[12] This data must be taken with a grain of salt: Box is generally less effective than Disjuncts; due to the incremental nature of our experiments, it did not attempt as many of the "harder" problems as Disjuncts did.

[13] We round to the closest n for which the tool can verify at least one instance

the poisoning n remains constant, increases exponentially: the running time averages on the order of one second at depth 1, but increases to the order of tens of minutes at depth 4. As expected, the depth of the tree is an important factor in the performance of the disjunctive domain, as each abstract operation expands the set of disjuncts.

We summarize the results presented in this section and answer **RQ2**: *in general, the disjunctive domain is more precise but slower than the non-disjunctive domain, and the depth of the learned trees and the number of poisoned elements in the dataset are the greatest factors affecting performance.*

2.6 Related Work

Instability in Decision Trees. Decision-tree learning has a long and storied history. A particular thread of work that is relevant to ours is the analysis of decision-tree *instability* (Dwyer and Holte, 2007; Turney, 1995; Li and Belford, 2002; Pérez et al., 2005). These works show that decision-tree learning algorithms are in general susceptible to small data-poisoning attacks—although they do not phrase it in those terms. For the most part, the works are motivated from the perspective that a decision tree represents a set of "rules," and they are concerned with conditions under which those rules will not change (either by quantifying forms of invariance or providing novel learning algorithms). Our work is different in that it *proves* that no poisoning attack exists on a formalization of very basic decision-tree learning, and we can often precisely allow for the "rules" to change so long as the ultimate classification does not.

Data Poisoning. Data-poisoning robustness has been studied extensively from an attacker perspective (Biggio et al., 2012; Xiao et al., 2012, 2015b; Newell et al., 2014; Mei and Zhu, 2015). This body of work has demonstrated attacks that can degrade classifier accuracy, sometimes dramatically. These works phrase the problem of identifying a poisoned set as

a constraint optimization problem. To make the problem tractable, they typically focus on support vector machines (SVMs) and forms of regression for which existing optimization techniques are readily available. Our approach differs from these works in multiple ways: (*i*) Our work focuses on *decision trees*. The greedy, recursive nature of decision-tree learning is fundamentally different from the optimization problem solved in learning SVMs. (*ii*) While our technique is general, in this chapter we consider a poisoning model in which training elements have been added (Xiao et al., 2015a; Chen et al., 2017). Some works instead focuses on a model in which elements of the training set can be modified (Alfeld et al., 2016). (*iii*) Final and most important, our work *proves* that no poisoning attack exists using abstract interpretation, while existing techniques largely provide search techniques for finding poisoned training sets.

Recently, techniques have been proposed to modify the training processes of machine learning models to make them robust to various data-poisoning attacks (while remaining computationally efficient). These techniques (Laishram and Phoha, 2016; Steinhardt et al., 2017; Diakonikolas et al., 2019a,b) are often based on robust estimation, e.g. outlier removal; see Diakonikolas and Kane (2019) for a survey. In general, these approaches provide limited probabilistic guarantees about certain kinds of attacks; the works are orthogonal to ours, though they raise an interesting question for future work: Can one verify that, on a given training set, these models actually make the training process resistant to data poisoning? Finally, some contemporary work utilizes *randomized smoothing* to certifiably mitigate the power of a data-poisoning adversary, including works that replace randomness with deterministic operations that then yield exact guarantees (Rosenfeld et al., 2020; Weber et al., 2020).

Abstract Interpretation for Robustness. Abstract interpretation (Cousot and Cousot, 1977) is one of the most popular models for static program analysis. Our work is inspired by that of Gehr et al. (2018), where abstract

interpretation is used to prove input-robustness for neural networks. (Recently, Ranzato and Zanella (2020) have done similar work for decision tree ensembles.) Many papers have followed improving on this problem (Anderson et al., 2019; Singh et al., 2019). The main difference between these works and ours is that we tackle the problem of verifying training-time robustness, while existing works focus on test-time robustness. The former problem requires abstracting sets of training sets, while the latter only requires abstracting sets of individual inputs. In particular, Gehr et al. rely on well-known abstract domains—e.g., intervals and zonotopes—to represent sets of real vectors, while our work presents entirely new abstract domains for reasoning about sets of training sets. To our knowledge, our work is the first that even tries to tackle the problem of verifying data-poisoning robustness.

Other works have focused on *provable training* of neural networks to exhibit test-time robustness by construction (Wong and Kolter, 2018; Mirman et al., 2018): this is done by using abstract interpretation to over-approximate the worst-case loss formed by any adversarial perturbation to any element in the training set. One can think of these techniques as performing a form of symbolic training, which is conceptually similar to our core idea. Note, however, two important distinctions: (*i*) These works address the problem of adversarial changes to test inputs, while we address adversarial changes to the training set; (*ii*) These works construct a different, robust model, while we verify a property of an unchanged model (or rather, the learner).

3 FAIRSQUARE: PROBABILISTIC VERIFICATION OF PROGRAM FAIRNESS

We now turn our attention to the problem of verifying the *fairness* of a *decision-making program*, for example, a machine-learned model used to make hiring decisions. We think of decision-making algorithms as probabilistic programs, in the sense that they are invoked on inputs drawn from a probability distribution, e.g., representing the demographics of some population. Fairness properties are then formalized as probabilistic specifications to which the decision-making program must adhere.

Consider a hiring program P that takes as input a vector of arguments $\boldsymbol{v}$ representing a job applicant's record and returns a Boolean value indicating whether the applicant is hired. One of the arguments v_s in the vector $\boldsymbol{v}$ states whether the person is a member of a protected minority or not, and similarly v_q in $\boldsymbol{v}$ states whether the person is qualified or not for the job. Our goal may be to prove a *group fairness* property that is augmented with a notion of qualification—that the algorithm is *just as likely* to hire a qualified minority applicant as it is for other qualified non-minority applicants. Formally, we state this probabilistic condition as follows:

$$\frac{\Pr[\mathrm{P}(\boldsymbol{v}) = \mathit{true} \mid v_s = \mathit{true} \wedge v_q = \mathit{true}]}{\Pr[\mathrm{P}(\boldsymbol{v}) = \mathit{true} \mid v_s = \mathit{false} \wedge v_q = \mathit{true}]} > 1 - \epsilon$$

Here, ϵ is a small constant. In other words, the probability of hiring a person $\boldsymbol{v}$, conditioned on them being a qualified minority applicant, is very close to (or greater than) the probability of hiring a person conditioned on them being a qualified non-minority applicant. We note that, while most recent concerns of fairness have focused on automation of bureaucratic processes, e.g., employment and loan applications, our view of the problem is broad. For instance, fairness properties can be extended to actions and decisions of autonomous agents, like robots and self-driving cars, that

interact with us and affect our environment.

We envision a future in which those who employ algorithmic decision-making in sensitive domains are required to prove fairness of their processes. Towards this vision, our goal in this chapter is to develop an automated technique that can prove fairness properties of programs, like the one shown above, as well as others. With that in mind, we have two key criteria: First, we require a technique that can *construct a proof* of fairness or unfairness of a given program with respect to a specified fairness property. Second, we need to ensure that our technique can handle real-world classes of decision-making programs.

Since our aim is to construct proofs of fairness or unfairness, we focus our development on *exact* probabilistic verification techniques, in contrast with approximate techniques that may provide probabilistic guarantees. We first attempted to reason about fairness using a range of recent probabilistic static analysis techniques that provide exact guarantees (Sankaranarayanan et al., 2013; Gehr et al., 2016), but we observed that these existing techniques are unable to handle the programs and properties we consider. We therefore set out to design a new technique that is suited for our domain of verifying fairness of decision-making programs. The layout of this chapter is as follows:

- In Section 3.1 we provide a high-level overview of our technique for verifying the fairness of a simple (toy) program.
- In Section 3.2 we formalize a simple, but expressive, language for specifying fairness verification problems, and we reduce the problem of automating these verification problems to a set of *weighted-volume-computation problems*.
- In Section 3.3 we present a novel weighted-volume-computation algorithm for formulas over *real closed fields* and prove that it converges to the exact volume (in the limit).

- In Section 3.4 we implement our techniques in a tool, FairSquare, and demonstrate its ability to outperform then-state-of-the-art probabilistic program analyses on a broad spectrum of fairness benchmarks.

Proofs of theorems stated throughout this chapter can be found in Appendix B.2. This chapter is based on the work of Albarghouthi et al. (2017b).

3.1 Overview and Illustration

Our problem setting is as follows: First, we are given a *decision-making program* P_{dec}. Second, we have a *probabilistic precondition* defining a probability distribution over inputs of P_{dec}. We define the probability distribution operationally as a probabilistic program P_{pop}, which we call the *population model*. Intuitively, the population model provides a probabilistic picture of the population from which the inputs of P_{dec} are drawn. Third, we are given a quantitative postcondition φ_{post} that correlates the probabilities of various program outcomes. This postcondition can encode various fairness properties. Intuitively, our goal is to prove the following triple:

$$\{\boldsymbol{v} \sim \mathsf{P}_{\text{pop}}\} \quad \mathrm{r} \leftarrow \mathsf{P}_{\text{dec}}(\boldsymbol{v}) \quad \{\varphi_{\text{post}}\}$$

In this section, we consider a specific fairness property. We will discuss in Section 3.2 how several formulations of fairness can be captured by our framework.

A Simple Verification Problem

Consider the two programs in Figure 3.1(a). The program popModel is a probabilistic program describing a simple model of the population. Here, a member of the population has three attributes, all of which are real-valued: (*i*) ethnicity; (*ii*) colRank, the ranking of the college the person attended (lower is better); and (*iii*) yExp, the years of work experience a

(a)

The *population model* defines a joint probability distribution on attributes of members of a population: (*i*) the rank of the college a person attended (`colRank`), (*ii*) the years of work experience they have (`yExp`), and (*iii*) their ethnicity (`ethnicity`). ***Note that*** `colRank` ***is influenced by a persons's ethnicity***.

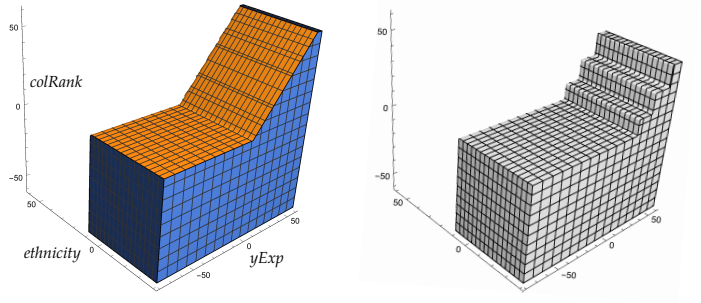
```
1 define popModel()
2   ethnicity ~ gauss(0,10)
3   colRank ~ gauss(25,10)
4   yExp ~ gauss(10,5)
5   if (ethnicity > 10)
6     colRank ← colRank + 5
7   return colRank, yExp
```

The *decision-making* program takes an applicant's record and decides whether to hire them. A person is hired if they are from a *top*-5 college (`colRank <= 5`) or have lots of experience compared to their college rank (`expRank > -5`). ***Note that this program does not access an applicant's ethnicity***.

```
1 define dec(colRank, yExp)
2   expRank ← yExp - colRank
3   if (colRank <= 5)
4     hire ← true
5   elif (expRank > -5)
6     hire ← true
7   else
8     hire ← false
9   return hire
```

(b) Formula φ in $\mathbb{R}^3$ (blue faces are unbounded)

(c) Underapproximation of φ as a union of hyperrectangles



(d) FairSquare ratio computation on `dec` and `popModel` (unfair)

(e) FairSquare ratio computation on modified `dec` (fair)

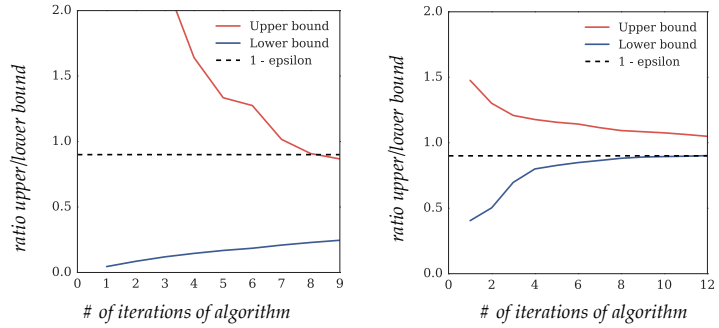


Figure 3.1: Simple illustrative example of FairSquare

person has. We consider a person to be a member of a protected group if ethnicity > 10; we call this the *sensitive condition*. The population model can be viewed as a *generative model* of records of individuals—the more likely a combination is to occur in the population, the more likely it will be generated. For instance, the years of experience an individual has (line 4) follows a Gaussian (normal) distribution with mean 10 and standard deviation 5. Observe that our model specifies that members of a protected minority will probably attend a lower-ranked college, as encoded in lines 5-6.

The program dec is a decision-making program that takes a job applicant's college ranking and years of experience and decides whether they get hired. The program implements a simple decision tree, perhaps one generated by a machine-learning algorithm or written by a person. A person is hired if they attended a *top-5* college (colRank <= 5) or have lots of experience compared to their college's ranking (expRank > -5). Observe that dec *does not access an applicant's ethnicity*.

Our goal is to establish whether the hiring algorithm dec discriminates against members of the protected minority. Concretely, we attempt to prove the following property:

$$\frac{\Pr[\text{hire} \mid \text{min}]}{\Pr[\text{hire} \mid \neg\text{min}]} > 1 - \epsilon$$

where min is shorthand for the sensitive condition ethnicity > 10, and ϵ is a small parameter set to 0.1 for purposes of illustration.

We can rewrite the above statement to eliminate conditional probabilities as follows:

$$\frac{\Pr[\text{hire} \wedge \text{min}] / \Pr[\text{min}]}{\Pr[\text{hire} \wedge \neg\text{min}] / \Pr[\neg\text{min}]} > 1 - \epsilon \tag{3.1}$$

Therefore, to prove the above statement, we need to compute a value for

each of the probability terms: $\Pr[\text{hire} \wedge \text{min}]$, $\Pr[\text{min}]$, and $\Pr[\text{hire} \wedge \neg\text{min}]$. (Note that $\Pr[\neg\text{min}] = 1 - \Pr[\text{min}]$.) Observe that, to prove or disprove inequality 3.1, all we need are sufficiently precise bounds on probabilities—not their exact values.

For the purposes of illustration, we shall focus our description on computing $\Pr[\text{hire} \wedge \neg\text{min}]$.

Probabilistic Verification Conditions. To compute $\Pr[\text{hire} \wedge \neg\text{min}]$, we need to reason about the *composition* of the two programs, dec $\circ$ popModel. That is, we want to compute the probability that (*i*) popModel generates a non-minority applicant, and (*ii*) dec hires that applicant. To do so, we begin by encoding both programs as formulas in the linear-real-arithmetic theory of first-order logic. The process is analogous to that of standard *verification-condition generation* for loop-free program fragments.

First, we encode popModel as follows:

$$\begin{aligned}\varphi_{\text{pop}} \coloneqq\ & (\mathit{ethnicity} > 10 \Rightarrow \mathit{colRank}_1 = \mathit{colRank} + 5) \\ & \wedge\ (\mathit{ethnicity} \leqslant 10 \Rightarrow \mathit{colRank}_1 = \mathit{colRank})\end{aligned}$$

where subscripts are used to encode multiple occurrences of the same variable (i.e., SSA form). Note that assignments drawn from probability distributions do not appear in the encoding—we shall address them later.

Second, we encode dec as follows (after simplification):

$$\begin{aligned}\varphi_{\text{dec}} \coloneqq\ & \mathit{expRank} = \mathit{yExp}^{\mathtt{i}} - \mathit{colRank}^{\mathtt{i}} \\ & \wedge\ (\mathit{hire} \iff (\mathit{colRank}^{\mathtt{i}} \leqslant 5 \vee \mathit{expRank} > -5))\end{aligned}$$

where variables with the superscript $\mathtt{i}$ are the input arguments to dec. Now, to encode the composition dec $\circ$ popModel, we simply conjoin the two formulas—φ_{pop} and φ_{dec}—and add equalities between returns of popModel

and arguments of dec.

$$\varphi_P \coloneqq \varphi_{pop} \wedge \varphi_{dec} \wedge yExp^i = yExp \wedge colRank^i = colRank_1$$

Our goal is to compute the probability that a non-minority applicant gets hired. Formally, we are asking, *what is the probability that the following formula is satisfied?*

$$\varphi \coloneqq \exists V_d.\, \varphi_P \wedge hire \wedge ethnicity \leqslant 10$$

Here, V_d denotes the set of variables that are not probabilistically assigned—that is, all variables *except* $V_p = \{ethnicity, colRank, yExp\}$. Intuitively, by *projecting out* all non-probabilistic variables with existential quantifiers, we get a formula φ whose models are the set of all probabilistic samplings that lead to a non-minority applicant being generated and hired.

Weighted Volume Computation. To compute the probability that φ is satisfied, we begin by noting that φ is, geometrically, a region in $\mathbb{R}^3$, because it has three free, real-valued variables V_p. The region φ is partially illustrated in Figure 3.1(b). Informally, the probability of satisfying φ is the probability of drawing values for the variables in V_p that end up falling in the region described by φ. Therefore, the probability of satisfying φ is its *volume* in $\mathbb{R}^3$, *weighted* by the probability density of each of the three variables. Formally:

$$\Pr[\text{hire} \wedge \neg\text{min}] = \int_{\varphi} p_e p_y p_c \; dV_p$$

where, e.g., p_e is the *probability density function* of the distribution gauss(0,10) (the distribution from which the value of ethnicity is drawn in line 2 of popModel). Specifically, p_e is a function of *ethnicity*, namely, $p_e(ethnicity) = \frac{1}{10\sqrt{2\pi}} e^{-\frac{ethnicity^2}{200}}$.

The primary challenge here is that the region of integration is spec-

ified by an arbitrary SMT formula over an arithmetic theory. So, *how do we compute a numerical value for this integral?* We make two interdependent observations: (*i*) if the formula represents a *hyperrectangular region* in $\mathbb{R}^n$—i.e., a box—then integration is typically simple, due to the constant upper/lower bounds of all dimensions; (*ii*) we can *symbolically decompose* an SMT formula into an (infinite) set of hyperrectangles.

Specifically, given our formula φ, we construct a new formula, $\boxplus_\varphi$, where each model $m \models \boxplus_\varphi$ corresponds to a hyperrectangle that underapproximates φ. Therefore, by systematically finding disjoint hyperrectangles inside of φ and computing their weighted volume, we iteratively improve a *lower bound* on the exact weighted volume of φ. Figure 3.1(c) shows a possible underapproximation of φ composed of four hyperrectangles. The hyperrectangles form a ladder shape that underapproximates the slanted face of φ. We can analogously compute an *upper bound* on the weighted volume of φ: we simply find a lower bound for $\neg\varphi$ and apply the fact that $\Pr[\varphi] = 1 - \Pr[\neg\varphi]$. Section 3.3 formalizes this technique and proves its convergence for decidable arithmetic theories.

Proofs of Group Fairness. We demonstrated how our technique reduces the problem of computing probabilities to weighted volume computation. Figure 3.1(d) illustrates a run of our tool, FairSquare, on this example. FairSquare iteratively improves lower and upper bounds for the probabilities in the ratio, and, therefore, the ratio itself. Observe how the upper bound (red) of the ratio is decreasing and its lower bound (blue) is increasing. This example is not group fair for $\epsilon = 0.1$, because the upper bound goes below 0.9.

Recall that applicants of a protected minority tend to attend lower-ranked colleges, as defined by popModel. Looking at dec, we can point out that the cause for unfairness is the importance of college ranking for hiring. Let us attempt to fix this by modifying line 2 of dec to

```
expRank ← 5*yExp - colRank
```

In other words, we have made the algorithm value job experience far more than college ranking. The run of FairSquare on the modified dec is shown in Figure 3.1(e), where the lower bound on the ratio exceeds 0.9, thus proving our group fairness property.

3.2 A Framework for Verifying Fairness Properties

In this section, we formally define our program model, show how a number of fairness properties can be modeled as probabilistic properties, and present a general framework for specifying and verifying such properties.

Program Model and Semantics

Programs. A program P is a sequence of statements S:

$$
\begin{array}{rl}
\mathsf{S} \coloneqq & \mathsf{V} \leftarrow \mathsf{E} \qquad \text{assignment statement} \\
\mid & \mathsf{V} \sim \mathit{Dist} \qquad \text{probabilistic assignment} \\
\mid & \text{if } \mathsf{B} \text{ then } \mathsf{S} \text{ else } \mathsf{S} \qquad \text{conditional} \\
\mid & \mathsf{S}; \mathsf{S} \qquad \text{sequence of statements}
\end{array}
$$

where V is the set of *real-valued variables* that can appear in P, $e \in \mathsf{E}$ is an arithmetic expression over variables in V, and $b \in \mathsf{B}$ is a Boolean expression over variables in V. A probabilistic assignment is made by sampling from a probability distribution $p \in \mathit{Dist}$. A probability distribution can be, for example, a *Gaussian distribution*, denoted by $\text{gauss}(\mu, \sigma)$, where $\mu, \sigma \in \mathbb{R}$ are the mean and standard deviation of the Gaussian. Without loss of generality, we shall restrict distributions to be *univariate*. We will also assume distributions have only constant parameters, e.g., mean and standard deviation of a Laplacian or Gaussian—that is, we assume inde-

pendence of probabilistic assignments.[1] Given a probabilistic assignment $x \sim p$, we shall treat $p(x)$ as a *probability density function* (PDF) of the distribution from which the value assigned to x is drawn. For instance, if the distribution p is gauss(0,1), then $p(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}$.

We use $\boldsymbol{v}_i$ to denote a vector of *input variables* of P, and $\boldsymbol{v}_o$ to denote a vector of *output variables* of P; these variables appear in V and denote the arguments and returns of P. We say that a program is *closed* if it has no inputs, i.e., $\boldsymbol{v}_i$ is empty. We shall refer to the following subsets of V.

- $V_p \subseteq V$ is the set of *probabilistic variables*: those that get assigned to in probabilistic assignments.
- $V_d = V \setminus V_p$ is the set of *deterministic variables*: those that do not appear in probabilistic assignments.

This simple language can be used to describe typical machine-learning classifiers such as decision trees, support vector machines, Bayesian networks, neural networks, as well as loop-free probabilistic programs (loops with constant bounds can be unrolled). As demonstrated in Section 3.1, the same language is used to define population models programmatically.

Operational Semantics. The operational semantics of our program model is standard, following those introduced by Kozen (1981) and used by other recent papers on the topic (Sampson et al., 2014; Chistikov et al., 2015; Sankaranarayanan et al., 2013). We refer the reader to these texts for an account of the semantics.

Fairness as a Probabilistic Program Property

We now formalize probabilistic pre- and postconditions and use them to define the probabilistic verification problem. We then show how many

[1] Gaussian distributions with non-constant parameters can be handled through properties of Gaussians. E.g., $y \sim \text{gauss}(x, \sigma)$, where $x \in V$ and $\sigma \in \mathbb{R}$, can be transformed into an equivalent sequence of assignments $y \sim \text{gauss}(0, \sigma); y \leftarrow y + x$.

fairness definitions can be expressed in our verification framework.

Probabilistic Verification Problems. A verification problem is a triple $(P_{pop}, P_{dec}, \varphi_{post})$, where

- P_{pop}, called the *population model*, is a closed program over variables V^{pop} and output variables $\boldsymbol{v}_o^{pop}$.
- P_{dec}, called the *decision-making program*, is an open program over variables V^{dec}; its input arguments are $\boldsymbol{v}_i^{dec}$, with $|\boldsymbol{v}_i^{dec}| = |\boldsymbol{v}_o^{pop}|$; and its output variables are $\boldsymbol{v}_o^{dec}$. (We assume that $V^{pop} \cap V^{dec} = \emptyset$.)
- φ_{post} is a *probabilistic postcondition*, which is a Boolean expression over probabilities of program outcomes. Specifically, φ_{post} is defined as follows:

$$\begin{aligned} \varphi_{post} \in \mathrm{BExp} &\coloneqq \mathrm{PExp} > c \mid \mathrm{BExp} \wedge \mathrm{BExp} \mid \neg \mathrm{BExp} \\ \mathrm{PExp} &\coloneqq \Pr[\varphi] \mid c \mid \mathrm{PExp}\, \{+, -, \div, \times\}\, \mathrm{PExp} \end{aligned}$$

  where $c \in \mathbb{R}$ and φ is a linear arithmetic formula over input and output variables of P_{dec}; e.g., φ_{post} might be of the form

$$\Pr[x > 0] > 0.5 \wedge \Pr[y + z > 7] - \Pr[t > 5] > 0$$

The goal of verification is to prove that φ_{post} is true for the program $P_{dec} \circ P_{pop}$, i.e., the composition of the two programs where we first run P_{pop} to generate an input for P_{dec}. Since P_{pop} is closed, $P_{dec} \circ P_{pop}$ is also closed. To avoid division-by-zero problems, we assume that divisors never have value zero. We will use the following definition when stating the meta-properties of our algorithm:

Definition 3.1 (Robust Verification Problem)**.** *Let $\{A_i\}_{i \in \{1,\ldots,n\}}$ be the set of probability events occurring in φ_{post}, and let $\mathbf{a} \in \mathbb{R}^n$ denote the exact values of*

these probabilities as induced by $\mathsf{P}_{\text{dec}} \circ \mathsf{P}_{\text{pop}}$. *We say* $(\mathsf{P}_{\text{pop}}, \mathsf{P}_{\text{dec}}, \varphi_{\text{post}})$ *is "robust" if there exists some* $\varepsilon > 0$ *such that, for every* $\mathbf{b} \in \mathbb{R}^n$ *with each component satisfying* $|b_i - a_i| < \varepsilon$, *substituting each* $\Pr[A_i]$ *with* b_i *in* φ_{post} *always evaluates to the same Boolean value.*

Intuitively, this definition excludes cases where the postcondition "barely" is or is not satisfied, such as $\varphi_{\text{post}} \coloneqq \Pr[A] > \frac{1}{2}$ when $\Pr[A]$ exactly equals $\frac{1}{2}$.

Fairness Properties. We now show how prominent fairness definitions from the literature can be encoded as probabilistic postconditions. At a high-level, all proposed fairness definitions aim to ensure fair decision making, and while some focus on fairness at the granularity of groups, others focus on fairness at the individual level.

We first consider group fairness formulations. Feldman et al. (2015) introduced the following definition, inspired by *Equality of Employment Opportunity Commission's* recommendation in the US (EEOC, 2014):

$$\frac{\Pr[r = \mathit{true} \mid \mathit{min}(\boldsymbol{v}) = \mathit{true}]}{\Pr[r = \mathit{true} \mid \mathit{min}(\boldsymbol{v}) = \mathit{false}]} > 1 - \epsilon$$

Assuming P_{dec} returns a Boolean value r—indicating whether an applicant $\boldsymbol{v}$ is hired—this *group fairness* property states that the selection rate from a protected minority group, $\mathit{min}(\boldsymbol{v}) = \mathit{true}$, is as good as the selection rate from the rest of the population. One can thus view this verification problem as proving a probabilistic property involving *two sets of program traces*: one set where the input $\mathit{min}(\boldsymbol{v})$ is true, and another where it is false. Alternatively, the above definition could be strengthened by conjoining that the reciprocal of the ratio is also at least $1 - \epsilon$, thus ensuring that the selection rate of the two groups is nearly the same (*demographic parity*). Further, we could additionally condition on *qualified* individuals, e.g., if the job has some minimum qualification, we do not want to characterize group fairness for arbitrary applicants, but only within the qualified

subpopulation. Various comparable notions of group fairness have been proposed and used in the literature, e.g., Feldman et al. (2015); Zemel et al. (2013); Datta et al. (2016).

While the above definition is concerned with fairness at the level of subsets of the domain of the decision-making program, *individual fairness* (Dwork et al., 2012) is concerned with similar outcomes for similar elements of the domain. In our hiring example, one potential formulation is as follows:

$$\Pr[r_1 = r_2 \mid \boldsymbol{v}_1 \sim \boldsymbol{v}_2] > 1 - \epsilon$$

In other words, for any two similar individuals (denoted $\boldsymbol{v}_1 \sim \boldsymbol{v}_2$), we want them to receive similar outcomes ($r_1 = r_2$) with a high probability. This is a *hyperproperty*—as it considers two copies of P_{dec}—and can be encoded through *self-composition* (Barthe et al., 2004). This property is close in nature to differential privacy (Dwork, 2006) and robustness (Chaudhuri et al., 2011; Bastani et al., 2016).

Of course, various definitions of fairness have their merits, shortcomings, and application domains, and there is an ongoing discussion on this subject (Friedler et al., 2016; Dwork et al., 2012; Hardt et al., 2016; Feldman et al., 2015; Ajunwa et al., 2016). Our contribution is not to add to this debate, but to cast fairness as a quantitative property of programs, and therefore enable automated reasoning about fairness of decision-making programs.

Probabilistic Inference through Volume Computation

Now that we have defined our program model and the properties we are interested in verifying, we switch attention to constructing *probabilistic verification conditions*.

Following Chistikov et al. (2015), we reduce the problem of computing the probability that the program terminates in a state satisfying φ to

weighted volume computation (wvc) over formulas describing regions in $\mathbb{R}^n$. In what follows, we begin by formalizing the wvc problem.

Weighted Volume of a Formula. We will use $\mathcal{L}$ to denote first-order formulas in *linear real arithmetic* and the strictly richer *real closed fields*—Boolean combinations of polynomial inequalities. Given a formula $\varphi \in \mathcal{L}$, a model m of φ, (denoted $m \models \varphi$) is a point in $\mathbb{R}^n$ (where n is the number of free variables of φ). Thus, we view φ as a region in $\mathbb{R}^n$, i.e., $\varphi \subseteq \mathbb{R}^n$. We use $X_\varphi = \{x_1, \ldots, x_n\}$ to denote the free variables of φ.

We now define the *weighted volume* of a formula. We assume we are given a pair (φ, D), where $\varphi \in \mathcal{L}$ and $D = \{p_1, \ldots, p_n\}$ is a set of probability density functions such that each variable $x_i \in X_\varphi$ is distributed according to $p_i(x_i)$. The weighted volume of φ with respect to D, denoted by $\text{VOL}(\varphi, D)$, is defined as follows:

$$\text{VOL}(\varphi, D) \coloneqq \int_\varphi \prod_{x_i \in X_\varphi} p_i(x_i)\ dX_\varphi$$

Example 3.2. *Consider the formula* $\varphi \coloneqq x_1 + x_2 \geqslant 0$*, and let* $D = \{p_1, p_2\}$*, where* p_1 *and* p_2 *are the* PDF *of the Gaussian distribution with mean* 0 *and standard deviation* 1*. Then,*

$$\text{VOL}(\varphi, D) = \int_{x_1+x_2 \geqslant 0} p_1(x_1)p_2(x_2)\ dx_1 dx_2 = 0.5$$

Intuitively, if we are to randomly draw two values for x_1 *and* x_2 *from the Gaussian distribution, we will land in the region* $x_1 + x_2 \geqslant 0$ *with probability 0.5.*

Probabilistic Verification Conditions. Recall that our goal is to compute the probability of some predicate φ at the end of a program execution, denoted $\Pr[\varphi]$. We now show how to encode this problem as weighted volume computation. First, we encode program executions as a formula φ_P. The process is similar to standard verification condition generation

$$\frac{}{\langle x = [\![e]\!], \emptyset\rangle \rhd x \leftarrow e}\ \textsc{vc-asn} \qquad \frac{}{\langle \mathit{true}, \{p_i\}\rangle \rhd x_i \sim p_i}\ \textsc{vc-pasn}$$

$$\frac{\langle \varphi_1, D_1\rangle \rhd S_1 \qquad \langle \varphi_2, D_2\rangle \rhd S_2}{\langle \varphi_1 \wedge \varphi_2, D_1 \cup D_2\rangle \rhd S_1; S_2}\ \textsc{vc-seq}$$

$$\frac{\langle \varphi_1, D_1\rangle \rhd S_1 \qquad \langle \varphi_2, D_2\rangle \rhd S_2}{\langle \mathit{ite}([\![b]\!], \varphi_1, \varphi_2), D_1 \cup D_2\rangle \rhd \text{if } b \text{ then } S_1 \text{ else } S_2}\ \textsc{vc-cond}$$

$$\text{where } \mathit{ite}(a, b, c) \triangleq (a \Rightarrow b) \wedge (\neg a \Rightarrow c)$$

Figure 3.2: Probabilistic verification condition generation

(as used by verification (Barnett and Leino, 2005) and bounded model checking tools (Clarke et al., 2004)), with the difference that probabilistic assignments populate a set D of probability density functions.

Figure 3.2 inductively defines the construction of a *probabilistic verification condition* for a program P, denoted by a function $\textsc{pvc}(P)$, which returns a pair $\langle \varphi_P, D\rangle$. Without loss of generality, to simplify our exposition, we assume programs are in *static single assignment* (SSA) form (Cytron et al., 1991). Given a Boolean expression b, the denotation $[\![b]\!]$ is the same expression interpreted as an $\mathcal{L}$ formula. The same applies to arithmetic expressions e. For example, $[\![x + y > 0]\!] \triangleq x + y > 0$. Intuitively, the construction generates (*i*) a formula φ_P that encodes program executions, treating probabilistic assignments as non-deterministic, and (*ii*) a set D of the PDFs of distributions in probabilistic assignments (rule VC-PASN).

Now, suppose we are given a closed program P and a Boolean formula φ over its output variables. Then,

$$\Pr[\varphi] = \textsc{vol}(\exists V_d.\ \varphi_P \wedge \varphi, D)$$

That is, we project out all non-probabilistic variables V_d from $\varphi_P \wedge \varphi$

1: **function** VERIFY($P_{pop}, P_{dec}, \varphi_{post}$)
2: $\langle \varphi_{pop}, D_{pop} \rangle \leftarrow \textsc{pvc}(P_{pop})$
3: $\langle \varphi_{dec}, D_{dec} \rangle \leftarrow \textsc{pvc}(P_{dec})$
4: $\langle \varphi_P, D \rangle \leftarrow \langle \varphi_{pop} \wedge \varphi_{dec} \wedge \boldsymbol{v}_i^{dec} = \boldsymbol{v}_o^{pop}, D_{pop} \cup D_{dec} \rangle$
5: $V_d \leftarrow V_d^{pop} \cup V_d^{dec}$
6: $m \leftarrow \emptyset$
7: **for each** expression $\Pr[\varphi]$ in φ_{post} **do**
8: $m \leftarrow m[\Pr[\varphi] \mapsto \textsc{vol}(\exists V_d.\, \varphi_P \wedge \varphi, D)]$
9: **return** $m \models \varphi_{post}$

Figure 3.3: Abstract verification algorithm for FairSquare

and compute the weighted volume with respect to the densities $p_i \in D$. Intuitively, each model m of $\exists V_d.\, \varphi_P \wedge \varphi$ corresponds to a sequence of values drawn in probabilistic assignments in an execution of P. We note that our construction is closely related to that of Chistikov et al. (2015), to which we refer the reader for a measure-theoretic formalization.

Example 3.3. *Consider the following closed program* P*:*

```
x ~ gauss(0,2);
y ~ gauss(-1,1);
z ← x + y
```

where z *is the return variable. Using the encoding in Figure 3.2, we compute the pair* $\langle \varphi_P, D \rangle \rhd P$*, where* $\varphi_P \triangleq z = x + y$ *and* $D = \{p_x, p_y\}$*, where* p_x *and* p_y *are the* PDFs *of the two distributions from which values of* x *and* y *are drawn.*

Suppose that we would like to compute the probability that z *is positive when the program terminates:* $\Pr[z \geqslant 0]$*. Then, we can compute the following weighted volume:* $\textsc{vol}(\exists z.\, \varphi_P \wedge z \geqslant 0, D)$*, which is* ~ 0.327*.*

Verification Algorithm. We now describe an idealized verification algorithm that assumes the existence of an oracle VOL for measuring probability expressions appearing in the postcondition.The algorithm VERIFY, shown

in Figure 3.3, takes a verification problem and returns whether the probabilistic postcondition holds.

VERIFY begins by encoding the composition of the two programs, $P_{dec} \circ P_{pop}$, as the pair $\langle \varphi_P, D \rangle$ and adds the constraint $\boldsymbol{v}_i^{dec} = \boldsymbol{v}_o^{pop}$ to connect the outputs of P_{pop} to the inputs of P_{dec} (recall the example from Section 3.1 for an illustration). For each term of the form $\Pr[\varphi]$ appearing in φ_{post}, the algorithm computes its numerical value and maintains it in a map m. If m satisfies the φ_{post}—i.e., by replacing all terms $\Pr[\varphi]$ with their values in m—then the postcondition holds.

3.3 Symbolic Probabilistic Inference

We now turn our attention to our probabilistic inference algorithm, which reduces the problem to computing the weighted volume of a formula. Recall that we are given (*i*) a formula φ over real arithmetic constraints, encoding the semantics of a program, and (*ii*) a set D defining the PDFs of the distributions of free variables of φ. Our goal is to evaluate the integral $\int_{\varphi} \prod_{x_i \in X_\varphi} p_i(x_i)\, dX_\varphi$. We begin by describing limitations of existing approaches.

Existing Techniques. In general, there is no systematic technique for computing an exact value for such an integral. Moreover, even simpler linear versions of the volume computation problem, not involving probability distributions, are #P-hard (Dyer and Frieze, 1988). Existing techniques suffer from one or more of the following: they (*i*) restrict φ to a conjunction of linear inequalities (Sankaranarayanan et al., 2013; De Loera et al., 2012), (*ii*) restrict integrands to polynomials or uniform distributions (De Loera et al., 2012; Belle et al., 2015b; Chistikov et al., 2015; Belle et al., 2015a), (*iii*) compute approximate solutions with probabilistic guarantees (Chistikov et al., 2015; Vempala, 2005; Belle et al., 2015a), (*iv*) restrict φ to bounded regions of $\mathbb{R}^n$ (Chistikov et al., 2015), or (*v*) have no con-

vergence guarantees, e.g., computer algebra tools that find closed-form solutions, like Mathematica and PSI (Gehr et al., 2016). (See Section 3.5 for details.)

Symbolic Weighted Volume Computation. Our approach is novel in its generality and its algorithmic core. The following are the high-level properties of our algorithm:

1. It is guaranteed to converge to the exact value of the weighted volume in the limit. *This allows us to produce a sound and complete procedure for verifying fairness properties.*

2. It imposes no restrictions on PDFs, only that we can evaluate the *cumulative distribution functions* (CDFs) associated with the PDFs in D.[2] *This provides us with flexibility in defining population models.*

3. It accepts formulas in the decidable yet rich theory of *real closed fields*: Boolean combinations of polynomial inequalities. *This provides a rich language for encoding many decision-making programs, as we demonstrate in Section 3.4.*

At the algorithmic level, our approach makes the following contributions:

1. It exploits the power of SMT solvers and uses them as a black box, allowing it to directly benefit from future advances in solver technology.

2. It employs the idea of dividing the space into rectangular regions that are easy to integrate over. While this idea has been employed in various guises in verification (Bournez et al., 1999; Sankaranarayanan et al.,

[2]The *cumulative distribution function* of a real-valued random variable X is the function $f : \mathbb{R} \to \mathbb{R}$, such that $f(x) = \Pr[X \leqslant x]$. In practice, *evaluating* a CDF means either computing the value of the function exactly or approximating its value to specified high degree of precision; see Section 3.4.

2013; Asarin et al., 2000; Li et al., 2014), we utilize it in a new symbolic way to enable volume computation over SMT formulas.

3. It introduces a novel technique for approximately encoding PDFs as formulas and using them to guide the SMT solver towards making large leaps to the exact solution. This technique is crucial when dealing with decision-making programs comprised of halfspaces, as we show experimentally in Section 3.4.

Weighted Volume Computation Algorithm

To compute the integral over the region φ, we exploit the observation that if φ is a *hyperrectangular region*, i.e., an n-dimensional rectangle in $\mathbb{R}^n$, then we can evaluate the integral, because each dimension has constant lower and upper bounds. For instance, consider the following formula representing a rectangle in $\mathbb{R}^2$:

$$\varphi \coloneqq 0 \leqslant x_1 \leqslant 100 \land 4 \leqslant x_2 \leqslant 10$$

The following holds:

$$\begin{aligned}\int_{\varphi} p_1(x_1)p_2(x_2)\ dx_1dx_2 &= \left(\int_0^{100} p_1(x_1)\ dx_1\right)\left(\int_4^{10} p_2(x_2)\ dx_2\right)\\ &= (F_1(10) - F_1(4))(F_2(100) - F_2(0))\end{aligned}$$

where $F_i = \int_{-\infty}^{x} p_i(t)\ dt$ is the CDF of $p_i(x_i)$. That is, we independently compute the integral along each dimension of the rectangle and take the product. This holds since all variables are independently sampled.

Our algorithm is primarily composed of two steps: First, the *hyperrectangular decomposition* phase represents the formula φ as a set of hyperrectangles. Note that this set is likely to be infinite. Thus, we present a technique for defining all hyperrectangles that lie in φ symbolically as a

formula $\boxdot_\varphi$, where each model of $\boxdot_\varphi$ corresponds to a hyperrectangle that lies inside the region φ. Second, after characterizing the set $\boxdot_\varphi$ of all hyperrectangles in φ, we can iteratively *sample hyperrectangles* in φ, which can be done using an off-the-shelf SMT solver to find models of $\boxdot_\varphi$. For each hyperrectangle we sample, we compute its weighted volume and add it to our current solution. Therefore, the current solution maintained by the algorithm is the weighted volume of an underapproximation of φ—that is, a lower bound on the exact weighted volume of φ.

Hyperrectangular Decomposition. We begin by defining hyperrectangles as special formulas.

Definition 3.4 (Hyperrectangles and their weighted volume)**.** *A formula $H \in \mathcal{L}$ is a hyperrectangle if it can be written in the form $\bigwedge_{x \in X_H} c_x \leqslant x \leqslant c'_x$, where $c_x, c'_x \in \mathbb{R}$ are the lower and upper bounds of dimension x. We use $H_l(x)$ and $H_u(x)$ to denote the lower and upper bounds of x in H. The weighted volume of H, given a set of distributions D, is as follows:*

$$\text{VOL}(H, D) = \prod_{x_i \in X_H} \int_{H_l(x_i)}^{H_u(x_i)} p_i(x_i)\, dx_i$$

Ideally, we would take a formula φ and rewrite it as a disjunction of hyperrectangles $\bigvee H$, but this disjunction is most likely infinite. To see why, consider the simple formula representing a triangular polytope in Figure 3.4(a). Here, there is no finite number of rectangles whose union is the full region in $\mathbb{R}^2$ enclosed by the triangle.

While the number of hyperrectangles enclosed in φ is infinite, we can characterize them symbolically using universal quantifiers, as shown by Li et al. (2014). Specifically, we define the hyperrectangular decomposition of φ as follows:

Definition 3.5 (Hyperrectangular decomposition)**.** *Given φ, its* hyperrect-

angular decomposition $\square_\varphi$ *is:*

$$\square_\varphi \equiv \left(\bigwedge_{x \in X_\varphi} l_x < u_x \right) \wedge \forall X_\varphi . \left(\left(\bigwedge_{x \in X_\varphi} l_x \leqslant x \leqslant u_x \right) \Rightarrow \varphi \right)$$

where l_x, u_x *are fresh free variables introduced for each* $x \in X_\varphi$*, and* $\forall X_\varphi$ *is short for* $\forall x_1, \ldots, x_n$*, for* $x_i \in X_\varphi$*.*

Given a model $m \models \square_\varphi$*, we say that* H^m *is the* hyperrectangle induced by m*, as defined below:*

$$H^m \equiv \bigwedge_{x \in X_\varphi} m(l_x) \leqslant x \leqslant m(u_x)$$

Intuitively, $\square_\varphi$ characterizes every possible hyperrectangle that is subsumed by φ. The idea is that the hyperrectangle H^m induced by each model m of $\square_\varphi$ is subsumed by φ, that is, $H^m \Rightarrow \varphi$. The following example illustrates this process.

Example 3.6. *Consider the formula* $\varphi \equiv x \geqslant y \wedge y \geqslant 0$*, illustrated in Figure 3.4(c) as a gray, unbounded polygon. The formula* $\square_\varphi$*, after eliminating the universal quantifier, is:*

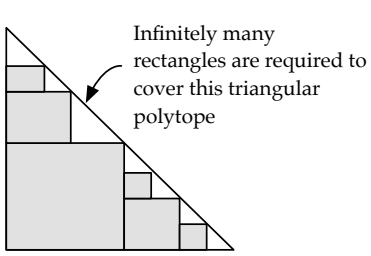
$$l_x < u_x \wedge l_y < u_y \wedge l_y \geqslant 0 \wedge l_x \geqslant u_y$$

Figure 3.4(c) shows two models $m_1, m_2 \models \square_\varphi$ *and their graphical representation as rectangles* H^{m_1}, H^{m_2} *in* $\mathbb{R}^2$*. Observe that both rectangles are subsumed by* φ*.*

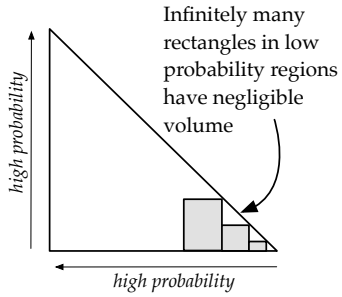
The following theorem states the soundness and completeness of hyperrectangular decomposition: models of $\square_\varphi$ characterize all hyperrectangles in φ and no others.

Theorem 3.7 (Correctness of $\square$)**.** *Let* $\varphi \in \mathcal{L}$*.*

- ***Soundness****: Let* $m \models \square_\varphi$*. Then,* $H^m \Rightarrow \varphi$ *is valid.*



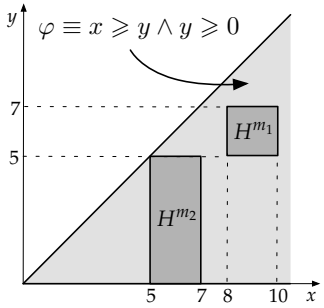
(a)



(b)

Two examples of models inducing rectangles

$m_1 \models \square\!\!\square_\varphi$	$m_2 \models \square\!\!\square_\varphi$
$l_x = 8$	$l_x = 5$
$u_x = 10$	$u_x = 7$
$l_y = 5$	$l_y = 0$
$u_y = 7$	$u_y = 5$



(c)

Figure 3.4: (a) $\mathbb{R}^2$ view of hyperrectangular decomposition. (b) Hyperrectangle sampling, where density is concentrated in the top-left corner. (c) Illustration of models of $\square\!\!\square_\varphi$.

- ***Completeness****: Let* H *be a hyperrectangle such that* $H \Rightarrow \varphi$*. Then, the following is satisfiable:*

$$\square\!\!\square_\varphi \wedge \bigwedge_{x \in X_\varphi} (l_x = H_l(x) \wedge u_x = H_u(x))$$

Hyperrectangle Sampling. Our symbolic weighted volume computation algorithm, SYMVOL, is shown in Figure 3.5 as two transition rules. Given a pair (φ, D), the algorithm maintains a state consisting of two variables: (*i*) *vol*, the current lower bound of the weighted volume, and (*ii*) Ψ, a

$$\frac{}{\mathit{vol} \leftarrow 0 \quad \Psi \leftarrow \boxplus_{\varphi}} \textsc{hdecomp}$$

$$\frac{\mathfrak{m} \models \Psi}{\mathit{vol} \leftarrow \mathit{vol} + \textsc{vol}(H^{\mathfrak{m}}, D) \quad \Psi \leftarrow \Psi \wedge \mathit{block}(H^{\mathfrak{m}})} \textsc{hsample}$$

$$\text{where } \mathit{block}(H^{\mathfrak{m}}) \coloneqq \bigvee_{x \in X_{\varphi}} u_x < H^{\mathfrak{m}}_{l}(x) \vee l_x > H^{\mathfrak{m}}_{u}(x)$$

Figure 3.5: SYMVOL: weighted volume computation algorithm

constraint that encodes the *remaining* rectangles in the hyperrectangular decomposition of φ.

The algorithm is presented as guarded rules. Initially, using the rule HDECOMP, *vol* is set to 0 and Ψ is set to $\boxplus_{\varphi}$. The algorithm then proceeds by iteratively applying the rule HSAMPLE. Informally, the rule HSAMPLE is used to find arbitrary hyperrectangles in φ and compute their weighted volume. Specifically, HSAMPLE finds a model $\mathfrak{m}$ of Ψ, computes the weighted volume of the hyperrectangle $H^{\mathfrak{m}}$ induced by $\mathfrak{m}$, and adds the result to *vol*.

To maintain soundness, HSAMPLE ensures that it never samples two overlapping hyperrectangles, as otherwise we would overapproximate the volume. To do so, every time a hyperrectangle $H^{\mathfrak{m}}$ is sampled, we conjoin an additional constraint to Ψ—denoted $\mathit{block}(H^{\mathfrak{m}})$ and defined in Figure 3.5—that ensures that for all models $\mathfrak{m}' \models \Psi$, $H^{\mathfrak{m}'}$ does not overlap with $H^{\mathfrak{m}}$, i.e., $H^{\mathfrak{m}'} \wedge H^{\mathfrak{m}}$ is unsatisfiable. Informally, the $\mathit{block}(H^{\mathfrak{m}})$ constraint specifies that any newly sampled hyperrectangle should be to the *left* or *right* of $H^{\mathfrak{m}}$ for at least one of the dimensions.

The following theorem states the correctness of *block*: it removes all hyperrectangles that overlap with $H^{\mathfrak{m}}$ (soundness), and it does not over-constrain Ψ by removing hyperrectangles that do not overlap with $H^{\mathfrak{m}}$ (completeness).

Theorem 3.8 (Correctness of *block*)**.** *Given* φ*, let* $\Psi \Rightarrow \boxplus_{\varphi}$*, and let* $m_1, m_2 \models \Psi$*.*

- ***Soundness****: If* $H^{m_1} \wedge H^{m_2}$ *is satisfiable, then* $m_2 \not\models \Psi \wedge \mathit{block}(H^{m_1})$*.*
- ***Completeness****: If* $H^{m_1} \wedge H^{m_2}$ *is unsatisfiable, then* $m_2 \models \Psi \wedge \mathit{block}(H^{m_1})$*.*

Lower and Upper Bounds. The following theorem states the soundness of SYMVOL: it maintains a lower bound on the exact weighted volume.

Theorem 3.9 (Soundness of SYMVOL)**.** *It is an invariant of* SYMVOL(φ, D) *that* $\mathit{vol} \leqslant \text{VOL}(\varphi, D)$*.*

Proof. At any point in the execution, $\mathit{vol} = \sum_{i=1}^{l} \int_{H_i} \prod p_i(x_i)\, dX_{\varphi}$, where l is the number of applications of HSAMPLE and H_i is the hyperrectangle sampled at step i. By definition, $\bigvee H_i \Rightarrow \varphi$. Since PDFs are positive functions, $\mathit{vol} \leqslant \text{VOL}(\varphi, D)$.

It follows from the above theorem that we can use SYMVOL to compute an upper bound on the exact volume. Specifically, because we are integrating over PDFs, we know that $\text{VOL}(\varphi, D) + \text{VOL}(\neg\varphi, D) = 1$. Therefore, by using SYMVOL to compute the weighted volume of $\neg\varphi$, we get an upper bound on the exact volume of φ.

Corollary 3.10 (Upper bounds)**.** *It is an invariant of* SYMVOL$(\neg\varphi, D)$ *that* $1 - \mathit{vol} \geqslant \text{VOL}(\varphi, D)$

Proof. By definition of PDFs and integration,

$$\int_{\mathbb{R}^n} \prod p_i(x_i)\, dX_{\varphi} = \int_{\varphi} \prod p_i(x_i)\, dX_{\varphi} + \int_{\neg\varphi} \prod p_i(x_i)\, dX_{\varphi}$$

for any $\varphi \subseteq \mathbb{R}^n$. From Theorem 3.9, it follows that at any point in the execution of SYMVOL$(\neg\varphi, D)$, we have $1 - \mathit{vol} \geqslant \text{VOL}(\varphi, D)$.

Density-Directed Sampling

While the SYMVOL algorithm is sound, it provides no progress guarantees. Consider, for example, that the algorithm might diverge by sampling hyperrectangles in φ that appear in very low probability density regions, as illustrated in Figure 3.4(b) on a triangular polytope in $\mathbb{R}^2$.

Ideally, the rule HSAMPLE would always find a model $\mathfrak{m}$ yielding the hyperrectangle $H^{\mathfrak{m}}$ with the *largest weighted volume*. Finding such a model amounts to solving the optimization problem:

$$\arg\max_{\mathfrak{m} \models \Psi} \prod_{x_i \in X_\varphi} \int_{H_l^{\mathfrak{m}}(x_i)}^{H_u^{\mathfrak{m}}(x_i)} p_i(x_i)\, dx_i$$

From a practical perspective, there are no known tools or techniques for finding models of first-order formulas that maximize such complex objective functions—with integrals over probability density functions.

However, we make the key observation that if $p(x)$ is a *step function*—i.e., piecewise constant—then we can symbolically encode $\int p(x)\, dx$ in linear arithmetic. As such, we propose to (*i*) *approximate* each density function $p(x)$ with a step function $\mathit{step}(x)$, (*ii*) *encode* the integrals $\int \mathit{step}(x)\, dx$ as linear arithmetic formulas, and (*iii*) *direct* sampling towards hyperrectangles that maximize these integrals, thus finding hyperrectangles of large volume.

Approximate Density Functions. We begin by defining *approximate density functions* (ADFs).

Definition 3.11 (Approximate density functions)**.** *An approximate density function* $\mathit{step}(x)$ *is of the form:*

$$\mathit{step}(x) = \begin{cases} c_i, & x \in [a_i, b_i) \quad \textit{for} \quad 1 \leqslant i \leqslant n \\ 0, & \textit{otherwise} \end{cases}$$

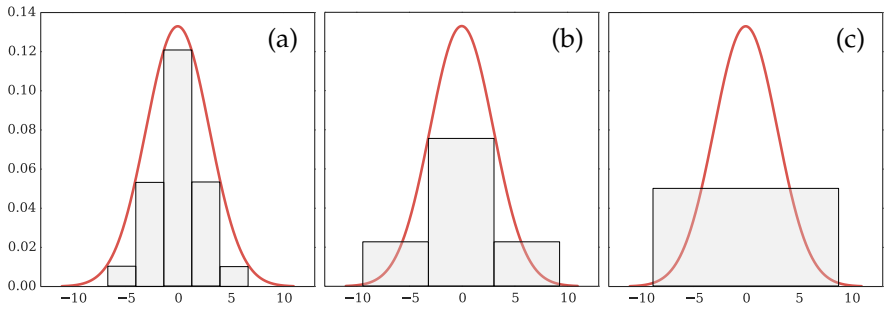


Figure 3.6: Three ADFs (gray) of a Gaussian PDF (red) with mean 0 and standard deviation 3: (a) *fine-grained*; (b) *coarse*; (c) *uniform*.

where $c_i, a_i, b_i \in \mathbb{R}$, $c_i > 0$, *and all* $[a_i, b_i)$ *are disjoint.*

We now show how to encode a formula $\mathit{step}^{\Phi}(x)$ over the free variables δ_x, l_x, u_x, where for any model $m \models \mathit{step}^{\Phi}(x)$, the value $m(\delta_x)$ is the area under $\mathit{step}(x)$ between $m(l_x)$ and $m(u_x)$, i.e.: $m(\delta_x) = \int_{m(l_x)}^{m(u_x)} \mathit{step}(x)\ dx$. Intuitively, the value of this integral is the sum of the areas of each *bar* in $\mathit{step}(x)$, restricted to $[m(l_x), m(u_x)]$.

Definition 3.12 (Encoding area under an ADF)**.** *Given an* ADF $\mathit{step}(x)$*, we define* $\mathit{step}^{\Phi}(x)$ *as follows:*

$$\mathit{step}^{\Phi}(x) \equiv \delta_x = \sum_{i=1}^{n} c_i \cdot \left|[a_i, b_i) \cap [l_x, u_x]\right|$$

The finite sum in $\mathit{step}^{\Phi}(x)$ computes the size of the intersection of $[l_x, u_x]$ with each interval $[a_i, b_i)$ in $\mathit{step}(x)$, and multiplies the intersection with c_i, the value of the *step* in that interval. Note that the constraint $\mathit{step}^{\Phi}(x)$ is directly expressible in linear arithmetic, since

$$\left|[a_i, b_i) \cap [l_x, u_x]\right| = \max(\min(b_i, u_x) - \max(a_i, l_x), 0)$$

The following theorem states the correctness of the ADF encoding:

Theorem 3.13 (Correctness of step^{Φ})**.** *Fix an* ADF $\mathit{step}(x)$.

- ***Soundness**: For any model $m \models step^{\Phi}$, the following is true: $m(\delta_x) = \int_{m(l_x)}^{m(u_x)} step(x)\, dx$.*
- ***Completeness**: For any constants $a, b, c \in \mathbb{R}$ such that $c = \int_a^b step(x)\, dx$, the following formula is satisfiable: $\delta_x = c \wedge l_x = a \wedge u_x = b \wedge step^{\Phi}(x)$.*

ADF-Directed Volume Computation. We now present the algorithm ADF-SYMVOL (Figure 3.7), an extension of our volume computation algorithm SYMVOL that uses ADFs to steer the sampling process. The ADFs are only used for guiding the rule HSAMPLE towards dense hyperrectangles, and thus do not affect soundness of the volume computation. For example, Figure 3.6 shows three approximations of a Gaussian; all three are valid approximations. In Section 3.4, we discuss the impact of different ADFs on performance.

Formally, we create a set of ADFs $\mathcal{A} = \{step_1, \ldots, step_n\}$, where, for each variable $x_i \in X_\varphi$, we associate the ADF $step_i(x_i)$. The rule HSAMPLE now encodes $step_i^{\Phi}(x_i)$ and attempts to find a hyperrectangle such that for each dimension x, δ_x is greater than some lower bound *lb*, which is initialized to 1. Of course, we need to reduce the value *lb* as we run out of hyperrectangles of a given volume. Therefore, the rule DECAY is used to shrink *lb* using a fixed *decay rate* $\lambda \in (0, 1)$ and can be applied when HSAMPLE fails to find a sufficiently large hyperrectangle.

Example 3.14. *Suppose we want to find the weighted volume of a single-variable formula*

$$\varphi \equiv 0 \leqslant x \leqslant 1$$

where x is uniformly distributed over the interval $[0, 1]$. Its hyperrectangular decomposition is

$$\square_\varphi \equiv l_x < u_x \wedge \forall x.\, (l_x \leqslant x \leqslant u_x \Rightarrow 0 \leqslant x \leqslant 1)$$

or equivalently, if we eliminate the quantifier,

$$\square_\varphi \equiv l_x < u_x \wedge 0 \leqslant l_x \leqslant 1 \wedge 0 \leqslant u_x \leqslant 1$$

It's clear that an arbitrary model of $\square_\varphi$ *can be any single interval* $I \subseteq [0,1]$*, and since* x *is uniformly distributed over* $[0,1]$*, the weighted volume of* I *is exactly its size. Thus, we would like the models to be* large *intervals; to do so, we employ a constraint based on the* ADF *of* x*.*

Since x *is uniformly distributed, we can use its actual distribution as its* ADF*. We then have that*

$$step^\Phi(x) \equiv \delta_x = 1 \cdot \left|[0,1) \cap [l_x, u_x]\right| \equiv \delta_x = max(min(1, u_x) - max(0, l_x), 0)$$

Here, δ_x *represents the weighted volume contribution of the variable* x *(which happens to be the only variable in* φ*), and so if we obtain a model not of* $\square_\varphi$*, but instead of the formula* $\square_\varphi \wedge \exists \delta_x.\, (step^\Phi(x) \wedge \delta_x \geqslant lb)$*, explicitly written as*

$$\begin{aligned} &l_x < u_x \wedge 0 \leqslant l_x \leqslant 1 \wedge 0 \leqslant u_x \leqslant 1 \\ &\wedge \exists \delta_x.\, \big(\delta_x = max(min(1, u_x) - max(0, l_x), 0) \wedge \delta_x \geqslant lb\big) \end{aligned}$$

then the weighted volume of the worst *model approximately increases as a function of* lb*. In fact, when* $lb = 1$*, the only model is the whole unit interval (where* $l_x = 0$ *and* $u_x = 1$*), which contains all of the probability mass of* x*.*

Note that, ideally, we would look for a model m such that $\prod_{x \in X_\varphi} \delta_x$ is maximized, thus, finding the hyperrectangle with the largest weighted volume with respect to the ADFs. However, this constraint is non-linear. To lower the complexity of the problem to that of linear arithmetic, we set a decaying lower bound and attempt to find a model where each δ_x is greater than the lower bound.

$$\frac{}{vol \leftarrow 0 \quad \Psi \leftarrow \boxplus_{\varphi} \quad lb \leftarrow 1}\ \textsc{hdecomp} \qquad \frac{}{lb \leftarrow \lambda * lb}\ \textsc{decay}$$

$$\frac{m \models \Psi \wedge \bigwedge_{x_i \in X_\varphi} \exists \delta_{x_i}.\, step_i^{\phi}(x_i) \wedge \delta_{x_i} \geqslant lb}{vol \leftarrow vol + \textsc{vol}(H^m, D) \qquad \Psi \leftarrow \Psi \wedge block(H^m)}\ \textsc{hsample}$$

Figure 3.7: ADF-SYMVOL: ADF-directed volume computation

Convergence of Algorithm

We now discuss the convergence properties of ADF-SYMVOL. Suppose we are given a formula φ, a set of PDFs D, and a set of ADFs $\mathcal{A}$. Let $R \subset \mathbb{R}^n$ be the region where all the ADFs in $\mathcal{A}$ are non-zero. We will show that ADF-SYMVOL monotonically converges, in the limit, to the exact weighted volume restricted to R; that is, ADF-SYMVOL converges to

$$\int_{\varphi \cap R} \prod_{x_i \in X_\varphi} p_i(x_i)\, dX_\varphi$$

The fascinating part here is that we do not impose any restrictions on the ADFs: they do not have to have any correspondence with the PDFs they approximate; they need only be step functions. Of course, in practice, the quality of the approximation dictates the rate of convergence, but we delay this discussion to Section 3.4.

The following theorem states convergence of ADF-SYMVOL; it assumes that HSAMPLE is applied iteratively and DECAY is only applied when HSAMPLE cannot find a model.

Theorem 3.15 (Monotone convergence to R)**.** *Assume* ADF-SYMVOL *is run on* (φ, D) *and a set of* ADFs $\mathcal{A}$ *that are non-zero for* $R \subset \mathbb{R}^n$*. Let* vol_n *be the value*

of vol after n *applications of* HSAMPLE. *Then,*

$$\lim_{n\to\infty} vol_n = \int_{\varphi\cap R} \prod_{x_i\in X_\varphi} p_i(x_i)\ dX_\varphi \quad \textit{and} \quad \forall j \geqslant k \geqslant 1.\, vol_j \geqslant vol_k$$

Proof. The algorithm constructs two series in parallel: the actual volume computation series $\sum v_n$ and the approximated series $\sum a_n$, where each v_n and a_n correspond to the actual and approximate volume of the nth sampled hyperrectangle (note that the latter is not explicitly maintained in the algorithm). Each series corresponds to a sequence of partial sums: Let

$$v_n^\Sigma = \sum_{j=1}^{n} v_j \qquad a_n^\Sigma = \sum_{j=1}^{n} a_j$$

It is maintained that

$$\forall n.\, v_n^\Sigma \leqslant EVol_{R\cap\varphi} \coloneqq \int_{R\cap\varphi} \prod_{x_i\in X_\varphi} p_i(x_i)\ dX_\varphi$$
$$\forall n.\, a_n^\Sigma \leqslant AVol \coloneqq \int_{R\cap\varphi} \prod_{x_i\in X_\varphi} step_i(x_i)\ dX_\varphi$$

Since $\{v_n^\Sigma\}_n$ and $\{a_n^\Sigma\}_n$ are non-decreasing sequences bounded from above, they converge to *some* limit; call the limits v^Σ and a^Σ, respectively. It does not matter what the value of a^Σ is, but we would like to ensure that v^Σ is actually equal to $EVol_{R\cap\varphi}$. Since the a_n determine which hyperrectangles we sample, the potential concern is that they negatively affect the limit v^Σ; we will prove below that this is not possible.

Suppose, for the sake of obtaining a contradiction, that our sequence of samples to construct $\{v_n^\Sigma\}_n$ and $\{a_n^\Sigma\}_n$ results in the limit v^Σ being strictly less than the actual weighted volume $EVol_{R\cap\varphi}$. Then there is some subregion $R' \subseteq R$ that is completely disjoint from the infinite set of hyperrectangles we sample and has non-zero weighted volume. In particular,

there must exist some hyperrectangle $H \subseteq R'$ contained in this unsampled region that also has non-zero weighted volume.

In the limit, a_n^Σ approaches a^Σ: by definition, for all $\varepsilon > 0$, there exists N such that for all $n > N$, $a^\Sigma - a_n^\Sigma < \varepsilon$. Let $\delta = \int_H \prod step(x)\, dX_\varphi$: at some point when we have fixed a threshold $\tau < \delta$ and have run out of samples in $R \setminus R'$ with $a_n \geqslant \tau$ (guaranteed when $a^\Sigma - a_n^\Sigma < \tau$ by letting $\varepsilon = \tau$) we reach a contradiction, since $H \subseteq R'$ would have satisfied the conditions to apply HSAMPLE. This property ensures that the limit $\nu^\Sigma = EVol_{R \cap \varphi}$.

Note that the above theorem directly gives us a way to approach the exact volume. Specifically, by performing runs of ADF-SYMVOL on subsets in an infinite partition of $\mathbb{R}^n$ induced by the ADFs, we can ensure that the sum over the ADF-SYMVOL processes approaches the exact volume.[3] For all i, let $\mathcal{A}_i$ be a set of ADFs corresponding to an ADF-SYMVOL process P_i, where $R_i \subset \mathbb{R}^n$ is the non-zero region of $\mathcal{A}_i$. We require an infinite set of P_i to partition $\mathbb{R}^n$: (*i*) for all $i \neq j$, $R_i \cap R_j = \emptyset$, and (*ii*) $\bigcup_{i=1}^{\infty} R_i = \mathbb{R}^n$. The following theorem formalizes the argument:

Theorem 3.16 (Monotone convergence)**.** *Let* $P_1, P_2, \ldots$ *be* ADF-SYMVOL *processes that partition* $\mathbb{R}^n$*. Assume an execution where each* P_i *executes infinitely often and each* P_i *performs* HSAMPLE *infinitely often, and let* vol_n *be the total computed volume across all* P_i *after* n *successful calls to* HSAMPLE*. Then,*

$$\lim_{n \to \infty} vol_n = \text{VOL}(\varphi, D) \quad \textit{and} \quad \forall j \geqslant k \geqslant 1.\, vol_j \geqslant vol_k$$

Proof. We require that ADF-SYMVOL calls HSAMPLE on each P_i (and its $\mathcal{A}_i$ defined over R_i) infinitely often: we can refer to H_n as the nth hyperrectangle obtained by HSAMPLE in the serialized execution. Clearly the partial sums $\{\sum_{j=1}^{n} \text{VOL}(H_j, D)\}_n$ form a non-decreasing series. It is bounded above by its supremum, which is exactly $\text{VOL}(\varphi, D)$ since each individual P_i

[3] Note that this odd dovetailing arises when the program includes primitive distributions with infinite support, such as Gaussians, as an ADF necessarily has finite support.

converges to the weighted volume restricted to R_i. This completes the proof, since the limit of any non-decreasing sequence bounded above by its supremum is identically its supremum.

Completeness in Verification. Given that we have established monotone convergence of ADF-SYMVOL, we can use it to construct a verification procedure that is complete whenever the postcondition is *robust* (Section 3.2, Definition 3.1). Without loss of generality, the definition gives us that for any subformula $\Pr[\varphi] > c$, we have that $\Pr[\varphi] \neq c$. Using this property, we can use ADF-SYMVOL to iteratively improve a lower and an upper bound for $\Pr[\varphi]$, one of which will prove or disprove the subformula $\Pr[\varphi] > c$.

Theorem 3.17 (Convergence of FairSquare)**.** *When the verification problem is* robust*, FairSquare eventually proves or disproves the postcondition.*

3.4 Implementation and Evaluation

In this section, we describe our implementation of FairSquare (including several optimizations) and evaluate its performance on a variety of fairness verification problems.

Implementation

We implemented our algorithms in a new tool called FairSquare, which employs Z3 (De Moura and Bjørner, 2008) for SMT solving and Redlog (Dolzmann and Sturm, 1997) for quantifier elimination. FairSquare accepts as input the population model and the decision-making program in a Python-like syntax, where the definitions of predicates in the probability events are provided as program annotations.

FairSquare computes upper and lower bounds for each probability in the postcondition using weighted volume computation. A *round* of

sampling involves (*i*) obtaining a sample (hyperrectangle) for each of the quantities, (*ii*) computing these samples' weighted volumes, (*iii*) updating the bounds on each quantity and (*iv*) checking if the bounds are precise enough to determine the validity of the postcondition, i.e., to prove *fairness* or *unfairness*. Rounds of sampling are performed until a proof is found or a timeout is reached.

Sample Maximization. A key optimization implemented in FairSquare is the maximization of hyperrectangles obtained during sampling. We use Z3's optimization capability to maximize and minimize the finite bounds of all hyperrectangles, while still satisfying the formula Ψ (in Figures 3.5 and 3.7). This process is performed greedily by extending a hyperrectangle in one dimension at a time to find a maximal hyperrectangle. If a dimension extends to infinity, then we drop that bound, thus resulting in an unbounded hyperrectangle.

Numerical Precision. All of the arithmetic performed by FairSquare is over arbitrary precision rationals, and therefore we do not encounter any loss of precision. The only place where floating point numbers appear is when we evaluate CDFs with scipy. We truncate (underapproximate) the result and convert it to a rational number. Truncating the results ensures that our implementation is sound—that at any point in our volume computation, the current volume is a lower bound—at the cost of a small possibility of incompleteness.

Evaluation

Next, we evaluate the effectiveness and performance of FairSquare. Specifically, we investigate the following questions:[4]

[4]All experiments are performed on an Intel Core i7 4.00GHz CPU with 16 GB of RAM.

Q1 Can FairSquare verify fairness properties of real machine-learned programs?

Q2 Do ADFs and sample maximization improve the performance of FairSquare?

Q3 Can FairSquare verify fairness properties other probabilistic analysis tools cannot?

Q4 Can FairSquare verify the benchmarks solved by other probabilistic analysis tools?

Benchmarks

Fairness Postconditions. In our experiments, we consider a *group fairness* postcondition augmented with a notion of qualification. We ultimately obtained our benchmarks by datamining a popular income dataset (UCI, 1996) used in related research on algorithmic fairness (Feldman et al., 2015; Zemel et al., 2013; Calders and Verwer, 2010); accordingly, our postconditions are defined in terms of that dataset's features. Specifically, they are of the form:

$$\frac{\Pr[\mathit{high\ income} \mid \mathit{female} \wedge \mathit{qual}(\boldsymbol{v})]}{\Pr[\mathit{high\ income} \mid \mathit{male} \wedge \mathit{qual}(\boldsymbol{v})]} > 1 - \epsilon \tag{3.2}$$

Suppose, for example, machine-learned models inferred from the dataset would be used to determine the salary of an employee: *high* ($>$ \$50,000) or *low*. We consider $\mathit{qual}(\boldsymbol{v})$ in two different scenarios—first, the case when *qual* is tautologically true, and second, when individuals are qualified if they are at least 18 years of age. In short, we would like to verify whether salary decisions are fair to qualified female employees.

Throughout our evaluation, we refer to the left-hand side of inequality 3.2 as the *fairness ratio*. We fix $\epsilon = 0.15$, and thus FairSquare terminates when either (*i*) its lower bound for the fairness ratio is at least 0.85, or (*ii*) its upper bound for the fairness ratio is at most 0.85.

Decision-Making Programs. We obtained our set of decision-making programs by training a variety of machine-learning models on the income dataset to classify *high* vs *low* income. Using the Weka machine learning suite (Hall et al., 2009), we learned 11 different decision-making programs (see, e.g., Bishop (2006) for background), which are listed in Figure 3.1: (*i*) four *decision trees*, named $\textsc{dt}_n$, where n is the number of conditionals in the program, and the number of variables and the depth of the tree each varies from 2 to 3; (*ii*) four *support vector machines* with linear kernels, named $\textsc{svm}_n$, where n is the number of variables in the linear separator; (*iii*) three *neural networks* using *rectified linear units* (Nair and Hinton, 2010), named $\textsc{nn}_{n,m}$, where n is the number of input variables, and m is the number of nodes in the single hidden layer.

As we will show in the next section, some of these programs do not satisfy the fairness property we consider. We introduced modifications of $\textsc{dt}_{16}$ and $\textsc{svm}_4$, called $\textsc{dt}^{\alpha}_{16}$ and $\textsc{svm}^{\alpha}_4$, that implement rudimentary forms of *affirmative action* for female applicants. For $\textsc{dt}^{\alpha}_{16}$, there is a 15% chance it will flip a decision to give the low salary; for $\textsc{svm}^{\alpha}_4$, the linear separator is moved to increase the likelihood of hiring.

Population Models. For our population models, we used three different probabilistic programs that were inferred from the same dataset: (*i*) a set of *independently distributed* variables (Ind), (*ii*) a *Bayesian network* using a simple graph structure (BN1), and (*iii*) the same Bayesian network, but with an integrity constraint in the form of an inequality between two of the variables (BN2). Note that the first model is sometimes a trivial case: since there is there is no dependence between variables, a program will be fair if it does not access an individual's sex; this simplicity serves well as a baseline for our evaluation. The Bayesian models permit correlations between the variables, allowing for more subtle sources of fairness or unfairness. The benchmarks we use are derived from each combination of population models with decision-making programs.

Effectiveness of FairSquare

Figure 3.1 shows the results of applying FairSquare to 39 fairness verification problems, as described earlier. Only the instances using the tautologically true notion of qualification are shown, since the qualitative results are near-identical to the non-trivial qualification. FairSquare was able to solve 32 of the 39 problems within a timeout period of 900 seconds each, proving 21 fair and 11 unfair.

Consider the results for DT_4: FairSquare proved it fair with respect to the independent population model after 0.5 seconds of an initial quantifier elimination procedure and 1.3 seconds of the actual volume computation algorithm, which required 10 SMT queries. The more sophisticated Bayesian network models took longer for sampling, but due to the correlations between variables, were proved unfair.

In contrast, consider the results for DT_{44} under the Bayes Net 1 population model: FairSquare was unable to conclude fairness or unfairness after 900 seconds of volume computation (denoted by TO in the *Vol* column). The lower and upper bounds of the fairness ratio it had computed at that time are listed in the *Res* column: in this case, the value of the fairness ratio is within $[0.70, 0.88]$, which is not precise enough for the $\epsilon = 0.15$ requirement (but would be precise enough for ϵ outside of $[0.12, 0.30]$).

In general, all conclusive results using the independent population model were proved to be fair, as expected, but many are unfair with respect to the clusters and Bayes net models because of the correlations those population models capture. This difference illustrates the sensitivity of fairness to the population model; in particular, none of the decision trees syntactically access *sex*, yet several are unfair.

Figure 3.1 shows that the affirmative action modifications in DT_{16}^{α} and SVM_4^{α} are sufficient to make the programs fair with respect to every population model without substantially impacting the training set accuracy.

In summary, the answer to **Q1** is that **FairSquare is powerful enough**

Decision Program	Acc	Population Model: Independent				Bayes Net 1				Bayes Net 2			
		Res	#	*Vol*	*QE*	*Res*	#	*Vol*	*QE*	*Res*	#	*Vol*	*QE*
DT_4	0.79	✓	10	1.3	0.5	✗	12	2.2	0.9	✗	18	6.6	2.2
DT_{14}	0.71	✓	20	4.2	1.4	✓	38	52.3	11.4	✓	73	130.9	33.6
DT_{16}	0.79	✓	21	7.7	2.0	✗	22	15.3	6.3	✗	22	38.2	14.3
DT_{16}^{α}	0.76	✓	18	5.1	3.0	✓	34	32.0	8.2	✓	40	91.0	19.4
DT_{44}	0.82	✓	55	63.5	9.8	✗	113	178.9	94.3	✗	406	484.0	222.4
SVM_3	0.79	✓	10	2.6	0.6	✗	10	3.7	1.7	✗	10	10.8	6.2
SVM_4	0.79	✓	10	2.7	0.8	✗	18	13.3	3.1	✗	14	33.7	20.1
SVM_4^{α}	0.78	✓	10	3.0	0.8	✓	22	15.7	3.2	✓	14	33.4	63.2
SVM_5	0.79	✓	10	8.5	1.3	✗	10	12.2	6.3	TO_q	-	-	TO
SVM_6	0.79	$^{0.02}_{35.3}$	634	TO	2.4	$^{0.09}_{3.03}$	434	TO	12.8	TO_q	-	-	TO
$\text{NN}_{2,1}$	0.65	✓	78	21.6	0.8	✓	466	456.1	3.4	✓	154	132.9	7.2
$\text{NN}_{2,2}$	0.67	✓	62	27.8	2.0	✓	238	236.5	7.2	✓	174	233.5	18.2
$\text{NN}_{3,2}$	0.74	$^{0.03}_{674.7}$	442	TO	10.0	$^{0.00}_{5.24}$	34	TO	55.9	TO_q	-	-	TO

Res: ✓ for fair; ✗ for unfair. *Vol*: time (s) of the sampling procedure; #: number of SMT calls. If sampling timed out (900s), *Res* denotes the latest bounds on the fairness ratio. *QE*: time (s) of the quantifier elimination procedure used prior to sampling; if *QE* times out (900s), no sampling is performed, denoted by TO_q for *Res*. *Acc*: training set accuracy of the programs.

Table 3.1: Results of FairSquare applied to 39 fairness verification problems.

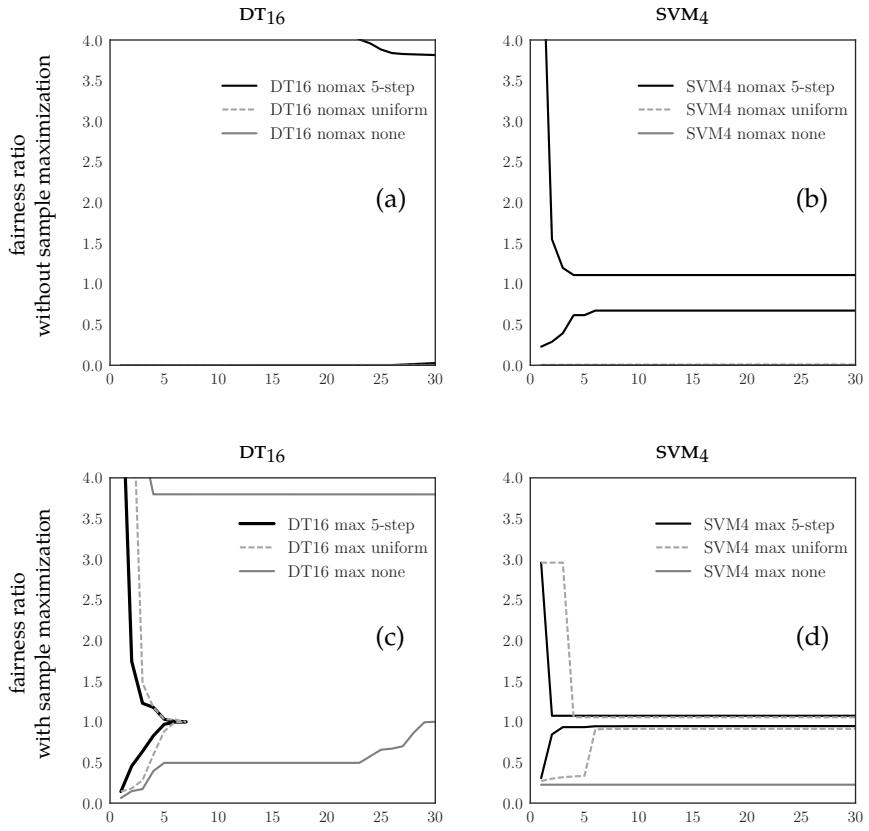


Figure 3.8: Fairness ratio vs. rounds of sampling for DT$_{16}$ and SVM$_4$ (Ind pop model) differing on ADFs and sample maximization. In (c) two runs end at the exact value. Outside the visible range are: (a)(b) upper and lower bounds of *uniform* and *none*; (d) upper bounds of *none*.

to reason about group fairness for many non-trivial machine-learned programs.

Effect of Parameters

The experiments in Figure 3.1 were all performed using sample maximization (as described at the beginning of this section); additionally, to guide volume computation, all Gaussian distributions with mean μ and

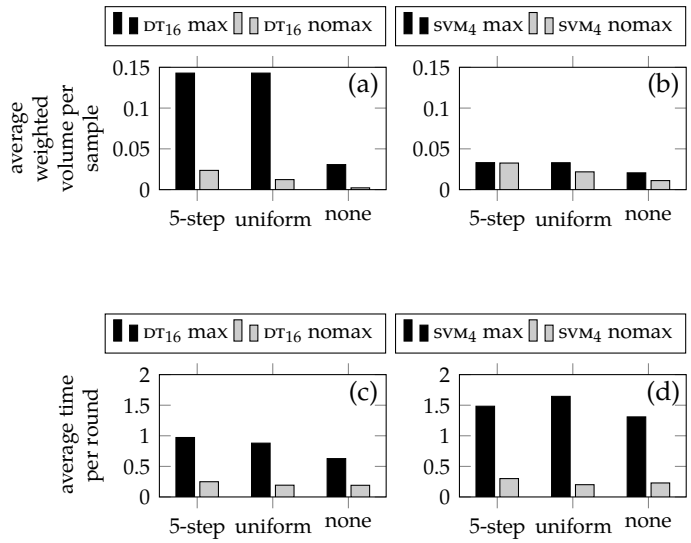


Figure 3.9: Effect of optimizations on FairSquare for DT$_{16}$ and SVM$_4$ (Ind pop model). (a) and (b) show the average weighted volume per sample (averaged across all probabilities). (c) and (d) show the average time (s) per round of sampling.

variance σ^2 use ADFs (see Section 3.3) with 5 equal-width steps spanning $(\mu - 3\sigma^2, \mu + 3\sigma^2)$—analogous to Figure 3.6(a). In this section, we explore the effects of the approximate density functions and of the sample maximization optimizations. These results are captured in Figures 3.8 and 3.9.

There are three instances of ADFs in Figure 3.8 used to guide the sampling to high-probability regions: (*i*) *none* indicates that no ADF is used, i.e., we used SYMVOL instead of ADF-SYMVOL; (*ii*) *uniform* indicates that each gauss(μ, σ^2) is approximated by a uniform function spanning $(\mu - 3\sigma^2, \mu + 3\sigma^2)$ (similar to Figure 3.6(c)); and (*iii*) *5-step* indicates that each Gaussian is approximated by a step function of 5 equal-width regions spanning that same domain (similar to Figure 3.6(a)). Another variable, *max* or *nomax*, denotes whether the sample maximization optimization is enabled.

Each combination of these techniques is run on two of our benchmarks:

DT$_{16}$ and SVM$_4$ under the independent population model. Figure 3.8(a) and (b) show how convergence to the fairness ratio is improved by the choice of ADFs when sample maximization is not employed: in particular, the runs using *uniform* and *none* are not even visible, as the bounds never fall within $[0.01, 4.0]$. Plots (c) and (d) show that when sample maximization *is* employed, the choice between the uniform and 5-step ADFs is not as substantial on these benchmarks, although (*i*) the better approximation gets better bounds faster, and (*ii*) using none results in substantially worse bounds.

Figure 3.9 plot (a) and (b) show that employing ADFs and using sample maximization each increases the average weighted volume per sample, allowing volume computation to be done with fewer samples. Plots (c) and (d) illustrate the trade-off: the average time per sampling round tends to be greater for more complex optimizations.

We present these results for two particular problems and observe the same results across our suite. In summary, the answer to **Q2** is that **ADFs and sample maximization improve the performance of FairSquare**, and FairSquare requires both of these features to verify most benchmarks.

Comparison to Other Tools

We ran our benchmarks on the two other recent probabilistic program analysis tools that accept the same class of problems and provide exact guarantees on probabilities. First, we compare to the tool of Sankaranarayanan et al. (2013) (which we denote VC),[5] which is algorithmically similar to our tool: it finds bounds for probabilities on individual paths by approximating convex polytopes with bounding and inscribed hyperrectangles. Second, we compare to PSI (Gehr et al., 2016), which symbolically computes representations of the posterior distributions of variables.

[5] Acquired directly from the authors.

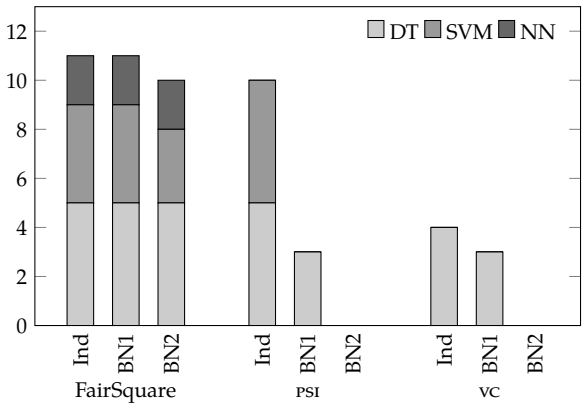


Figure 3.10: Comparison of the number of benchmarks that FairSquare, PSI, and VC were able to solve.

Figure 3.10 shows the number of benchmarks solved per category per tool. Tools were deemed to have failed on a benchmark when they timed out after a 900s period or returned an inconclusive solution (in the case of PSI). For instance, in the case of the population model BN1, FairSquare solved 11 benchmarks, while PSI and VC only solved 3 (the decision trees). For BN2, neither PSI nor VC was able to complete any benchmark.

The figure illustrates some qualitative properties of the applicability of the tools. In general, most of the decision trees are solvable because they partition the decision space with inequalities between a single variable and a *constant*. However, inequalities involving multiple variables can result in (*i*) the lack of closed form posterior CDFs, as reflected in the output of PSI, and (*ii*) angled boundaries in the decision space that are hard to approximate with hyperrectangles; these inequalities occur in the SVMs, neural networks, and the BN2 population model. Consequently, VC fails to produce good bounds in these cases. Similarly, PSI fails because the integrals do not have closed forms or cannot be constructed within the timeout period.

In summary, the answer to **Q3** is that **FairSquare can verify fairness properties that other tools cannot** and therefore extends the class of prob-

lems that can be solved by state-of-the-art probabilistic analysis tools.

We now discuss the results of applying the weighted volume computation algorithm of FairSquare to the benchmarks from VC. (We omit a comparison to the benchmarks from PSI, since the output of PSI is a posterior distribution—which can be used to compute probabilities, but not vice-versa.) We first focus on VC's three loop-free benchmark programs, which have thousands of paths; VC computes various probabilities within two hours for each program. In FairSquare, however, the quantifier elimination procedure employed *before beginning sampling* does not terminate within two hours.

Second, we consider two of VC's programs, *cart* and *invPend*, which have loops explicitly bounded by constants and could be encoded in our framework using loop unrolling. The programs' loops have maximum depths of 5 and 10 iterations, respectively, (and the loop bodies contain if statements and probabilistic assignments). The fully unrolled versions of these programs are very large and cause FairSquare's quantifier elimination to timeout; to better understand the limitations of FairSquare, we tried unrolling the programs up to 4 and 7 iterations, respectively, which were the largest unrollings for which the quantifier elimination procedure would terminate within two hours. However, we found that the program paths corresponding to these fewer number of loop iterations have zero probability mass, so FairSquare was not able to compute non-trivial bounds for any probabilities.

In summary, the answer to **Q4** is that **FairSquare currently cannot solve the verification benchmarks solved by the tool VC**.

Quantifier elimination is difficult on programs with this many paths. VC is well-designed for these tasks because it performs weighted volume computation only on the most *important* paths. Specifically, VC heuristically picks a program path π through simulation, with the assumption that traversed paths will likely have a larger probability mass for the event

of interest. vc then computes the probability of executing π and a given property being true at the end. By iteratively choosing more and more paths through the program, it improves the computed bounds. Our approach considers the full set of paths symbolically by encoding them as a formula. As described above, this methodology works well for decision-making programs. Our evaluation indicates that our two techniques can complement each other, providing an important direction for future work. Specifically, we plan to investigate a lazily-evaluated quantifier elimination procedure, where we heuristically sample disjuncts (i.e., program paths), so that FairSquare can scale to benchmarks used by vc—where explicit quantifier elimination is prohibitively expensive.

3.5 Discussion and Related Work

A number of very interesting applications of program analysis have been explored in the probabilistic setting: reasoning about cyber-physical systems (Sankaranarayanan et al., 2013), proving differential privacy of complex algorithms (Barthe et al., 2014), reasoning about approximate programs and hardware (Carbin et al., 2013), synthesizing control programs (Chaudhuri et al., 2014), amongst many others. In this chapter, we turned our attention to the problem of *verifying fairness of decision-making programs*.

Algorithmic Fairness. Our work is inspired by concern in the fairness of modern decision-making programs (Zarsky, 2014; Barocas and Selbst, 2014). A number of works have explored algorithmic fairness (Zemel et al., 2013; Feldman et al., 2015; Hardt et al., 2016; Dwork et al., 2012; Calders and Verwer, 2010; Pedreshi et al., 2008; Datta et al., 2016, 2015). Accordingly, we developed a general-purpose framework for verifying a general class of probabilistic fairness properties. In contrast, for example, recent work by John et al. (2020) formally verifies individual fairness for programs using a global, deterministic definition.

Most works are interested in fairness from a machine learning perspective: how does one learn a fair classifier from data? For example, Zemel et al. (2013) and Feldman et al. (2015) aim to transform training data so as to erase correlations between the sensitive attributes of individuals and the rest of their features. Within this context, classification utility is important. Hardt et al. (2016) recently proposed a new fairness definition—equality of opportunity—that improves on demographic parity in terms of classification utility. Recently, Celis et al. (2019) provide a mechanism to support even non-convex fairness-related loss functions.

Discrimination in *black-box* systems has been studied through the lens of statistical analysis (Sweeney, 2013; Datta et al., 2015, 2016). Notably, Datta et al. (2015) created an automated tool that analyzes online advertising: it operates dynamically by surveying the ads produced by Google. Since then, in white-box systems, Albarghouthi and Vinitsky (2019) have proposed a framework where developers can specify fairness properties through code annotations that are then dynamically checked at run-time.

Probabilistic Abstract Interpretation. We refer the reader to Gordon et al. (2014) for a thorough survey on probabilistic program analysis. A number of works tackled analysis of probabilistic programs from an abstract interpretation perspective (Monniaux, 2001b, 2000, 2001a; Mardziel et al., 2011; Claret et al., 2013). The comparison between our solution through volume computation and abstract interpretation is perhaps analogous to SMT solving and software model checking versus abstract interpretation. For example, techniques proposed by Monniaux (2000) sacrifice precision of the analysis (through joins, abstraction, etc.) for the benefit of efficiency. Our approach, on the other hand, is aimed at eventually producing a proof, or iteratively improving probability bounds while guaranteeing convergence.

Sampling-Based Inference. In probabilistic verification, some techniques perform probabilistic inference by compiling programs or program paths

to Bayesian networks (Koller and Friedman, 2009) and applying hypothesis testing (Sampson et al., 2014). The verification technique proposed by Sampson et al. (2014) applies to properties of the form $\Pr[\varphi] > c$. The approach relies on concentration inequalities to determine a number of samples (executions) that would provide a result within an ϵ additive error with $1-\delta$ probability. In the case of properties where we have a ratio over two probabilities—like the ones considered here—we cannot a priori determine the number of samples required to achieve (ϵ, δ) guarantees.

Probabilistic programming languages often rely on sampling to approximate the posterior distribution of a program. The Church programming language (Goodman et al., 2008), for instance, employs the *Metropolis–Hastings* algorithm (Chib and Greenberg, 1995), a *Markov Chain Monte Carlo* (MCMC) technique. In MCMC techniques, there is usually no guarantee on how different the Markov chain is from the actual distribution at any point in execution, although the Markov chain is guaranteed to converge in the limit.

Volume Computation. The computation of weighted volume is known to be hard—even for a convex polytope, volume computation is #P-hard (Khachiyan, 1993). Two general approaches exist: approximate and exact solutions. Note that in general, any approximate technique at best can prove facts *with high probability*.

Our volume computation algorithm is inspired by (*i*) the formula decomposition procedure of Li et al. (2014), where quantifier elimination is used to underapproximate an LRA constraint as a Boolean combination of monadic predicates; and (*ii*) the technique for bounding the weighted volume of a polyhedron introduced by Sankaranarayanan et al. (2013), which is the closest volume computation work to ours. (The general technique of approximating complex regions with unions of orthogonal polyhedra is well-studied in the hybrid systems literature (Bournez et al., 1999).)

A number of factors differentiate our work from Sankaranarayanan et al.

(2013), which we compared with experimentally in Section 3.4. First, our approach is more general, in that it can operate on Boolean formulas over linear and polynomial inequalities, as opposed to just conjunctions of linear inequalities. Second, our approach employs ADFs to guide the sampling of hyperrectangles with large volume, which, as we have demonstrated experimentally, is a crucial feature of our approach. Third, we provide theoretical convergence guarantees.

FairSquare's volume computation algorithm relies on SMT encodings of the program behavior that did not scale well for neural networks. Since then, Converse et al. (2020) provide specialized techniques for probabilistic inference of neural networks.

LattE is a tool that performs exact integration of polynomial functions over polytopes (De Loera et al., 2012). Belle et al. (2015b, 2016) compute the volume of a linear real arithmetic (LRA) formula by, effectively, decomposing it into DNF—a set of polytopes—and using LattE to compute the volume of each polyhedron with respect to piece-wise polynomial densities. Our volume computation algorithm is more general in that it (*i*) handles formulas over real closed fields, which subsumes LRA, and (*ii*) handles probability distributions for which we can evaluate the CDF. Our implementation also supports polynomial approximations of the ADFs. Polynomial approximations provide better samples, but since the polynomials introduce non-linear constraints, the actual SMT calls become dramatically slower due to the lack of scalable solvers for non-linear arithmetic. This includes Z3's non-linear solver (Jovanović and de Moura, 2013), which implements a variant of cylindrical algebraic decomposition (CAD) (Basu et al., 2006), a technique for solving non-linear constraints implemented in tools such as Mathematica and Maple. Although Z3 was shown to be faster than all other non-linear solvers (Jovanović and de Moura, 2013), it still does not scale to formulas of size we consider, making polynomial approximations currently ineffective in practice. The same applies to Redlog (Dolzmann

and Sturm, 1997), which we used for quantifier elimination, and other state-of-the-art tools such as QEPCAD (Brown, 2003); they implement CAD and are comparable to Z3's non-linear solver in performance.

Chistikov et al. (2015) present a framework for approximate counting with probabilistic guarantees in SMT theories, which they specialize for bounded LRA. In contrast, our technique (*i*) handles unbounded formulas in LRA as well as real closed fields, (*ii*) handles arbitrary distributions, and (*iii*) provides converging lower-bound guarantees. It is important to note that there is a also a rich body of work investigating randomized polynomial algorithms for approximating the volume of a polytope, beginning with the seminal work of Dyer et al. (1991) (see Vempala (2005) for a survey).

Probabilistic Verification with Model Counting. A number of works have also addressed probabilistic analysis through symbolic execution (Filieri et al., 2013; Sankaranarayanan et al., 2013; Geldenhuys et al., 2012; Sampson et al., 2014). Filieri et al. (2013) and Geldenhuys et al. (2012) attempt to find the probability a safety invariant is preserved. Both methods reduce to a weighted model counting approach and are thus effectively restricted to variables over finite domains. Note that our technique is more general than a model counting approach, as we can handle discrete cases with a proper encoding of the variables into a continuous domain without loss of precision.

4 EFFICIENT SYNTHESIS WITH PROBABILISTIC CONSTRAINTS: REPAIRING UNFAIR PROGRAMS

In Chapter 3 we detailed a method for *verifying* whether a program meets a probabilistic correctness property; In this chapter, we will explore the dual problem of *synthesizing* programs that meet these properties by construction. 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 (Angwin et al., 2016) (as we've discussed prior). These programs can be generated automatically as classifiers using machine learning or can be written by hand using expert insight.

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 (Könighofer and Bloem, 2011; Mechtaev et al., 2015; D'Antoni et al., 2016; Von Essen and Jobstmann, 2015; Jobstmann et al., 2005). The world, however, is uncertain, and program correctness is not always a Boolean, black-or-white property. In particular, our goal is to synthesize programs that both (*i*) meet the probabilistic notion of correctness and (*ii*) optimize some quantitative objective, e.g. accuracy.

Technique: Distribution-guided inductive synthesis. We propose a novel program synthesis technique that we call *distribution-guided inductive synthesis* (DIGITS). The overall flow of DIGITS is illustrated in Figure 4.1. Suppose we have a program $\hat{\mathsf{P}}$ such that $\{pre\}\mathsf{P}\{post\}$ 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 $\hat{\mathsf{P}}$. To do so, DIGITS tightly integrates three phases:

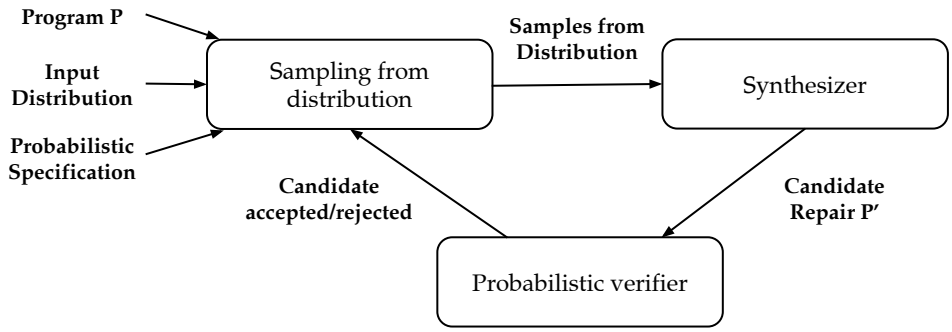


Figure 4.1: Abstract, high-level view of distribution-guided inductive synthesis (DIGITS)

Sampling Since the precondition *pre* specifies that inputs to the program are given by a probability distribution $x \sim \mathbb{D}$, DIGITS begins by sampling a finite set S of program inputs from $\mathbb{D}$—we call S the set of *samples*. The set S is used to sidestep having to deal with probability distributions directly in the synthesis process.

Synthesis The second step is a synthesis phase, where DIGITS searches for a set of candidate programs $\{P_1, \ldots, P_n\}$ such that each P_i classifies the set of samples S differently.

Quantitative verification Every generated candidate program P is checked for correctness and for close *semantic distance* with $\hat{P}$. Specifically, DIGITS employs an automated probabilistic inference technique, e.g. the symbolic probabilistic inference described in Chapter 3.

Somewhat remarkably, DIGITS posseses desirable convergence properties: under certain conditions, with high probability, DIGITS is guaranteed to synthesize near-optimal correct programs. Furthermore, a detailed analysis of the algorithm that draws from concepts in computational learning theory (Kearns and Vazirani, 1994) reveals that DIGITS performs its search quite efficiently. The layout of this chapter is as follows:

- In Section 4.1 we describe our *probabilistic program synthesis problem* that combines a Boolean probabilistic correctness property with a quantitative objective.
- In Section 4.2 we present *distribution-guided inductive synthesis* (DIGITS), a novel synthesis methodology for our probabilistic synthesis problem, and prove its (probable, near-)optimality.
- In Section 4.3 we describe an efficient, trie-based implementation strategy for DIGITS and analyze its complexity.
- In Section 4.4 we discuss an improved variant of DIGITS, which we call τ-DIGITS, that makes the optimality guarantees of DIGITS more practical.
- In Sections 4.5 and 4.6 we describe our implementations of these algorithms apply them to a range of benchmarks, including illustrative examples that elucidate our theoretical analysis, probabilistic repair problems of unfair programs, and probabilistic synthesis of controllers.

Proofs of theorems stated throughout this chapter can be found in Appendix B.3. This chapter is based on the work of Albarghouthi et al. (2017a) and Drews et al. (2019)

4.1 Preliminaries

In this section, we present a variant of the program model used in Chapter 3 and define the probabilistic synthesis problem.

Program Model. We will consider sets of programs defined as *program sketches* (Solar-Lezama, 2008) in a simple grammar (as before), where a program is written in a loop-free language, and "holes" defining the

sketch replace some constant terminals in expressions.[1] The syntax of the language is defined below:

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

Here, P is a program, V is the set of variables appearing in P, E (resp. B) is the set of linear arithmetic (resp. Boolean) expressions over V (where, again, constants in E and B can be replaced with holes), and $\mathsf{V} \leftarrow \mathsf{E}$ is an assignment. We assume a vector $\boldsymbol{\nu}_\mathrm{I}$ of variables in V that are inputs to the program. We also assume there is a single Boolean variable $\nu_\mathrm{r} \in \mathsf{V}$ that is returned by the program.[2] All variables are real-valued or Boolean. Given a vector of constant values $\mathbf{x}$, where $|\mathbf{x}| = |\boldsymbol{\nu}_\mathrm{I}|$, we use $\mathsf{P}(\mathbf{x})$ to denote the result of executing P on the input $\mathbf{x}$.

In our setting, the inputs to a program are distributed according to some *joint probability distribution* $\mathbb{D}$ over the variables $\boldsymbol{\nu}_\mathrm{I}$. Semantically, a program P is denoted by a *distribution transformer* $[\![\mathsf{P}]\!]$, whose input is a distribution over values of $\boldsymbol{\nu}_\mathrm{I}$ and whose output is a distribution over $\boldsymbol{\nu}_\mathrm{I}$ and ν_r.

A program also has a *probabilistic postcondition*, *post*, defined as an inequality over terms of the form $\Pr[\mathsf{B}]$, where B is a Boolean expression over $\boldsymbol{\nu}_\mathrm{I}$ and ν_r. Specifically, a probabilistic postcondition consists of Boolean combinations of the form $e > c$, where $c \in \mathbb{R}$ and e is an arithmetic expression over terms of the form $\Pr[\mathsf{B}]$, e.g., $\Pr[\mathsf{B}_1]/\Pr[\mathsf{B}_2] > 0.75$.

Given a triple $(\mathbb{D}, \mathsf{P}, \mathit{post})$, we say that P is *correct* with respect to $\mathbb{D}$ and *post*, denoted $[\![\mathsf{P}]\!](\mathbb{D}) \models \mathit{post}$, *iff* *post* is true on the distribution $[\![\mathsf{P}]\!](\mathbb{D})$.

[1] In the case of loop-free program sketches as considered in our program model, we can convert the input-output relation into a real arithmetic formula that guaranteedly has finite VC dimension (Goldberg and Jerrum, 1995). This is important for our convergence and complexity guarantees, which we discuss in Sections 4.2 and 4.3.

[2] Restricting the output to Boolean is required by the algorithm; other output types can be turned into Boolean by rewriting. See, e.g., thermostat example in Section 4.6.

Example 4.1. *Consider the set of intervals of the form* $[0, a] \subseteq [0, 1]$ *and inputs* x *uniformly distributed over* $[0, 1]$ *(i.e.* $\mathbb{D} = \mathit{Uniform}[0, 1]$*). We can write inclusion in the interval as a (C-style) program (left) and consider a postcondition stating that the interval must include at least half the input probability mass (right):*

```
if (0 <= x && x <= a) {
    return 1;
}
return 0;
```

$$\Pr_{x \sim \mathbb{D}}[P(x) = 1] \geqslant 0.5$$

Let P_c *denote the interval program where* a *is replaced by a constant* $c \in [0, 1]$*. Observe that* $[\![P_c]\!](\mathbb{D})$ *describes a joint distribution over* (x, v_r) *pairs, where* $[0, c] \times \{1\}$ *is assigned probability measure* c *and* $(c, 1] \times \{0\}$ *is assigned probability measure* $1 - c$*. Therefore,* $[\![P_c]\!](\mathbb{D}) \models$ *post if and only if* $c \in [0.5, 1]$*.*

Synthesis Problem. DIGITS outputs a program that is approximately "similar" to a given functional specification and that meets a postcondition. This functional specification is some input-output relation which we quantitatively want to match as closely as possible; in general, our input distribution induces a notion of distance between programs that we wish to minimize:

$$d_{\mathbb{D}}(P_1, P_2) \coloneqq \Pr_{x \sim \mathbb{D}}[P_1(x) \neq P_2(x)] \tag{4.1}$$

Specifically, we want to minimize the *error* (distance) of the output program P from the functional specification $\hat{P}$, denoted as follows (when $\mathbb{D}$ and $\hat{P}$ are clear from context):

$$\mathrm{Er}(P) \coloneqq d_{\mathbb{D}}(P, \hat{P}) = \Pr_{x \sim \mathbb{D}}[P(x) \neq \hat{P}(x)] \tag{4.2}$$

(Note that we represent the functional specification as a program.) The postcondition, on the other hand, is Boolean, and therefore we always want it to be true.

Stated formally, given some set of programs $\mathcal{P}$, a distribution over

```
function DIGITS(𝔻, P̂, post, 𝒫, m)
    S ← {x ∼ 𝔻 | i ∈ [1, ..., m]}
    progs ← ∅
    for all f : S → {0, 1} do
        P ← 𝒪_syn({(x, f(x)) | x ∈ S})
        if P ≠ ⊥ then
            progs ← progs ∪ {P}
    res ← {P ∈ progs | 𝒪_ver(𝔻, P, post)}
    return argmin_{P∈res} {𝒪_err(P)}
```

Figure 4.2: Naive DIGITS algorithm

program inputs $\mathbb{D}$, a postcondition *post*, and a functional specification $\hat{P}$, our goal is to find an optimal correct program P^* (assuming one exists): let $C = \{P \in \mathcal{P} \mid [\![P]\!](\mathbb{D}) \models \mathit{post}\}$ in $P^* = \operatorname{argmin}_{P \in C} \mathrm{Er}(P)$.

4.2 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.

Figure 4.2 shows a simplified, naive version of DIGITS, which employs a *synthesize-then-verify* approach. The idea of DIGITS is to utilize non-probabilistic synthesis techniques to synthesize a set of programs, and then apply a probabilistic verification step to check if any of the synthesized programs is a solution. Throughout, we assume we have access to a number of sound and complete oracles $\mathcal{O}_{\mathrm{syn}}$, $\mathcal{O}_{\mathrm{ver}}$, and $\mathcal{O}_{\mathrm{err}}$.

Specifically, this "Naive DIGITS" begins by sampling an appropriate number of inputs from the input distribution and stores them in the set S. Second, it iteratively explores each possible function f that maps the input samples to a Boolean and invokes a synthesis oracle $\mathcal{O}_{\mathrm{syn}}$ to synthesize a program P that implements f, i.e. that satisfies the set of input–output

examples in which each input $x \in S$ is mapped to the output $f(x)$. Naive DIGITS then finds which of the synthesized programs satisfy the postcondition (the set *res*); we assume that we have access to a probabilistic verifier $\mathcal{O}_{\text{ver}}$ to perform these computations. Finally, the algorithm outputs the program in the set *res* that has the lowest error with respect to the functional specification, once again assuming access to another oracle $\mathcal{O}_{\text{err}}$ that can measure the error.

Note that the number of such functions $f : S \to \{0, 1\}$ is exponential in the size of $|S|$. In Section 4.3, we present an efficient search strategy that considers asymptotically fewer functions f. The naive version described here is, however, sufficient to discuss the convergence properties of the full algorithm.

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 m of the sample set.

Throughout this section we assume we are given a set of programs $\mathcal{P}$, a functional specification $\hat{P}$, an input distribution $\mathbb{D}$, and a postcondition *post*, such that there exists an optimal solution $P^* \in \mathcal{P}$ to the corresponding probabilistic synthesis problem. The relationship between $\hat{P}$, $\mathcal{P}$, the programs which satisfy *post*, and P^* is visualized in Figure 4.3(a).

To state our main theorem, we need to recall the concept of Vapnik–Chervonenkis (VC) dimension from computational learning theory (Kearns and Vazirani, 1994). Intuitively, the VC dimension captures the expressiveness of our set of ($\{0, 1\}$-valued) programs $\mathcal{P}$. Given a set of inputs S, we say that $\mathcal{P}$ *shatters* S iff, for every partition of S into sets $S_0 \sqcup S_1$, there exists a program $P \in \mathcal{P}$ such that (*i*) for every $x \in S_0$, $P(x) = 0$, and (*ii*) for every $x \in S_1$, $P(x) = 1$.

Definition 4.2 (VC Dimension). *The* VC dimension *of a set of programs* $\mathcal{P}$ *is the largest integer* d *such that there exists a set of inputs* S *with cardinality* d *that is shattered by* $\mathcal{P}$.

Example 4.3. *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.*

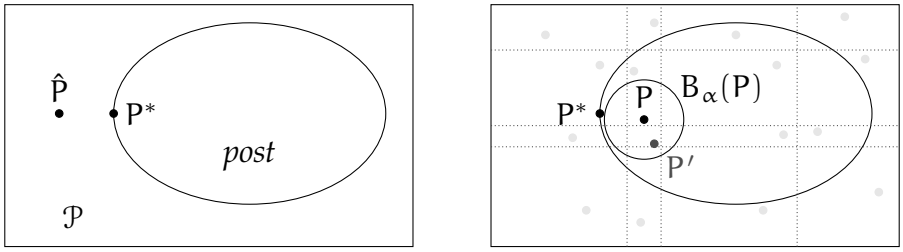
Additionally, we define the function $\text{VCcost}(\varepsilon, \delta, d) = \frac{1}{\varepsilon}(4\log_2(\frac{2}{\delta}) + 8d\log_2(\frac{13}{\varepsilon}))$ (Blumer et al., 1989), which we will use in the following theorems.

Lemma 4.4 (Error Bound of DIGITS). *Given a set of programs* $\mathcal{P}$ *with finite VC dimension* d, *for any fixed program* $P \in \mathcal{P}$ *and parameters* $\varepsilon > 0$ *and* $\delta > 0$, *if* $S \sim \mathbb{D}^m$ *with* $m \geqslant \text{VCcost}(\varepsilon, \delta, d)$, *then with probability* $\geqslant 1 - \delta$ *we have that* DIGITS *enumerates some candidate program* P' *such that* $d_{\mathbb{D}}(P, P') \leqslant \varepsilon$.

Lemma 4.4 extends the classic notion of learnability of concept classes with finite VC dimension (Blumer et al., 1989) to probabilistic program repair. Intuitively, if a $\mathcal{P}$ has finite VC dimension, any function that correctly synthesizes from finitely many samples in $\mathbb{D}$ will get arbitrarily close to a target solution—including P^*—with polynomially many samples. Lemma 4.4, 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 define a notion of *robustness* of the postcondition.

Definition 4.5 (α-Robust Programs). *Fix an input distribution* $\mathbb{D}$, *a postcondition post, and a set of programs* $\mathcal{P}$. *For any* $P \in \mathcal{P}$ *and any* $\alpha > 0$, *denote the*



(a) (b)

Figure 4.3: Visualization of aspects of DIGITS: (a) Programs that satisfy *post* are a subset of $\mathcal{P}$. (b) Samples split $\mathcal{P}$ into 16 regions, each with a candidate program. If P is α-robust, with high probability DIGITS finds P' close to P; if P is close to P^*, so is P'.

closed α-ball centered at P *as* $B_\alpha(P) = \{P' \in \mathcal{P} \mid d_{\mathbb{D}}(P, P') \leqslant \alpha\}$. *We say a program* P *is* α-robust *if* $\forall P' \in B_\alpha(P). [\![P']\!](\mathbb{D}) \models \mathit{post}$.

Figure 4.3(b) visualizes how the convergence of DIGITS follows from α-robustness: if P is α-robust, then DIGITS invoked on a sufficiently large set of samples S will, with high probability, encounter a function $f : S \to \{0, 1\}$ where every program consistent with f contained in $B_\alpha(P)$. Thus if P' is the result of $\mathcal{O}_{\text{syn}}(f)$, then $d_{\mathbb{D}}(P, P') \leqslant \alpha$, and P' satisfies *post*. We can now give our main theorem, which formalizes this property.

Theorem 4.6 (Convergence of DIGITS)**.** *Assume there exists an* α*-robust program* P *(for some* $\alpha > 0$*). Let* d *be the VC dimension of* $\mathcal{P}$*. For all bounds* $0 < \varepsilon \leqslant \alpha$ *and* $\delta > 0$*, if* $m \geqslant \text{VCCOST}(\varepsilon, \delta, d)$*, then with probability* $\geqslant 1-\delta$ *we have that* DIGITS *enumerates a program* P' *with* $d_{\mathbb{D}}(P, P') \leqslant \varepsilon$ *and* $[\![P']\!](\mathbb{D}) \models \mathit{post}$.

Corollary 4.7 (Convergence to P^*)**.** *In particular, if* P^* *is* α*-robust, and* ε*,* δ*, and* m *are constrained as above, and* $\text{DIGITS}(\mathbb{D}, \hat{P}, \mathit{post}, \mathcal{P}, m) = P$*, then with probability* $\geqslant 1-\delta$ *we have that* $P \neq \bot$*,* $\text{Er}(P) \leqslant \text{Er}(P^*)+\varepsilon$*, and* $[\![P]\!](\mathbb{D}) \models \mathit{post}$.

Theorem 4.6 and Corollary 4.7 represent the heart of the convergence result. However, there are two major technicalities.

First, P^* usually is not α-robust; in particular, if there exists $P \in B_\alpha(P^*)$ with $\mathrm{Er}(P) \leqslant \mathrm{Er}(P^*)$, 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 Figure 4.3(a). However, Theorem 4.6 still guarantees that with high probability, DIGITS will find a solution arbitrarily close to *any* α-robust program $P \in \mathcal{P}$; 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 4.8 (Weak convergence to P^*)**.** *For $\alpha > 0$, let $A \subseteq \mathcal{P}$ be the set of α-robust programs. Let $\Delta = \min_{P \in A}\{d_{\mathbb{D}}(P^*, P)\}$. If ε, δ, and m are constrained as above, and $\textsc{digits}(\mathbb{D}, \hat{P}, \mathit{post}, \mathcal{P}, m) = P$, then with probability $\geqslant 1 - \delta$ we have that $P \neq \bot$, $\mathrm{Er}(P) \leqslant \mathrm{Er}(P^*) + \Delta + \varepsilon$, and $[\![P]\!](\mathbb{D}) \models \mathit{post}$.*

Extensions of Corollary 4.8 still provide strong results on the convergence of DIGITS: for example, if P^* is not α-robust, but there exists an α-robust P with $P^* \in B_\alpha(P)$, then one can show $\lim_{\alpha \to 0} \Delta = 0$; in this case, running DIGITS for sufficiently large m preserves the desired convergence result from Corollary 4.7.

Second, an optimal P^* that satisfies *post* may not actually exist when *post* consists of open conditions. Consider, for example, if our interval programs from before (Section 4.1, Example 4.1) had a postcondition with a strict inequality $\Pr_{x \sim \mathbb{D}}[P(x) = 1] > 0.5$. Then every interval with a right endpoint at $0.5 + \varepsilon$ satisfies *post*, yet the interval using $0.5 + \frac{\varepsilon}{2}$ would have smaller error; unfortunately, the value 0.5 itself does *not* satisfy *post*. In such cases, we do consider P^* to be the infimum with respect to Er of the set $\{P \in \mathcal{P} \mid [\![P]\!](\mathbb{D}) \models \mathit{post}\}$. But since this P^* does not satisfy *post*, it is trivially not α-robust, and we rely on the result of Corollary 4.8.

The convergence of DIGITS relies on the existence of α-robust programs. Theorem 4.6—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 4.6 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 (Section 4.6) suggests, as we might expect, that this would be a pathological case.

Understanding Convergence

The importance of finite VC dimension is due to the fact that the convergence statement borrows directly from *probably approximately correct learning* (PAC learning). We will briefly discuss a core detail of efficient PAC learning that is relevant to understanding the convergence of DIGITS (and, in turn, our analysis of τ-DIGITS in Section 4.4), and refer the interested reader to Kearns and Vazirani (1994) for a complete overview. Specifically, we consider the notion of an *ε-net*, which establishes the approximate-definability of a target program in terms of points in its input space.

Definition 4.9 (ε-net)**.** *Suppose $P \in \mathcal{P}$ is a target program, and points in its input domain $\mathcal{X}$ are distributed $x \sim \mathbb{D}$. For a fixed $\varepsilon \in [0, 1]$, we say a set of points $S \subset \mathcal{X}$ is an* ε-net *for P (with respect to $\mathcal{P}$ and $\mathbb{D}$) if for every $P' \in \mathcal{P}$ with $d_{\mathbb{D}}(P, P') > \varepsilon$ there exists a witness $x \in S$ such that $P(x) \neq P'(x)$.*

In other words, if S is an ε-net for P, and if P' "agrees" with P on all of S, then P and P' can only differ by at most ε probability mass.

Observe the relevance of ε-nets to the convergence of DIGITS: the synthesis oracle is guaranteed not to "fail" by producing only programs ε-far from some ε-robust P^* if the sample set happens to be an ε-net for P^*. In fact, this observation is exactly the core of the PAC learning argument: having an ε-net exactly guarantees the approximate learnability.

A remarkable result of computational learning theory is that whenever $\mathcal{P}$ has finite VC dimension, the probability that m random samples fail to yield an ε-net becomes diminishingly small as m increases. Indeed, the given VCCOST function used in Theorem 4.6 is a dual form of this latter

result—that polynomially many samples are sufficient to form an ε-net with high probability.

4.3 Efficient Trie-Based Search

After providing details on the search strategy employed by DIGITS, we present our theoretical result on the polynomial bound on the number of synthesis queries that DIGITS requires.

The Trie-Based Search Strategy of DIGITS

Naive DIGITS, as presented in Figure 4.2, performs a very unstructured, exponential search over the output labelings of the sampled inputs—i.e., the possible Boolean functions f in Figure 4.2. We can do better by incrementally exploring the set of possible output labelings using a trie data structure. In this section, we study the complexity of this technique through the lens of computational learning theory and discover the surprising result that DIGITS requires a polynomial number of calls to the synthesizer in the size of the sample set! Our improved search algorithm (Section 4.4) inherits these results.

For the remainder of this chapter, we use DIGITS to refer to this incremental version. A full description is necessary for our analysis: Figure 4.4 (non-framed rules only) consists of a collection of guarded rules describing the construction of the trie used by DIGITS to incrementally explore the set of possible output labelings. Our improved version, τ-DIGITS (presented in Section 4.4), corresponds to the addition of the framed parts, but without them, the rules describe DIGITS.

Nodes in the trie represent partial output labelings—i.e., functions f assigning Boolean values to only some of the samples in $S = \{x_1, \ldots, x_m\}$. Each node is identified by a binary string $\sigma = b_1 \cdots b_k$ (k can be smaller than m) denoting the path to the node from the root. The string σ also

$$\frac{}{\textit{explored} \leftarrow \{\epsilon\} \quad P_\epsilon \leftarrow \hat{P} \quad \textit{depth} \leftarrow 0 \quad \textit{best} \leftarrow \bot} \text{ Initialize}$$

$$\frac{\begin{array}{c}\forall \sigma \in \textit{explored}.\, \forall b \in \{0,1\}. \\ (P_\sigma \neq \bot \wedge |\sigma b| \leqslant \textit{depth} \boxed{\wedge \textit{unblocked}(\sigma b)}) \Rightarrow \sigma b \in \textit{explored}\end{array}}{\textit{sample}_{\textit{depth}+1} \sim \mathbb{D} \quad \textit{depth} \leftarrow \textit{depth} + 1} \text{ Deepen}$$

$$\frac{\begin{array}{c}\sigma \in \textit{explored} \quad P_\sigma \neq \bot \quad b \in \{0,1\} \\ \sigma b \notin \textit{explored} \quad |\sigma b| \leqslant \textit{depth} \quad \boxed{\textit{unblocked}(\sigma b)}\end{array}}{\begin{array}{c}P_{\sigma b} \leftarrow \mathcal{O}_{\text{syn}}(\{(\textit{sample}_{i+1}, \sigma b(i)) : 0 \leqslant i < |\sigma b|\}) \\ \textit{explored} \leftarrow \textit{explored} \cup \{\sigma b\}\end{array}} \text{ Explore (Synthesis Query)}$$

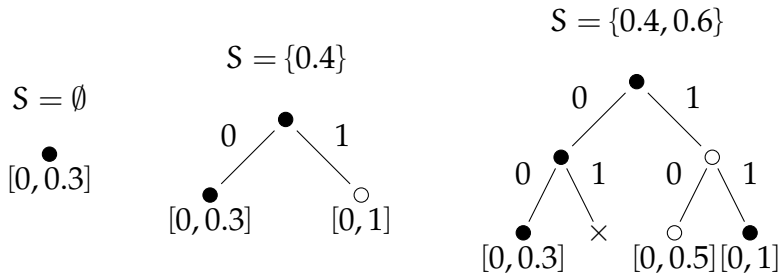
$$\frac{\begin{array}{c}\sigma \in \textit{explored} \quad P_\sigma \neq \bot \quad b \in \{0,1\} \quad \sigma b \notin \textit{explored} \\ |\sigma b| \leqslant \textit{depth} \quad \boxed{\textit{unblocked}(\sigma b)} \quad P_\sigma(\textit{sample}_{|\sigma b|}) = b\end{array}}{P_{\sigma b} \leftarrow P_\sigma \quad \textit{explored} \leftarrow \textit{explored} \cup \{\sigma b\}} \text{ Explore (Soln. Prop.)}$$

$$\frac{\sigma^* = \text{argmin}_\sigma\{\mathcal{O}_{\text{err}}(P_\sigma) \mid \sigma \in \textit{explored} \wedge P_\sigma \neq \bot \wedge \mathcal{O}_{\text{ver}}(P_\sigma) = \text{true}\}}{\textit{best} \leftarrow P_{\sigma^*}} \text{ Best}$$

$$\boxed{\text{where } \textit{unblocked}(\sigma) \coloneqq |\{i : 0 \leqslant i < |\sigma| \wedge \sigma(i) \neq \hat{P}(\textit{sample}_{i+1})\}| \leqslant \tau \cdot \textit{depth}}$$

Figure 4.4: Full DIGITS description and our new extension, τ-DIGITS, shown in boxes.

describes the partial output-labeling function f corresponding to the node—i.e., if the i-th bit b_i is set to 1, then $f(x_i) = \texttt{true}$. The set *explored* represents the nodes in the trie built thus far; for each new node, the algorithm synthesizes a program consistent with the corresponding partial output function ("Explore" rules). The variable *depth* controls the incremental aspect of the search and represents the maximum length of any σ in *explored*; it is incremented whenever all nodes up to that depth have been explored (the "Deepen" rule). The crucial part of the algorithm is that, if no program can be synthesized for the partial output function of a node identified by σ, the algorithm does not need to issue further synthesis queries for the descendants of σ.



Hollow circles denote calls to $\mathcal{O}_{\text{syn}}$ that yield new programs; the cross denotes a call to $\mathcal{O}_{\text{syn}}$ that returns $\perp$.

Figure 4.5: Example execution of incremental DIGITS on interval programs, starting from $[0, 0.3]$

Figure 4.5 shows how DIGITS builds a trie for an example run on the interval programs from Example 4.1, where we suppose we begin with an incorrect program describing the interval $[0, 0.3]$. Initially, we set the root program to $[0, 0.3]$ (left figure). The "Deepen" rule applies, so a sample is added to the set of samples—suppose it's 0.4. "Explore" rules are then applied twice to build the children of the root: the child following the 0 branch needs to map $0.4 \mapsto 0$, which $[0, 0.3]$ already does, thus it is propagated to that child without asking $\mathcal{O}_{\text{syn}}$ to perform a synthesis query. For the child following 1, we instead make a synthesis query, using the oracle $\mathcal{O}_{\text{syn}}$, for any value of a such that $[0, a]$ maps $0.4 \mapsto 1$—suppose it returns the solution $a = 1$, and we associate $[0, 1]$ with this node. At this point we have exhausted depth 1 (middle figure), so "Deepen" once again applies, perhaps adding 0.6 to the sample set. At this depth (right figure), only two calls to $\mathcal{O}_{\text{syn}}$ are made: in the case of the call at $\sigma = 01$, there is no value of a that causes both $0.4 \mapsto 0$ and $0.6 \mapsto 1$, so $\mathcal{O}_{\text{syn}}$ returns $\perp$, and we do not try to explore any children of this node in the future. The algorithm continues in this manner until a stopping condition is reached—e.g., enough samples are enumerated.

Polynomial Bound on the Number of Synthesis Queries

The convergence analysis of DIGITS relies on the finite VC dimension of the program model, but VC dimension itself is just a summary of the *growth function*, a function that describes a notion of complexity of the set of programs in question. We will see that the growth function much more precisely describes the behavior of the trie-based search; we will then use a classic result from computational learning theory to derive better bounds on the performance of the search. We define the growth function below, adapting the presentation from Kearns and Vazirani (1994).

Definition 4.10 (Realizable Dichotomies)**.** *We are given a set $\mathcal{P}$ of programs representing functions from $\mathcal{X} \to \{0,1\}$ and a (finite) set of inputs $S \subset \mathcal{X}$. We call any $f : S \to \{0,1\}$ a* dichotomy *of S; if there exists a program $P \in \mathcal{P}$ that extends f to its full domain $\mathcal{X}$, we call f a* realizable dichotomy *in $\mathcal{P}$. We denote the set of realizable dichotomies as*

$$\Pi_{\mathcal{P}}(S) \coloneqq \{f : S \to \{0,1\} \mid \exists P \in \mathcal{P}. \forall x \in S. P(x) = f(x)\}.$$

Observe that for any (infinite) set $\mathcal{P}$ and any finite set S that $1 \leqslant |\Pi_{\mathcal{P}}(S)| \leqslant 2^{|S|}$. We define the growth function in terms of the realizable dichotomies:

Definition 4.11 (Growth Function)**.** *The* growth function *is the maximal number of realizable dichotomies as a function of the number of samples, denoted*

$$\hat{\Pi}_{\mathcal{P}}(m) \coloneqq \max_{\substack{S \subset \mathcal{X}: \\ |S|=m}} \{|\Pi_{\mathcal{P}}(S)|\}.$$

Observe that $\mathcal{P}$ has VC dimension d if and only if d is the largest integer satisfying $\hat{\Pi}_{\mathcal{P}}(d) = 2^d$ (and infinite VC dimension when $\hat{\Pi}_{\mathcal{P}}(m)$ is identically 2^m)—in fact, VC dimension is often defined using this characterization.

Example 4.12. *Consider the set of intervals of the form* $[0, a]$ *as in Example 4.1 and Figure 4.5. For the set of two points* $S = \{0.4, 0.6\}$*, we have that* $|\Pi_{[0,a]}(S)| = 3$*, since, by example:* $a = 0.5$ *accepts* 0.4 *but not* 0.6*,* $a = 0.3$ *accepts neither, and* $a = 1$ *accepts both, thus these three dichotomies are realizable; however, no interval with* 0 *as a left endpoint can accept* 0.6 *and not* 0.4*, thus this dichotomy is not realizable. In fact, for any (finite) set* $S \subset [0, 1]$*, we have that* $|\Pi_{[0,a]}(S)| = |S| + 1$*; we then have that* $\hat{\Pi}_{[0,a]}(m) = m + 1$.

When DIGITS terminates having used a sample set S, it has considered all the dichotomies of S: the programs it has enumerated exactly correspond to extensions of the realizable dichotomies $\Pi_{\mathcal{P}}(S)$. The trie-based exploration is effectively trying to minimize the number of $\mathcal{O}_{\text{syn}}$ queries performed on non-realizable ones, but doing so without explicit knowledge of the full functional behavior of programs in $\mathcal{P}$. In fact, it manages to stay relatively close to performing queries only on the realizable dichotomies:

Lemma 4.13. DIGITS *performs at most* $|S||\Pi_{\mathcal{P}}(S)|$ *synthesis oracle queries. More precisely, let* $S = \{x_1, \ldots, x_m\}$ *be indexed by the depth at which each sample was added: the exact number of synthesis queries is* $\sum_{\ell=1}^{m} |\Pi_{\mathcal{P}}(\{x_1, \ldots, x_{\ell-1}\})|$.

Proof. Let T_d denote the total number of queries performed once depth d is completed. We perform no queries for the root,[3] thus $T_0 = 0$. Upon completing depth $d-1$, the realizable dichotomies of $\{x_1, \ldots, x_{d-1}\}$ exactly specify the nodes whose children will be explored at depth d. For each such node, one child is skipped due to solution propagation, while an oracle query is performed on the other, thus $T_d = T_{d-1} + |\Pi_{\mathcal{P}}(\{x_1, \ldots, x_{d-1}\})|$. Lastly, $|\Pi_{\mathcal{P}}(S)|$ cannot decrease by adding elements to S, so we have that $T_m = \sum_{\ell=1}^{m} |\Pi_{\mathcal{P}}(\{x_1, \ldots, x_{\ell-1}\})| \leqslant \sum_{\ell=1}^{m} |\Pi_{\mathcal{P}}(S)| \leqslant |S||\Pi_{\mathcal{P}}(S)|$.

Connecting DIGITS to the realizable dichotomies and, in turn, the growth function allows us to employ a remarkable result from computational

[3] We assume the functional specification itself is some $\hat{P} \in \mathcal{P}$ and thus can be used—the alternative is a trivial synthesis query on an empty set of constraints.

learning theory, stating that the growth function for any set exhibits one of two asymptotic behaviors: it is either *identically* 2^m (infinite VC dimension) or dominated by a polynomial! This is commonly called the Sauer-Shelah Lemma (Sauer, 1972; Shelah, 1972):

Lemma 4.14 (Sauer-Shelah)**.** *If $\mathcal{P}$ has finite VC dimension d, then for all $m \geqslant d$, $\hat{\Pi}_{\mathcal{P}}(m) \leqslant \left(\frac{em}{d}\right)^d$; i.e. $\hat{\Pi}_{\mathcal{P}}(m) = O(m^d)$.*

Combining our lemma with this famous one yields a surprising result—that for a fixed set of programs $\mathcal{P}$ with finite VC dimension, the number of oracle queries performed by DIGITS *is guaranteedly polynomial* in the depth of the search, where the degree of the polynomial is determined by the VC dimension:

Theorem 4.15. *If $\mathcal{P}$ has VC dimension d, then* DIGITS *performs $O(m^{d+1})$-many synthesis-oracle queries.*

In short, the reason an execution of DIGITS *seems* to enumerate a sub-exponential number of programs (as a function of the depth of the search) is because it literally must be polynomial. Furthermore, the algorithm performs oracle queries on *nearly* only those polynomially-many realizable dichotomies.

Example 4.16. *A* DIGITS *run on the $[0, a]$ programs as in Figure 4.5 using a sample set of size m will perform $O(m^2)$ oracle queries, since the VC dimension of these intervals is 1. (In fact, every run of the algorithm on these programs will perform exactly $\frac{1}{2}m(m+1)$ many queries.)*

4.4 Property-Directed τ-DIGITS

DIGITS has better convergence guarantees when it operates on larger sets of sampled inputs. In this section, we describe a new optimization of DIGITS that reduces the number of synthesis queries performed by the

algorithm so that it more quickly reaches higher depths in the trie, and thus allows to scale to larger samples sets. This optimized DIGITS, called τ-DIGITS, is shown in Figure 4.4 as the set of all the rules of DIGITS plus the framed elements. The high-level idea is to skip synthesis queries that are (quantifiably) unlikely to result in optimal solutions. For example, if the functional specification $\hat{P}$ maps every sampled input in S to 0, then the synthesis query on the mapping of every element of S to 1 becomes increasingly likely to result in programs that have maximal distance from $\hat{P}$ as the size of S increases; hence the algorithm could probably avoid performing that query. In the following, we make use of the concept of *Hamming distance* between pairs of programs:

Definition 4.17 (Hamming Distance)**.** *For any finite set of inputs* S *and any two programs* P_1, P_2*, we denote* $\text{Hamming}_S(P_1, P_2) \coloneqq |\{x \in S \mid P_1(x) \neq P_2(x)\}|$ *(we will also allow any* $\{0,1\}$*-valued string to be an argument of* Hamming_S*).*

Algorithm Description

Fix the given functional specification $\hat{P}$ and suppose that there exists an ε-robust solution P^* with (nearly) minimal error $k = \text{Er}(P^*)$. We would be happy to find *any* program P in P^*'s ε-ball. Suppose we angelically know k a priori, and we thus restrict our search (for each depth m) only to constraint strings (i.e. σ in Figure 4.4) that have Hamming distance not much larger than km.

To be specific, we first fix some threshold $\tau \in (k, 1]$. Intuitively, the optimization corresponds to modifying DIGITS to consider only paths σ through the trie such that $\text{Hamming}_S(\hat{P}, \sigma) \leqslant \tau|S|$. This is performed using the *unblocked* function in Figure 4.4. Since we are ignoring certain paths through the trie, we need to ask: *How much does this decrease the probability of the algorithm succeeding?*—It depends on the tightness of the threshold, which we will address next. Then, we will discuss how to

adaptively modify the threshold τ as τ-DIGITS is executing, which is useful when a good τ is unknown a priori.

Analyzing Failure Probability with Thresholding

Using τ-DIGITS, the choice of τ will affect both (*i*) how many synthesis queries are performed, and (*ii*) the likelihood that we *miss* optimal solutions; in this section we explore the latter point.[4] Interestingly, we will see that all of the analysis is dependent only on parameters directly related to the threshold; notably, none of this analysis is dependent on the complexity of $\mathcal{P}$ (i.e. its VC dimension).

If we really want to learn (something close to) a program P^*, then we should use a value of the threshold τ such that $\Pr_{S\sim\mathbb{D}^m}[\text{Hamming}_S(\hat{P}, P^*) \leqslant \tau m]$ is large—to do so requires knowledge of the distribution of $\text{Hamming}_S(\hat{P}, P^*)$. Recall the *binomial distribution*: for parameters (n, p), it describes the number of successes in n-many trials of an experiment that has success probability p.

Claim 4.18. *Fix* P *and let* $k = \text{Er}(P) = \Pr_{x\sim\mathbb{D}}[\hat{P}(x) \neq P(x)]$. *If* $S \sim \mathbb{D}^m$, *then* $\text{Hamming}_S(\hat{P}, P)$ *is binomially distributed with parameters* (m, k).

Next, we will use our knowledge of this distribution to reason about the *failure probability*, i.e. that τ-DIGITS does not preserve the convergence result of DIGITS.

The simplest argument we can make is a union-bound argument: the thresholded algorithm can "fail" by (*i*) failing to sample an ε-net (Section 4.2, Definition 4.9), or otherwise (*ii*) sampling a set on which the optimal solution has a Hamming distance that is not representative of its actual distance. We provide the quantification of this failure probability in the following theorem:

[4] The former point is a difficult combinatorial question that to our knowledge has no precedent in the computational learning literature, and so we leave it as future work.

Theorem 4.19. *Let* P^* *be a target* ε*-robust program with* $k = \mathrm{Er}(P^*)$*, and let* δ *be the probability that* m *samples do not form an* ε*-net for* P^**. If we run* τ-DIGITS *with* $\tau \in (k, 1]$*, then the failure probability is at most* $\delta + \Pr[X > \tau m]$ *where* $X \sim \mathrm{Binomial}(m, k)$.

In other words, we can use tail probabilities of the binomial distribution to bound the probability that the threshold causes us to "miss" a desirable program we otherwise would have enumerated. Explicitly, we have the following corollary:

Corollary 4.20. τ-DIGITS *increases failure probability (relative to* DIGITS*) by at most* $\Pr[X > \tau m] = \sum_{i=\lfloor \tau m \rfloor + 1}^{m} \binom{m}{i} k^i (1-k)^{m-i}$.

Informally, when m is *not too small*, k is *not too large*, and τ is *reasonably forgiving*, these tail probabilities can be quite small. We can even analyze the asymptotic behavior by using any existing upper bounds on the binomial distribution's tail probabilities—importantly, the additional error diminishes exponentially as m increases, dependent on the size of τ relative to k.

Corollary 4.21. τ-DIGITS *increases failure probability by at most* $e^{-2m(\tau-k)^2}$.[5]

Example 4.22. *Suppose* $m = 100$, $k = 0.1$, *and* $\tau = 0.2$. *Then the extra failure probability term in Theorem 4.19 is less than* 0.001.

As stated at the beginning of this subsection, the balancing act is to choose τ (*i*) small enough so that the algorithm is still fast for large m, yet (*ii*) large enough so that the algorithm is still likely to learn the desired programs. The further challenge is to relax our initial strong assumption that we know the optimal k a priori when determining τ, which we address in the following subsection.

[5] A more precise (though less convenient) bound is $e^{-m(\tau \ln \frac{\tau}{k} + (1-\tau) \ln \frac{1-\tau}{1-k})}$.

Adaptive Threshold

Of course, we do not have the angelic knowledge that lets us pick an ideal threshold τ; the only absolutely sound choice we can make is the trivial $\tau = 1$. Fortunately, we can begin with this choice of τ and *adaptively* refine it as the search progresses. Specifically, every time we encounter a correct program P such that $k = \mathrm{Er}(P)$, we can refine τ to reflect our newfound knowledge that "the best solution has distance of at most k."

We refer to this refinement as *adaptive* τ-DIGITS. The modification involves the addition of the following rule to Figure 4.4:

$$\frac{\mathit{best} \neq \bot}{\tau \leftarrow g(\mathcal{O}_{\mathrm{err}}(\mathit{best}))} \text{ Refine Threshold (for some } g : [0,1] \to [0,1])$$

We can use any (non-decreasing) function g to update the threshold $\tau \leftarrow g(k)$. The simplest choice would be the identity function (which we use in our experiments), although one could use a looser function so as not to over-prune the search. If we choose functions of the form $g(k) = k + b$, then Corollary 4.21 allows us to make (slightly weak) claims of the following form:

Claim 4.23. *Suppose the adaptive algorithm completes a search of up to depth m yielding a best solution with error k (so we have the final threshold value $\tau = k + b$). Suppose also that P^* is an optimal ε-robust program at distance $k - \eta$. The optimization-added failure probability (as in Corollary 4.20) for a run of (non-adaptive) τ-DIGITS completing depth m and using this τ is at most $e^{-2m(b+\eta)^2}$.*

4.5 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 mechanism for specifying

sets of programs $\mathcal{P}$, a procedure for $\mathcal{O}_{\text{syn}}$ that produces programs in $\mathcal{P}$ consistent with labeled examples, and a probabilistic inference algorithm to (*i*) check whether the synthesized program respects the postcondition ($\mathcal{O}_{\text{ver}}$), (*ii*) compute the semantic difference $d_{\mathbb{D}}$ between the synthesized program P and the functional specification $\hat{P}$ ($\mathcal{O}_{\text{err}}$). In this section, we describe the concrete choices of these components for our implementation.

Real-Paramterized Program Sketches. Since we are mostly interested in repairing machine-learned classifiers, a natural design for sets of programs is to allow modifications to real-valued constants appearing in some initial program. These constants are essentially the *weights* of the classifier.

Formally, let P be a 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 candidate programs is defined as

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

We only consider programs containing linear real arithmetic expressions. As such, our set of programs can always 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 (Sharma et al., 2014), and therefore, so will any sketch in our language.

The implementation of $\mathcal{O}_{\text{syn}}(f)$ (for some $f : S \to \{0, 1\}$) follows a sketch-like approach (Solar-Lezama, 2008), where we encode the program and the samples as a formula whose solution instantiates the parameters of $\mathcal{P}$. Let P be a 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 $P_{\mathcal{P}} = P[c_1/h_1, \ldots, c_n/h_n]$, 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 $P_{\mathcal{P}}$ as

a formula as follows, using the function PVC. To simplify the encoding, and without loss of generality, we assume that $P_{\mathcal{P}}$ is in *static single assignment* (SSA) form.

$$\begin{aligned}
\textsc{pvc}(v \leftarrow E) &\triangleq v = [\![E]\!] \\
\textsc{pvc}(P_1\ P_2) &\triangleq \textsc{pvc}(P_1) \wedge \textsc{pvc}(P_2) \\
\textsc{pvc}(\text{if } B \text{ then } P_1 \text{ else } P_2) &\triangleq ([\![B]\!] \Rightarrow \textsc{pvc}(P_1)) \wedge (\neg[\![B]\!] \Rightarrow \textsc{pvc}(P_2))
\end{aligned}$$

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

Once we have encoded the program $P_{\mathcal{P}}$ as a formula φ, for each sample $x_i \in S$, we will construct the formula

$$\varphi_i \triangleq \exists V.\, \varphi[\boldsymbol{v}_I/x_i] \wedge v_r = f(x_i)$$

where V is the set of variables of P, which do not include the introduced holes $h_1, \ldots, h_n$. 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 $\mathcal{P}$ that correctly labels the positive and negative examples specified by f and S. Specifically, $\mathcal{O}_{\text{syn}}(f)$ finds a model $m \models \bigwedge_i \varphi_i$ and returns the program $P_{\mathcal{P}}[h_1/m(h_1), \ldots, h_n/m(h_n)]$, where $m(h_i)$ is the value of h_i in the model m. If $\bigwedge_i \varphi_i$ is unsatisfiable, then $\mathcal{O}_{\text{syn}}$ returns $\perp$.

Theorem 4.24 (Soundness and completeness of $\mathcal{O}_{\text{syn}}$)**.** *Suppose we are given a set of programs $\mathcal{P}$ represented by the program sketch $P_{\mathcal{P}}$ as defined above, along with a labeling of samples $f : S \to \{0, 1\}$. Then, if $\mathcal{O}_{\text{syn}}(f)$ returns a program P, P must appear in $\mathcal{P}$ and classifies S exactly as f does. Otherwise, there is no program in $\mathcal{P}$ consistent with f on S.*

Conflict-driven pruning. We can also learn from the instances in which $\mathcal{O}_{\text{syn}}$ returns $\perp$. In particular, when a failure occurs, we can identify a

subset of the labelings that caused the failure and use it to reduce the set of nodes in the trie for which we explicitly perform the SMT query within $\mathcal{O}_{\text{syn}}$. In our implementation, we will use the *unsatisfiable cores* produced by the SMT solver to compute the subsets of the samples that induce failures and elide any future calls to $\mathcal{O}_{\text{syn}}$ on supersets of these cores.

Probabilistic inference. In our implementation, this component can be instantiated with any probabilistic inference tool—e.g., PSI (Gehr et al., 2016). We use FairSquare; 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.

4.6 Evaluation

In this section, we explore the ability of DIGITS to repair the unfair benchmarks from Chapter 3; we then compare its performance to τ-DIGITS to conclude that our improvements do improve the quality of solutions and guarantees. Finally, we apply τ-DIGITS on a few other problems to test its synthesis capabilities.

Benchmarks. We used an online dataset (UCI, 1996) 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 (Hall et al., 2009) until we obtained 3 SVMs and

3 DTs that did *not* satisfy a probabilistic postcondition describing group fairness.[6] In particular, we used the following postcondition:

$$\frac{\Pr[\text{high income} \mid \text{female}]}{\Pr[\text{high income} \mid \text{male}]} \geqslant 0.85$$

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 DIGITS. Table 4.1 details the performance of DIGITS on our suite of fairness benchmarks that were given 600 seconds to perform repair. For example, on the DT labeled DT_{16}, the table shows that within the best-effort period, DIGITS was able to enumerate all possible labelings for a set of 50 samples (the depth of the trie), and found a solution P satisfying the postcondition that differs from the original program with probability $\text{Er}(\mathsf{P}) = 0.098$. Despite the fact that there are $2^{50} \approx 10^{15}$ such labelings, only 1,903 were realizable by a program in $\mathcal{P}$. In fact, during the construction of the trie, DIGITS needed only to invoke calls to $\mathcal{O}_{\text{syn}}$ for 26,628 (2,967 + 23,661) of them: among these, 2,967 resulted in SMT queries to construct a consistent program in $\mathcal{P}$, while 23,661 (88%) potential queries that would have been unsatisfiable and return $\perp$ were avoided using conflict-driven pruning.

It is visibly apparent that the trie structure and conflict-driven pruning save many synthesis and verification queries. However, savings from

[6] These benchmarks did originate from the unfair verification instances in FairSquare's evaluation, Section 3.4.

Name	Holes	Best Error	$\lvert S \rvert$	# Satisfiable	# Queries	# Unsat Pruned
DT_{16}	5	0.098	50	1,903	2,967	23,661
DT_{44}	3	0.030	91	1,741	2,159	31,867
DT_{4}	2	0.140	79	1,746	2,453	44,914
SVM_3	4	0.067	16	1,000	1,338	2,170
	3	0.062	23	1,600	1,988	7,001
	2	0.046	91	1,980	2,227	59,741
	1	0.060	661	662	1,967	216,825
SVM_4	5	0.197	10	508	608	154
	4	0.204	13	1,018	1,223	1,077
	3	0.195	22	1,424	1,830	5,613
	2	0.040	83	1,674	1,891	44,357
	1	0.044	628	629	1,857	195,650
SVM_5	6	0.131	8	240	251	3
	5	0.122	10	632	696	151
	4	0.103	13	1,018	1,212	1,082
	3	0.096	23	1,348	1,680	6,001
	2	0.056	88	1,736	1,960	49,865
	1	0.067	598	599	1,775	177,327

Table 4.1: Results of DIGITS on fairness benchmarks (600s time limit).

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 set of programs. Therefore, we expect the instances with a more expressive set of programs to perform worse.

Accordingly, Table 4.1 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 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 represent a superset of programs and thus have the potential to contain solutions with better error $\mathrm{Er}(\cdot)$, 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 error minimality because the trie is explored deeper. This trend continues until the only hole that remains is the constant offset, when the set of programs is no longer expressive enough to capture solutions with such a minimal difference.

Effectiveness of τ-DIGITS

Next we evaluate our new algorithm τ-DIGITS (Figure 4.4) and its adaptive variant (Section 4.4) against DIGITS (i.e., τ-DIGITS with $\tau = 1$). Our evaluation aims to answer the following questions:

RQ1 Is adaptive τ-DIGITS more effective/precise than τ-DIGITS?

RQ2 Is τ-DIGITS more effective/precise than DIGITS?

RQ3 Can τ-DIGITS solve challenging synthesis problems?

We experiment on three sets of benchmarks: (*i*) synthetic examples for which the optimal solutions can be computed analytically, (*ii*) the aforementioned fairness repair benchmarks, (*iii*) a variant of the thermostat-controller synthesis problem presented in Chaudhuri et al. (2014).

Synthetic Benchmarks

We consider a class of synthetic programs for which we can compute the optimal solution exactly; this lets us compare the results of our implementation to an ideal baseline. Here, the program model $\mathcal{P}$ is defined as

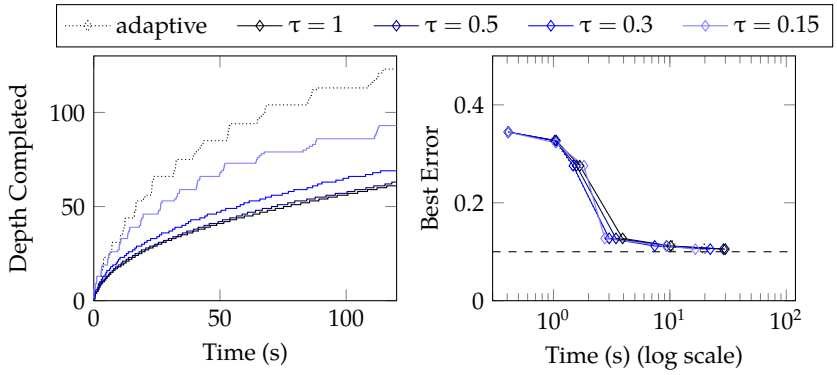


Figure 4.6: Synthetic hyperrectangle problem instance with parameters $d = 1$, $b = 0.1$

the set of axis-aligned hyperrectangles within $[-1, 1]^d$ ($d \in \{1, 2, 3\}$ and the VC dimension is $2d$), and the input distribution $\mathbb{D}$ is such that inputs are distributed uniformly over $[-1, 1]^d$. We fix some probability mass $b \in \{0.05, 0.1, 0.2\}$ and define the benchmarks so that the best error for a correct solution is exactly b (see Appendix C.2).

We run our implementation using thresholds $\tau \in \{0.07, 0.15, 0.3, 0.5, 1\}$, omitting those values for which $\tau < b$; additionally, we also consider an adaptive run where τ is initialized as the value 1, and whenever a new best solution is enumerated with error k, we update $\tau \leftarrow k$. Each combination of parameters was run for a period of 2 minutes. Figure 4.6 fixates on $d = 1$, $b = 0.1$ and shows each of the following as a function of time: (*i*) the depth completed by the search (i.e. the current size of the sample set), and (*ii*) the best solution found by the search. (See Appendix C.2 for other configurations of (d, b).)

By studying Figure 4.6 we see that the adaptive threshold search performs at least as well as the tight thresholds fixed a priori because reasonable solutions are found early. In fact, all search configurations find solutions very close to the optimal error (indicated by the horizontal dashed line). Regardless, they reach different depths, and *the main advantage of*

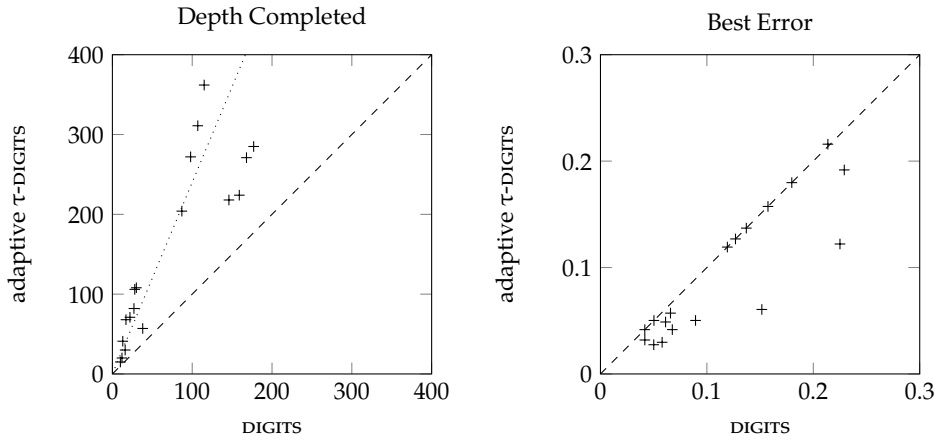


Figure 4.7: Improvement of using adaptive τ-DIGITS on the fairness benchmarks. Left: the dotted line marks the 2.4× average increase in depth.

reaching large depths concerns the strength of the optimality guarantee. Note, also, that small τ values are necessary to see improvements in the completed depth of the search. Indeed, the discrepancy between the depth-versus-time functions diminishes drastically for the problem instances with larger values of b (see Appendix C.2); the gains of the optimization are contingent on the existence of correct solutions *close* to the functional specification.

Findings (RQ1): τ-DIGITS *does* tend to find *reasonable* solutions at early depths and near-optimal solutions at later depths, thus adaptive τ-DIGITS is more effective than τ-DIGITS, and we use it throughout our remaining experiments.

Fairness Benchmarks

For each repair problem, we run both DIGITS and adaptive τ-DIGITS (again, with initial $\tau = 1$ and the identity refinement function). Each benchmark is run for 10 minutes, where the same sample set is used for both algorithms.

Figure 4.7 shows, for each benchmark, (*i*) the largest sample set size

completed by adaptive τ-DIGITS versus DIGITS (left—above the diagonal line indicates adaptive τ-DIGITS reaches further depths), and (*ii*) the error of the best solution found by adaptive τ-DIGITS versus DIGITS (right—below the diagonal line indicates adaptive τ-DIGITS finds better solutions). We see that adaptive τ-DIGITS reaches further depths on every problem instance, many of which are substantial improvements, and that it finds better solutions on 10 of the 18 problems. For those which did not improve, either the search was already deep enough that DIGITS was able to find near-optimal solutions, or the complexity of the synthesis queries is such that the search is still constrained to small depths.

Findings (RQ2): Adaptive τ-DIGITS can find better solutions than those found by DIGITS and can reach greater search depths.

Thermostat Controller

We challenge adaptive τ-DIGITS with the task of synthesizing a thermostat controller, borrowing the benchmark from Chaudhuri et al. (2014). The input to the controller is the initial temperature of the environment; since the world is uncertain, there is a specified probability distribution over the temperatures. The controller itself is a program sketch consisting primarily of a single main loop: iterations of the loop correspond to timesteps, during which the synthesized parameters dictate an incremental update made by the thermostat based on the current temperature. The loop runs for 40 iterations, then terminates, returning the absolute value of the difference between its final actual temperature and the target temperature.

The postcondition is a Boolean probabilistic correctness property intuitively corresponding to controller safety, e.g. with high probability, the temperature should never exceed certain thresholds. In Chaudhuri et al. (2014), there is a quantitative objective in the form of minimizing the expected value $\mathrm{E}[|\mathit{actual} - \mathit{target}|]$—our setting does not admit optimizing with respect to expectations, so we must modify the problem. Instead,

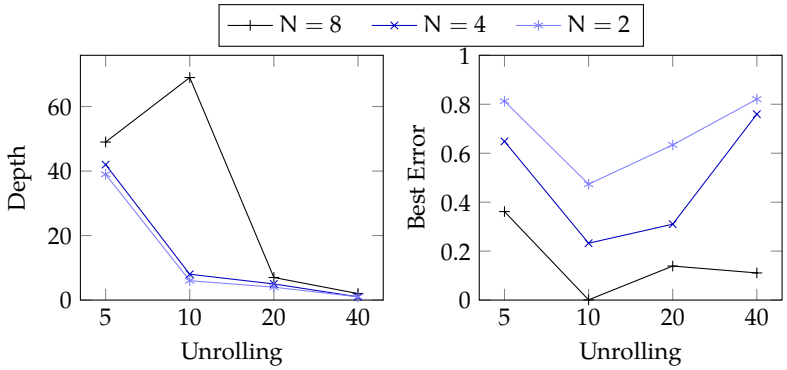


Figure 4.8: Thermostat controller results

we fix some value N ($N \in \{2, 4, 8\}$) and have the program return 0 when $|\mathit{actual} - \mathit{target}| < N$ and 1 otherwise. Our quantitative objective is to minimize the error from the constant-zero functional specification $\hat{P}(x) \coloneqq 0$ (i.e. the actual temperature always gets close enough to the target).

We consider variants of the program where the thermostat runs for fewer timesteps and try loop unrollings of size $\{5, 10, 20, 40\}$. We run each benchmark for 10 minutes: the final completed search depths and best error of solutions are shown in Figure 4.8. For this particular experiment, we use the SMT solver CVC4 (Barrett et al., 2011) because it performs better than Z3 on the occurring SMT instances.

As we would expect, for larger values of N it is "easier" for the thermostat to reach the target temperature threshold and thus the quality of the best solution increases in N. However, with small unrollings (i.e. 5) the synthesized controllers do not have enough iterations (time) to modify the temperature enough for the probability mass of extremal temperatures to reach the target: as we increase the number of unrollings to 10, we see that better solutions can be found since the set of programs are capable of stronger behavior.

On the other hand, the completed depth of the search plummets as the unrolling increases due to the complexity of the $\mathcal{O}_{\text{syn}}$ queries. Con-

sequently, for 20 and 40 unrollings, adaptive τ-DIGITS synthesizes worse solutions because it cannot reach the necessary depths to obtain better guarantees.

One final point of note is that for $N = 8$ and 10 unrollings, it seems that there is a sharp spike in the completed depth. However, this is somewhat artificial: because $N = 8$ creates a very lenient quantitative objective, an early $\mathcal{O}_{\text{syn}}$ query happens to yield a program with an error less than 10^{-3}. Adaptive τ-DIGITS then updates $\tau \leftarrow\approx 10^{-3}$ and skips most synthesis queries.

Findings (RQ3): Adaptive τ-DIGITS can synthesize small variants of a complex thermostat controller, but cannot solve variants with many loop iterations.

4.7 Related Work

Over the past few years, progress in automatic program synthesis has touched many application domains, including automating data wrangling and data extraction tasks (Polozov and Gulwani, 2015; Raza and Gulwani, 2017; Wang et al., 2016; Gulwani, 2016; Barowy et al., 2015; Gulwani, 2011), generating network configurations that meet user intents (Subramanian et al., 2017; El-Hassany et al., 2017), optimizing low-level code (Srinivasan and Reps, 2015; Schkufza et al., 2016), and more (Bastani et al., 2017; Gulwani, 2014).

Program repair and synthesis. Automated repair has been studied in the non-probabilistic setting (Könighofer and Bloem, 2011; Mechtaev et al., 2015; D'Antoni et al., 2016; Von Essen and Jobstmann, 2015; Jobstmann et al., 2005). 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 (D'Antoni et al., 2016). 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 (Solar-Lezama, 2008) 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., Bartocci et al. (2011); Chen et al. (2013). 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* (Chaudhuri et al., 2014) 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. Smoothed proof search can minimize expectation; τ-DIGITS minimizes probability only. However, unlike τ-DIGITS, smoothed proof search lacks formal convergence guarantees and cannot support the rich probabilistic postconditions we support, e.g., as in the fairness benchmarks.

Works on synthesis of probabilistic programs are aimed at a different problem (Nori et al., 2015; Chasins and Phothilimthana, 2017; Saad et al., 2019): that of synthesizing a generative model of data. For example, Nori et al. (2015) use sketches of probabilistic programs and complete them with

a stochastic search. Recently, Saad et al. (2019) synthesize an ensemble of probabilistic programs for learning Gaussian processes and other models.

Kučera et al. (2017) present a technique for automatically synthesizing program transformations that introduce uncertainty into a given program with the goal of satisfying given privacy policies—e.g., preventing information leaks. They leverage the specific structure of their problem to reduce it to an SMT constraint solving problem. The problem tackled in Kučera et al. (2017) is orthogonal to the one targeted in this chapter and the techniques are therefore very different.

Stochastic satisfiability. Our problem is closely related to, and subsumes, the problem of E-MAJSAT (Littman et al., 1998), a special case of *stochastic satisfiability* (SSAT) (Papadimitriou, 1985) 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 (Dwork et al., 2012; Friedler et al., 2016; Hardt et al., 2016; Feldman et al., 2015; Datta et al., 2016). Approaches to enforcing fairness in machine-learned classifiers include altering the data to remove correlations with protected attributes (Feldman et al., 2015) and imposing a fairness definition as a requirement of the learning algorithm (Hardt et al., 2016). However, the general problem presented in this chapter of modifying an existing program (be it learned or manually constructed) to meet a quantitative probabilistic property is novel.

5 CONCLUSION

Thus ends this dissertation's cursory voyage into providing formal guarantees within automated decision-making. To recapitulate, we first presented Antidote, a tool to verify data-poisoning robustness in decision-tree learning, which served as an example of how we might formally reason about the relation between changes in a training set and changes in model behavior. Second, we presented FairSquare and DIGITS, a pair of techniques to certify and enforce probabilistic properties of decision-making programs, such as algorithmic fairness.

5.1 Future Work

Beyond Antidote. In our development of Antidote, we considered two simple forms of abstract domains; these were initial ideas on how to represent abstractions of large collections of training sets, but there is a lot of room to explore other abstract domains for programs that manipulate large data tensors. One immediate direction for future work is to look for a version of our disjunctive domain that performs partial joins on similar disjuncts in attempt to moderate the resource-precision tradeoff.

Our analysis was performed for a very specific formalization of decision-tree learning. Ideally, a practical tool would support a more expressive set of learner behaviors, e.g. pruning. (The simplest way to soundly handle pruning would be to take the join of the possible outcomes if the learner could stop recursing at each depth; being more precise would require a proper abstraction over trees.) A natural extension would be to lift our techniques from individual trees to decision-forest learning. The key challenge here is assessing how best to handle the non-determinism present in such ensemble methods while still providing a guarantee about all possible executions of the learner. Moving beyond trees and forests,

it would be exciting to see abstract transformers devised to prove the poisoning-robustness of gradient descent.

Finally, devising the abstract transformers required careful, manual effort to maintain precision, and it's possible that slightly different expressions would have been more amenable to verification. It would be interesting to see, in general, how insights into developing sound, precise abstract transformers could be applied for the purpose of designing more robust learning algorithms. On a related note, Antidote focused on *verifying* data-poisoning robustness; what ought to be the corresponding synthesis problem? Can we synthesize, for example, greedy predicate selection expressions that result in a provably robust learner? What about other properties we would like the learner or model to posses by construction?

Extending DIGITS. One potentially unsatisfying aspect of FairSquare is that its proofs of unfairness take the form of a large collection of hyperrectangles whose weighted volumes witness that the postcondition does not hold: this is not easily interpretable by humans, and does not give us intuition into what was *wrong*. (Similarly, DIGITS repairs unfairness while completely agnostic to this information.) Is there some kind of explanation we could synthesize from this artifact? In traditional verification, a counterexample is a clear artifact that falsifies a postcondition. In the probabilistic setting, however, there is no single execution trace that explains why a postcondition does not hold. Exploring debugging in the probabilistic setting is an interesting problem for future work.

An immediate direction for future work is to extend the expressivity of DIGITS. In particular, FairSquare is applicable for the probabilistic verification of (loop-free, etc.) programs representing functions mapping $\mathcal{X} \to \mathcal{Y}$ for (largely) arbitrary choices of $\mathcal{X}$ and $\mathcal{Y}$; in its current form, DIGITS can only handle the case of $\mathcal{X} \to \{0, 1\}$. For learning multiclass (N-categorical) functions, implementation is as simple as adapting the binary-trie search

to N-ary, and for understanding convergence guarantees we look to the *Natarajan Dimension* (Natarajan, 1989), which exactly generalizes VC Dimension to the multiclass setting. (It is unclear what the ramifications will be on the complexity of the algorithm.) The further step of learning real-valued outputs could be done by some process of iteratively refining a partition over the reals and otherwise reducing to the categorical case; we can look to the computational learning theory work on the learnability of real-valued functions for inspiration (Bartlett et al., 1994).

The ultimate goal is for FairSquare and DIGITS to be a relatively complete pair of procedures for probabilistic program analysis problems. Further stretch goals include extending them beyond the current limitations of their restricted grammars and reliance on SMT. For example, to handle loops efficiently, there is room to explore the applicability of loop summarization techniques; simpler SMT queries could likely be obtained through appropriate uses of program slicing; SMT itself may not be necessary in all cases, particularly as the synthesis oracle $\mathcal{O}_{\text{syn}}$ in DIGITS, where any programming-by-examples framework would be applicable. Furthermore, could we use unsound or incomplete, faster synthesis oracles to make DIGITS even more practical?—Could we still have relativized, weaker convergence guarantees? More generally, can we combine the aspects of machine learning that excel at fitting to large quantities of data with the aspects of program synthesis that give us formal guarantees, achieving the best of both worlds? Such explorations would build towards bridging the gap between (*i*) what these tools *can* do in their current state, and (*ii*) what these tools *need to be able to do* so that problems such as the certification and enforcement of algorithmic fairness at scale can become a reality.

5.2 Concluding Remarks

Computing and automated reasoning are already impacting society in *profound* ways, a trend that is sure to continue in the years to come. The need for *societally responsible* automated reasoning will only grow more paramount as this frontier advances. I can only hope that the ideas in this thesis contribute to satisfying this necessity. (Perhaps a modest goal is for the machine learning suites used by practitioners of the future to be equipped with formal guarantees, e.g. learners produce not just a model, but also some certificate of fairness alongside it.) All of us who work as STEM researchers should not shy from the potentially uncomfortable intersection of our technologies with the Humanities; we must take an interdisceplanary perspective when shaping the world of tomorrow.

REFERENCES

Ajunwa, Ifeoma, Sorelle Friedler, Carlos E Scheidegger, and Suresh Venkatasubramanian. 2016. Hiring by algorithm: predicting and preventing disparate impact. *Available at SSRN 2746078*.

Albarghouthi, Aws, Loris D'Antoni, and Samuel Drews. 2017a. Repairing decision-making programs under uncertainty. In *Computer aided verification*, ed. Rupak Majumdar and Viktor Kunčak, 181–200. Cham: Springer International Publishing.

Albarghouthi, Aws, Loris D'Antoni, Samuel Drews, and Aditya V. Nori. 2017b. FairSquare: probabilistic verification of program fairness. *Proceedings of the ACM on Programming Languages* 1(OOPSLA):1–30.

Albarghouthi, Aws, and Samuel Vinitsky. 2019. Fairness-aware programming. In *Proceedings of the conference on fairness, accountability, and transparency, fat* 2019, atlanta, ga, usa, january 29-31, 2019*, 211–219. ACM.

Alfeld, Scott, Xiaojin Zhu, and Paul Barford. 2016. Data poisoning attacks against autoregressive models. In *Thirtieth aaai conference on artificial intelligence*.

Anderson, Greg, Shankara Pailoor, Isil Dillig, and Swarat Chaudhuri. 2019. Optimization and abstraction: a synergistic approach for analyzing neural network robustness. In *Proceedings of the 40th acm sigplan conference on programming language design and implementation*, 731–744. ACM.

Angwin, Julia, Jeff Larson, Surya Mattu, and Lauren Kirchner. 2016. Machine bias: There's software used across the country to predict future criminals. and it's biased against blacks. https://www.propublica.org/article/machine-bias-risk-assessments-in-criminal-sentencing. (Accessed on 06/18/2016).

Asarin, Eugene, Olivier Bournez, Thao Dang, and Oded Maler. 2000. Approximate reachability analysis of piecewise-linear dynamical systems. In *International workshop on hybrid systems: Computation and control*, 20–31. Springer.

Barnett, Mike, and K Rustan M Leino. 2005. Weakest-precondition of unstructured programs. In *Acm sigsoft software engineering notes*, vol. 31, 82–87. ACM.

Barocas, Solon, and Andrew D Selbst. 2014. Big data's disparate impact. *Available at SSRN 2477899*.

Barowy, Daniel W., Sumit Gulwani, Ted Hart, and Benjamin G. Zorn. 2015. Flashrelate: extracting relational data from semi-structured spreadsheets using examples. In *Proceedings of the 36th ACM SIGPLAN conference on programming language design and implementation, portland, or, usa, june 15-17, 2015*, 218–228.

Barrett, Clark, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. Cvc4. In *Proceedings of the 23rd international conference on computer aided verification*, 171–177. CAV'11, Berlin, Heidelberg: Springer-Verlag.

Barthe, Gilles, Pedro R. D'Argenio, and Tamara Rezk. 2004. Secure information flow by self-composition. In *Csfw*.

Barthe, Gilles, Marco Gaboardi, Emilio Jesús Gallego Arias, Justin Hsu, César Kunz, and Pierre-Yves Strub. 2014. Proving differential privacy in hoare logic. In *IEEE 27th computer security foundations symposium, CSF 2014, vienna, austria, 19-22 july, 2014*, 411–424.

Bartlett, Peter L., Philip M. Long, and Robert C. Williamson. 1994. Fat-shattering and the learnability of real-valued functions. In *Proceedings*

of the seventh annual conference on computational learning theory - COLT '94. ACM Press.

Bartocci, Ezio, Radu Grosu, Panagiotis Katsaros, C. R. Ramakrishnan, and Scott A. Smolka. 2011. *Model repair for probabilistic systems*, 326–340. Berlin, Heidelberg: Springer Berlin Heidelberg.

Bastani, Osbert, Yani Ioannou, Leonidas Lampropoulos, Dimitrios Vytiniotis, Aditya Nori, and Antonio Criminisi. 2016. Measuring neural net robustness with constraints. *CoRR* abs/1605.07262.

Bastani, Osbert, Rahul Sharma, Alex Aiken, and Percy Liang. 2017. Synthesizing program input grammars. In *Proceedings of the 38th ACM SIGPLAN conference on programming language design and implementation, PLDI 2017, barcelona, spain, june 18-23, 2017*, 95–110.

Basu, Saugata, Richard Pollack, and Marie-Françoise Roy. 2006. *Algorithms in real algebraic geometry (algorithms and computation in mathematics)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc.

BBC. 2015. Google apologises for Photos app's racist blunder. *BBC News*.

Belle, Vaishak, Guy Van den Broeck, and Andrea Passerini. 2015a. Hashing-based approximate probabilistic inference in hybrid domains. In *Proceedings of the 31st conference on uncertainty in artificial intelligence (uai)*.

Belle, Vaishak, Guy Van den Broeck, and Andrea Passerini. 2016. Component caching in hybrid domains with piecewise polynomial densities. In *Proceedings of the thirtieth AAAI conference on artificial intelligence, february 12-17, 2016, phoenix, arizona, USA.*, 3369–3375.

Belle, Vaishak, Andrea Passerini, and Guy Van den Broeck. 2015b. Probabilistic inference in hybrid domains by weighted model integration.

In *Proceedings of the twenty-fourth international joint conference on artificial intelligence, IJCAI 2015, buenos aires, argentina, july 25-31, 2015*, 2770–2776.

Berg, Nate. 2014. Predicting crime, lapd-style. https://www.theguardian.com/cities/2014/jun/25/predicting-crime-lapd-los-angeles-police-data-analysis-algorithm-minority-report. (Accessed on 06/18/2016).

Biggio, Battista, Blaine Nelson, and Pavel Laskov. 2012. Poisoning attacks against support vector machines. In *Proceedings of the 29th international coference on international conference on machine learning*, 1467–1474. ICML'12, USA: Omnipress.

Bishop, Christopher M. 2006. Pattern recognition. *Machine Learning* 128.

Blumer, Anselm, Andrzej Ehrenfeucht, David Haussler, and Manfred K Warmuth. 1989. Learnability and the vapnik-chervonenkis dimension. *Journal of the ACM (JACM)* 36(4):929–965.

Bournez, Olivier, Oded Maler, and Amir Pnueli. 1999. *Orthogonal polyhedra: Representation and computation*, 46–60. Berlin, Heidelberg: Springer Berlin Heidelberg.

Breiman, Leo. 2017. *Classification and regression trees*. Routledge.

Brown, Christopher W. 2003. Qepcad b: A program for computing with semi-algebraic sets using cads. *SIGSAM Bull.* 37(4):97–108.

Calders, Toon, and Sicco Verwer. 2010. Three naive bayes approaches for discrimination-free classification. *Data Mining and Knowledge Discovery* 21(2):277–292.

Carbin, Michael, Sasa Misailovic, and Martin C. Rinard. 2013. Verifying quantitative reliability for programs that execute on unreliable hardware. In *Proceedings of the 2013 ACM SIGPLAN international conference on object*

oriented programming systems languages & applications, OOPSLA 2013, part of SPLASH 2013, indianapolis, in, usa, october 26-31, 2013, 33–52.

Carlini, Nicholas, and David Wagner. 2017. Towards evaluating the robustness of neural networks. In *2017 ieee symposium on security and privacy (sp)*, 39–57. IEEE.

Celis, L. Elisa, Lingxiao Huang, Vijay Keswani, and Nisheeth K. Vishnoi. 2019. Classification with fairness constraints: A meta-algorithm with provable guarantees. In *Proceedings of the conference on fairness, accountability, and transparency, fat* 2019, atlanta, ga, usa, january 29-31, 2019*, 319–328. ACM.

Chang, Hongyan, Ta Duy Nguyen, Sasi Kumar Murakonda, Ehsan Kazemi, and Reza Shokri. 2020. On adversarial bias and the robustness of fair machine learning. *CoRR* abs/2006.08669. 2006.08669.

Chasins, Sarah, and Phitchaya Mangpo Phothilimthana. 2017. Data-driven synthesis of full probabilistic programs. In *International conference on computer aided verification*, 279–304. Springer.

Chaudhuri, Swarat, Martin Clochard, and Armando Solar-Lezama. 2014. Bridging boolean and quantitative synthesis using smoothed proof search. In *Popl*, vol. 49, 207–220. ACM.

Chaudhuri, Swarat, Sumit Gulwani, Roberto Lublinerman, and Sara Navidpour. 2011. Proving programs robust. In *Proceedings of the 19th acm sigsoft symposium and the 13th european conference on foundations of software engineering*, 102–112. ESEC/FSE '11, New York, NY, USA: ACM.

Chen, Taolue, Ernst Moritz Hahn, Tingting Han, Marta Kwiatkowska, Hongyang Qu, and Lijun Zhang. 2013. Model repair for markov decision processes. In *Theoretical aspects of software engineering (tase), 2013 international symposium on*, 85–92. IEEE.

Chen, Tianqi, and Carlos Guestrin. 2016. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, 785–794. ACM.

Chen, Xinyun, Chang Liu, Bo Li, Kimberly Lu, and Dawn Song. 2017. Targeted backdoor attacks on deep learning systems using data poisoning. *arXiv preprint arXiv:1712.05526*.

Chib, Siddhartha, and Edward Greenberg. 1995. Understanding the metropolis-hastings algorithm. *The american statistician* 49(4):327–335.

Chistikov, Dmitry, Rayna Dimitrova, and Rupak Majumdar. 2015. Approximate counting in SMT and value estimation for probabilistic programs. In *Tools and algorithms for the construction and analysis of systems - 21st international conference, TACAS 2015, held as part of the european joint conferences on theory and practice of software, ETAPS 2015, london, uk, april 11-18, 2015. proceedings*, 320–334.

Claret, Guillaume, Sriram K Rajamani, Aditya V Nori, Andrew D Gordon, and Johannes Borgström. 2013. Bayesian inference using data flow analysis. In *Proceedings of the 2013 9th joint meeting on foundations of software engineering*, 92–102. ACM.

Clarke, Edmund, Daniel Kroening, and Flavio Lerda. 2004. A tool for checking ansi-c programs. In *International conference on tools and algorithms for the construction and analysis of systems*, 168–176. Springer.

Converse, Hayes, Antonio Filieri, Divya Gopinath, and Corina S. Pasareanu. 2020. Probabilistic symbolic analysis of neural networks. In *31st IEEE international symposium on software reliability engineering, ISSRE 2020, coimbra, portugal, october 12-15, 2020*, ed. Marco Vieira, Henrique Madeira, Nuno Antunes, and Zheng Zheng, 148–159. IEEE.

Cousot, P., and R. Cousot. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints.

Cytron, Ron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13(4):451–490.

D'Antoni, Loris, Roopsha Samanta, and Rishabh Singh. 2016. *Qlose: Program repair with quantitative objectives*, 383–401. Cham: Springer International Publishing.

Datta, Amit, Michael Carl Tschantz, and Anupam Datta. 2015. Automated experiments on ad privacy settings. *Proceedings on Privacy Enhancing Technologies* 2015(1):92–112.

Datta, Anupam, Shayak Sen, and Yair Zick. 2016. Algorithmic transparency via quantitative input influence. In *Proceedings of 37th ieee symposium on security and privacy*.

De Loera, JA, Brandon Dutra, Matthias Koeppe, Stanislav Moreinis, Gregory Pinto, and Jianqiu Wu. 2012. Software for exact integration of polynomials over polyhedra. *ACM Communications in Computer Algebra* 45(3/4):169–172.

De Moura, Leonardo, and Nikolaj Bjørner. 2008. Z3: An efficient smt solver. In *International conference on tools and algorithms for the construction and analysis of systems*, 337–340. Springer.

Diakonikolas, Ilias., Gautam. Kamath, Daniel. Kane, Jerry. Li, Ankur. Moitra, and Alistair. Stewart. 2019a. Robust estimators in high-dimensions without the computational intractability. *SIAM Journal on Computing* 48(2):742–864. https://doi.org/10.1137/17M1126680.

Diakonikolas, Ilias, Gautam Kamath, Daniel Kane, Jerry Li, Jacob Steinhardt, and Alistair Stewart. 2019b. Sever: A robust meta-algorithm for stochastic optimization. In *Proceedings of the 36th international conference on machine learning, ICML 2019, 9-15 june 2019, long beach, california, USA*, ed. Kamalika Chaudhuri and Ruslan Salakhutdinov, vol. 97 of *Proceedings of Machine Learning Research*, 1596–1606. PMLR.

Diakonikolas, Ilias, and Daniel M. Kane. 2019. Recent advances in algorithmic high-dimensional robust statistics. *CoRR* abs/1911.05911. 1911.05911.

Dolzmann, Andreas, and Thomas Sturm. 1997. REDLOG: computer algebra meets computer logic. *SIGSAM Bull.* 31(2):2–9.

Drews, Samuel, Aws Albarghouthi, and Loris D'Antoni. 2019. Efficient synthesis with probabilistic constraints. In *Computer aided verification - 31st international conference, CAV 2019, new york city, ny, usa, july 15-18, 2019, proceedings, part I*, ed. Isil Dillig and Serdar Tasiran, vol. 11561 of *Lecture Notes in Computer Science*, 278–296. Springer.

———. 2020. Proving data-poisoning robustness in decision trees. In *Proceedings of the 41st ACM SIGPLAN international conference on programming language design and implementation, PLDI 2020, london, uk, june 15-20, 2020*, ed. Alastair F. Donaldson and Emina Torlak, 1083–1097. ACM.

Dua, Dheeru, and Casey Graff. 2017. UCI machine learning repository.

Dwork, Cynthia. 2006. Differential privacy. In *Automata, languages and programming*, 1–12. Springer.

Dwork, Cynthia, Moritz Hardt, Toniann Pitassi, Omer Reingold, and Richard S. Zemel. 2012. Fairness through awareness. In *Innovations in theoretical computer science 2012, cambridge, ma, usa, january 8-10, 2012*, 214–226.

Dwyer, Kenneth, and Robert Holte. 2007. Decision tree instability and active learning. In *Machine learning: ECML 2007, 18th european conference on machine learning, warsaw, poland, september 17-21, 2007, proceedings*, 128–139.

Dyer, Martin, Alan Frieze, and Ravi Kannan. 1991. A random polynomial-time algorithm for approximating the volume of convex bodies. *Journal of the ACM (JACM)* 38(1):1–17.

Dyer, Martin E., and Alan M. Frieze. 1988. On the complexity of computing the volume of a polyhedron. *SIAM Journal on Computing* 17(5): 967–974.

EEOC. 2014. Code of federal regulations. https://www.gpo.gov/fdsys/pkg/CFR-2014-title29-vol4/xml/CFR-2014-title29-vol4-part1607.xml. (Accessed on 06/18/2016).

El-Hassany, Ahmed, Petar Tsankov, Laurent Vanbever, and Martin Vechev. 2017. Network-wide configuration synthesis.

Eubanks, Virginia. 2015. The dangers of letting algorithms enforce policy. http://www.slate.com/articles/technology/future_tense/2015/04/the_dangers_of_letting_algorithms_enforce_policy.html. (Accessed on 06/18/2016).

Feldman, Michael, Sorelle A. Friedler, John Moeller, Carlos Scheidegger, and Suresh Venkatasubramanian. 2015. Certifying and removing disparate impact. In *Proceedings of the 21th ACM SIGKDD international conference on knowledge discovery and data mining, sydney, nsw, australia, august 10-13, 2015*, 259–268.

Filieri, Antonio, Corina S Păsăreanu, and Willem Visser. 2013. Reliability analysis in symbolic pathfinder. In *Proceedings of the 2013 international conference on software engineering*, 622–631. IEEE Press.

Friedler, Sorelle A., Carlos Scheidegger, and Suresh Venkatasubramanian. 2016. On the (im)possibility of fairness. *CoRR* abs/1609.07236.

Gehr, T., M. Mirman, D. Drachsler-Cohen, P. Tsankov, S. Chaudhuri, and M. Vechev. 2018. Ai2: Safety and robustness certification of neural networks with abstract interpretation. In *2018 ieee symposium on security and privacy (sp)*, 3–18.

Gehr, Timon, Sasa Misailovic, and Martin Vechev. 2016. Psi: Exact symbolic inference for probabilistic programs. In *Computer aided verification*. Springer.

Geldenhuys, Jaco, Matthew B Dwyer, and Willem Visser. 2012. Probabilistic symbolic execution. In *Proceedings of the 2012 international symposium on software testing and analysis*, 166–176. ACM.

Goel, Karan, Albert Gu, Yixuan Li, and Christopher Ré. 2020. Model patching: Closing the subgroup performance gap with data augmentation. *CoRR* abs/2008.06775. 2008.06775.

Goldberg, Paul W., and Mark Jerrum. 1995. Bounding the vapnik-chervonenkis dimension of concept classes parameterized by real numbers. *Machine Learning* 18(2-3):131–148.

Goodman, Noah D., Vikash K. Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. 2008. Church: a language for generative models. In *UAI 2008, proceedings of the 24th conference in uncertainty in artificial intelligence, helsinki, finland, july 9-12, 2008*, 220–229.

Gordon, Andrew D, Thomas A Henzinger, Aditya V Nori, and Sriram K Rajamani. 2014. Probabilistic programming. In *Proceedings of the on future of software engineering*, 167–181. ACM.

Grigorescu, Sorin, Bogdan Trasnea, Tiberiu Cocias, and Gigel Macesanu. 2020. A survey of deep learning techniques for autonomous driving.

Journal of Field Robotics 37(3):362–386. https://onlinelibrary.wiley.com/doi/pdf/10.1002/rob.21918.

Gulwani, Sumit. 2011. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th ACM SIGPLAN-SIGACT symposium on principles of programming languages, POPL 2011, austin, tx, usa, january 26-28, 2011*, 317–330.

———. 2014. Program synthesis. In *Software systems safety*, 43–75.

———. 2016. Programming by examples - and its applications in data wrangling. In *Dependable software systems engineering*, 137–158.

Hall, Mark, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. 2009. The weka data mining software: an update. *ACM SIGKDD explorations newsletter* 11(1):10–18.

Hao, Karen. 2019. This is how ai bias really happens—and why it's so hard to fix. *MIT Technology Review*.

Hardt, Moritz, Eric Price, and Nathan Srebro. 2016. Equality of opportunity in supervised learning. *CoRR* abs/1610.02413.

Hern, Alex. 2018. Google's solution to accidental algorithmic racism: ban gorillas. *The Guardian*.

Hoare, C. A. R. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12(10):576–580.

Jobstmann, Barbara, Andreas Griesmayer, and Roderick Bloem. 2005. Program repair as a game. In *International conference on computer aided verification*, 226–238. Springer.

John, Philips George, Deepak Vijaykeerthy, and Diptikalyan Saha. 2020. Verifying individual fairness in machine learning models. In *Proceedings*

of the thirty-sixth conference on uncertainty in artificial intelligence, UAI 2020, virtual online, august 3-6, 2020, ed. Ryan P. Adams and Vibhav Gogate, vol. 124 of *Proceedings of Machine Learning Research*, 749–758. AUAI Press.

Jovanović, Dejan, and Leonardo de Moura. 2013. Solving non-linear arithmetic. *ACM Commun. Comput. Algebra* 46(3/4):104–105.

Karpathy, Andrej. 2017. Software 2.0. https://medium.com/@karpathy/software-2-0-a64152b37c35. (Accessed on 11/28/2020).

Katz, Guy, Clark Barrett, David L Dill, Kyle Julian, and Mykel J Kochenderfer. 2017. Reluplex: An efficient smt solver for verifying deep neural networks. In *International conference on computer aided verification*, 97–117. Springer.

Kearns, Michael J., and Umesh V. Vazirani. 1994. *An introduction to computational learning theory*. Cambridge, MA, USA: MIT Press.

Khachiyan, Leonid. 1993. *Complexity of polytope volume computation*. Springer.

Kilbertus, Niki, Mateo Rojas Carulla, Giambattista Parascandolo, Moritz Hardt, Dominik Janzing, and Bernhard Schölkopf. 2017. Avoiding discrimination through causal reasoning. In *Advances in neural information processing systems 30*, ed. I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, 656–666. Curran Associates, Inc.

Klein, Gerwin, June Andronick, Matthew Fernandez, Ihor Kuz, Toby C. Murray, and Gernot Heiser. 2018. Formally verified software in the real world. *Commun. ACM* 61(10):68–77.

Kleinberg, Jon, Sendhil Mullainathan, and Manish Raghavan. 2017. Inherent trade-offs in the fair determination of risk scores. In *Itcs*.

Knight, Will. 2017. The dark secret at the heart of ai.

Kobie, Nicole. 2016. Who do you blame when an algorithm gets you fired? http://www.wired.co.uk/article/make-algorithms-accountable. (Accessed on 06/18/2016).

Koller, Daphne, and Nir Friedman. 2009. *Probabilistic graphical models: principles and techniques*. MIT press.

Könighofer, Robert, and Roderick Bloem. 2011. Automated error localization and correction for imperative programs. In *Formal methods in computer-aided design (fmcad), 2011*, 91–100. IEEE.

Kozen, Dexter. 1981. Semantics of probabilistic programs. *Journal of Computer and System Sciences* 22(3):328–350.

Kusner, Matt J, Joshua Loftus, Chris Russell, and Ricardo Silva. 2017. Counterfactual fairness. In *Advances in neural information processing systems 30*, ed. I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, 4066–4076. Curran Associates, Inc.

Kučera, Martin, Petar Tsankov, Timon Gehr, Marco Guarnieri, and Martin Vechev. 2017. Synthesis of probabilistic privacy enforcement. In *Proceedings of the 2017 acm sigsac conference on computer and communications security*, 391–408. CCS '17, New York, NY, USA: ACM.

Laishram, Ricky, and Vir Virander Phoha. 2016. Curie: A method for protecting SVM classifier from poisoning attack. *CoRR* abs/1606.01584. 1606.01584.

LeCun, Yann, Corinna Cortes, and Christopher J. C. Burges. The MNIST database of handwritten digits.

Li, Ruey-Hsia, and Geneva G. Belford. 2002. Instability of decision tree classification algorithms. In *Proceedings of the eighth ACM SIGKDD inter-*

national conference on knowledge discovery and data mining, july 23-26, 2002, edmonton, alberta, canada, 570–575.

Li, Yi, Tian Huat Tan, and Marsha Chechik. 2014. Management of time requirements in component-based systems. In *Fm 2014: Formal methods*, 399–415. Springer.

Littman, Michael L, Judy Goldsmith, and Martin Mundhenk. 1998. The computational complexity of probabilistic planning. *Journal of Artificial Intelligence Research* 9:1–36.

Mandal, Debmalya, Samuel Deng, Suman Jana, Jeannette M. Wing, and Daniel J. Hsu. 2020. Ensuring fairness beyond the training data. In *Advances in neural information processing systems 33: Annual conference on neural information processing systems 2020, neurips 2020, december 6-12, 2020, virtual*, ed. Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin.

Mardziel, Piotr, Stephen Magill, Michael Hicks, and Mudhakar Srivatsa. 2011. Dynamic enforcement of knowledge-based security policies. In *Computer security foundations symposium (csf), 2011 ieee 24th*, 114–128. IEEE.

Mazurowski, Maciej A., Mateusz Buda, Ashirbani Saha, and Mustafa R. Bashir. 2019. Deep learning in radiology: An overview of the concepts and a survey of the state of the art with focus on mri. *Journal of Magnetic Resonance Imaging* 49(4):939–954. https://onlinelibrary.wiley.com/doi/pdf/10.1002/jmri.26534.

Mechtaev, Sergey, Jooyong Yi, and Abhik Roychoudhury. 2015. Directfix: Looking for simple program repairs. In *Proceedings of the 37th international conference on software engineering-volume 1*, 448–458. IEEE Press.

Mei, Shike, and Xiaojin Zhu. 2015. Using machine teaching to identify optimal training-set attacks on machine learners. In *Proceedings of the twenty-ninth aaai conference on artificial intelligence*, 2871–2877. AAAI'15, AAAI Press.

Miller, Claire Cain. 2015. Can an algorithm hire better than a human? http://www.nytimes.com/2015/06/26/upshot/can-an-algorithm-hire-better-than-a-human.html. (Accessed on 06/18/2016).

Mirman, Matthew, Timon Gehr, and Martin T. Vechev. 2018. Differentiable abstract interpretation for provably robust neural networks. In *Proceedings of the 35th international conference on machine learning, ICML 2018, stockholmsmässan, stockholm, sweden, july 10-15, 2018*, ed. Jennifer G. Dy and Andreas Krause, vol. 80 of *Proceedings of Machine Learning Research*, 3575–3583. PMLR.

Monniaux, David. 2000. Abstract interpretation of probabilistic semantics. In *Static analysis*, 322–339. Springer.

———. 2001a. An abstract monte-carlo method for the analysis of probabilistic programs. In *Acm sigplan notices*, vol. 36, 93–101. ACM.

———. 2001b. Backwards abstract interpretation of probabilistic programs. In *Programming languages and systems*, 367–382. Springer.

Nair, Vinod, and Geoffrey E Hinton. 2010. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (icml-10)*, 807–814.

Natarajan, B. K. 1989. On learning sets and functions. *Machine Learning* 4(1):67–97.

Newell, Andrew, Rahul Potharaju, Luojie Xiang, and Cristina Nita-Rotaru. 2014. On the practicality of integrity attacks on document-level sentiment

analysis. In *Proceedings of the 2014 workshop on artificial intelligent and security workshop*, 83–93. AISec '14, New York, NY, USA: ACM.

Nori, Aditya V., Sherjil Ozair, Sriram K. Rajamani, and Deepak Vijaykeerthy. 2015. Efficient synthesis of probabilistic programs. *SIGPLAN Not.* 50(6):208–217.

Papadimitriou, Christos H. 1985. Games against nature. *Journal of Computer and System Sciences* 31(2):288–301.

Pedreshi, Dino, Salvatore Ruggieri, and Franco Turini. 2008. Discrimination-aware data mining. In *Proceedings of the 14th acm sigkdd international conference on knowledge discovery and data mining*, 560–568. ACM.

Pérez, Jesús M., Javier Muguerza, Olatz Arbelaitz, Ibai Gurrutxaga, and José Ignacio Martín. 2005. Consolidated trees: Classifiers with stable explanation. A model to achieve the desired stability in explanation. In *Pattern recognition and data mining, third international conference on advances in pattern recognition, ICAPR 2005, bath, uk, august 22-25, 2005, proceedings, part I*, 99–107.

Perry, Walt L. 2013. *Predictive policing: The role of crime forecasting in law enforcement operations*. Rand Corporation.

Polozov, Oleksandr, and Sumit Gulwani. 2015. Flashmeta: a framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN international conference on object-oriented programming, systems, languages, and applications, OOPSLA 2015, part of SPLASH 2015, pittsburgh, pa, usa, october 25-30, 2015*, 107–126.

Quinlan, J. Ross. 1986. Induction of decision trees. *Machine learning* 1(1): 81–106.

Quinlan, J Ross. 1993. C4.5: Programs for machine learning. *The Morgan Kaufmann Series in Machine Learning, San Mateo, CA: Morgan Kaufmann, | c1993*.

Quinlan, J.R. 1987. Simplifying decision trees. *International Journal of Man-Machine Studies* 27(3):221 – 234.

Ranzato, Francesco, and Marco Zanella. 2020. Abstract interpretation of decision tree ensemble classifiers. In *Proceedings of the thirty-fourth aaai conference on artificial intelligence*, ed. V. Conitzer and F. Sha. AAAI'20.

Rawls, John. 2009. *A theory of justice*. Harvard university press.

Raza, Mohammad, and Sumit Gulwani. 2017. Automated data extraction using predictive program synthesis. In *Proceedings of the thirty-first AAAI conference on artificial intelligence, february 4-9, 2017, san francisco, california, USA.*, 882–890.

Ringer, Talia, Karl Palmskog, Ilya Sergey, Milos Gligoric, and Zachary Tatlock. 2019. QED at large: A survey of engineering of formally verified software. *Found. Trends Program. Lang.* 5(2-3):102–281.

Rosenfeld, Elan, Ezra Winston, Pradeep Ravikumar, and Zico Kolter. 2020. Certified robustness to label-flipping attacks via randomized smoothing. In *Proceedings of the 37th international conference on machine learning, ICML 2020, 13-18 july 2020, virtual event*, vol. 119 of *Proceedings of Machine Learning Research*, 8230–8241. PMLR.

Rudin, Cynthia, Caroline Wang, and Beau Coker. 2020. The age of secrecy and unfairness in recidivism prediction. *Harvard Data Science Review* 2(1). Https://hdsr.mitpress.mit.edu/pub/7z10o269.

Ruggieri, Salvatore. 2014. Using t-closeness anonymity to control for non-discrimination. *Transactions on Data Privacy* 7(2):99–129.

Saad, Feras A, Marco F Cusumano-Towner, Ulrich Schaechtle, Martin C Rinard, and Vikash K Mansinghka. 2019. Bayesian synthesis of probabilistic programs for automatic data modeling. *Proceedings of the ACM on Programming Languages* 3(POPL):37.

Sagawa, Shiori, Pang Wei Koh, Tatsunori B. Hashimoto, and Percy Liang. 2019. Distributionally robust neural networks for group shifts: On the importance of regularization for worst-case generalization. *CoRR* abs/1911.08731. 1911.08731.

Sampson, Adrian, Pavel Panchekha, Todd Mytkowicz, Kathryn S McKinley, Dan Grossman, and Luis Ceze. 2014. Expressing and verifying probabilistic assertions. In *Acm sigplan notices*, vol. 49, 112–122. ACM.

Sánchez-Monedero, Javier, Lina Dencik, and Lilian Edwards. 2020. What does it mean to 'solve' the problem of discrimination in hiring?: social, technical and legal perspectives from the UK on automated hiring systems. In *Fat* '20: Conference on fairness, accountability, and transparency, barcelona, spain, january 27-30, 2020*, ed. Mireille Hildebrandt, Carlos Castillo, Elisa Celis, Salvatore Ruggieri, Linnet Taylor, and Gabriela Zanfir-Fortuna, 458–468. ACM.

Sankaranarayanan, Sriram, Aleksandar Chakarov, and Sumit Gulwani. 2013. Static analysis for probabilistic programs: inferring whole program properties from finitely many paths. In *ACM SIGPLAN conference on programming language design and implementation, PLDI '13, seattle, wa, usa, june 16-19, 2013*, 447–458.

Sauer, Norbert. 1972. On the density of families of sets. *Journal of Combinatorial Theory, Series A* 13(1):145–147.

Schkufza, Eric, Rahul Sharma, and Alex Aiken. 2016. Stochastic program optimization. *Commun. ACM* 59(2):114–122.

Sharma, Rahul, Aditya V. Nori, and Alex Aiken. 2014. Bias-variance tradeoffs in program analysis. In *Proceedings of the 41st acm sigplan-sigact symposium on principles of programming languages*, 127–137. POPL '14, New York, NY, USA: ACM.

Shelah, Saharon. 1972. A combinatorial problem; stability and order for models and theories in infinitary languages. *Pacific Journal of Mathematics* 41(1):247–261.

Singh, Gagandeep, Timon Gehr, Markus Püschel, and Martin Vechev. 2019. An abstract domain for certifying neural networks. *Proceedings of the ACM on Programming Languages* 3(POPL):41.

Solar-Lezama, Armando. 2008. Program synthesis by sketching. Ph.D. thesis, Berkeley, CA, USA. AAI3353225.

Srinivasan, Venkatesh, and Thomas W. Reps. 2015. Synthesis of machine code from semantics. In *Proceedings of the 36th ACM SIGPLAN conference on programming language design and implementation, portland, or, usa, june 15-17, 2015*, 596–607.

Steinhardt, Jacob, Pang Wei W Koh, and Percy S Liang. 2017. Certified defenses for data poisoning attacks. In *Advances in neural information processing systems*, 3517–3529.

Subramanian, Kausik, Loris D'Antoni, and Aditya Akella. 2017. Genesis: synthesizing forwarding tables in multi-tenant networks. In *Proceedings of the 44th ACM SIGPLAN symposium on principles of programming languages, POPL 2017, paris, france, january 18-20, 2017*, 572–585.

Sweeney, Latanya. 2013. Discrimination in online ad delivery. *Queue* 11(3):10.

Szegedy, Christian, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. 2014. Intriguing properties of neural networks. In *2nd international conference on learning representations, ICLR 2014, banff, ab, canada, april 14-16, 2014, conference track proceedings*.

Turney, Peter D. 1995. Technical note: Bias and the quantification of stability. *Machine Learning* 20(1-2):23–33.

Tutt, Andrew. 2016. An fda for algorithms. *Available at SSRN 2747994*.

UCI. 1996. Uci machine learning repository: Census income. https://archive.ics.uci.edu/ml/datasets/Adult/.

Valentino-Devries, Jennifer, Jeremy Singer-Vine, and Ashkan Soltani. 2012. Websites vary prices, deals based on users' information. http://www.wsj.com/articles/SB10001424127887323777204578189391813881534. (Accessed on 06/18/2016).

Vempala, Santosh. 2005. Geometric random walks: a survey. *Combinatorial and computational geometry* 52(573-612):2.

Von Essen, Christian, and Barbara Jobstmann. 2015. Program repair without regret. *Formal Methods in System Design* 47(1):26–50.

Wang, Shiqi, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. 2018a. Formal security analysis of neural networks using symbolic intervals. In *27th {USENIX} security symposium ({USENIX} security 18)*, 1599–1614.

Wang, Xinyu, Sumit Gulwani, and Rishabh Singh. 2016. FIDEX: filtering spreadsheet data using examples. In *Proceedings of the 2016 ACM SIGPLAN international conference on object-oriented programming, systems, languages, and applications, OOPSLA 2016, part of SPLASH 2016, amsterdam, the netherlands, october 30 - november 4, 2016*, 195–213.

Wang, Yizhen, Somesh Jha, and Kamalika Chaudhuri. 2018b. Analyzing the robustness of nearest neighbors to adversarial examples. In *Proceedings of the 35th international conference on machine learning, ICML 2018, stockholmsmässan, stockholm, sweden, july 10-15, 2018*, 5120–5129.

Weber, Maurice, Xiaojun Xu, Bojan Karlas, Ce Zhang, and Bo Li. 2020. RAB: provable robustness against backdoor attacks. *CoRR* abs/2003.08904. 2003.08904.

Wieringa, Maranke. 2020. What to account for when accounting for algorithms: a systematic literature review on algorithmic accountability. In *Fat* '20: Conference on fairness, accountability, and transparency, barcelona, spain, january 27-30, 2020*, ed. Mireille Hildebrandt, Carlos Castillo, Elisa Celis, Salvatore Ruggieri, Linnet Taylor, and Gabriela Zanfir-Fortuna, 1–18. ACM.

Wong, Eric, and J. Zico Kolter. 2018. Provable defenses against adversarial examples via the convex outer adversarial polytope. In *Proceedings of the 35th international conference on machine learning, ICML 2018, stockholmsmässan, stockholm, sweden, july 10-15, 2018*, ed. Jennifer G. Dy and Andreas Krause, vol. 80 of *Proceedings of Machine Learning Research*, 5283–5292. PMLR.

Xiao, Han, Huang Xiao, and Claudia Eckert. 2012. Adversarial label flips attack on support vector machines. In *Proceedings of the 20th european conference on artificial intelligence*, 870–875. ECAI'12, Amsterdam, The Netherlands, The Netherlands: IOS Press.

Xiao, Huang, Battista Biggio, Gavin Brown, Giorgio Fumera, Claudia Eckert, and Fabio Roli. 2015a. Is feature selection secure against training data poisoning? In *International conference on machine learning*, 1689–1698.

Xiao, Huang, Battista Biggio, Blaine Nelson, Han Xiao, Claudia Eckert, and Fabio Roli. 2015b. Support vector machines under adversarial label contamination. *Neurocomput.* 160(C):53–62.

Zarsky, Tal. 2014. Understanding discrimination in the scored society. *Washington Law Review* 89(4).

Zemel, Richard S., Yu Wu, Kevin Swersky, Toniann Pitassi, and Cynthia Dwork. 2013. Learning fair representations. In *Proceedings of the 30th international conference on machine learning, ICML 2013, atlanta, ga, usa, 16-21 june 2013*, 325–333.

A.1 Real-Valued Features for DTrace

In this appendix, we provide a complete exposition of the technique used to handle real-valued features in Antidote (filling the gap left in Section 2.4). We repeat some of the arguments given in the main text to make this appendix self-contained. For real-valued features, there are infinitely many possible predicates of the form $\lambda x_i.\, x_i \leqslant \tau$ (where $\tau \in \mathbb{R}$), and the learner $\mathsf{DTrace}_{\mathbb{R}}$ chooses a finite set of possible τ values dynamically, based on the values that occur in T. Throughout this section, we use the subscript $\mathbb{R}$ to denote the real-valued versions of existing operations.

From DTrace to DTrace$_{\mathbb{R}}$

The new learner $\mathsf{DTrace}_{\mathbb{R}}$ is almost identical to DTrace. However, to formalize the aforementioned operation in the concrete semantics of $\mathsf{DTrace}_{\mathbb{R}}$, we make a single modification in $\mathsf{bestSplit}_{\mathbb{R}}$, which maintains the original definition of bestSplit, but it first computes a finite set of predicates $\Phi_{\mathbb{R}}$ used in the remainder of its computation: consider all of the values appearing in T for the ith feature in $\mathcal{X}$, sorted in ascending order. For each pair of adjacent values (a, b) (i.e., such that there exists no c in T such that $a < c < b$), we include in $\Phi_{\mathbb{R}}$ the predicate $\varphi = \lambda x_i.\, x_i \leqslant \frac{a+b}{2}$.

Example A.1. *In our running example from Figure 2.2, we have training set elements in* T_{bw} *whose features take the numeric values* $\{0, 1, 2, 3, 4, 7, \ldots, 14\}$. $\mathsf{bestSplit}_{\mathbb{R}}(T_{bw})$ *would thus enumerate over the predicates* $\Phi_{\mathbb{R}} = \{\lambda x.\, x \leqslant \tau \mid \tau \in \{\frac{1}{2}, \frac{3}{2}, \frac{5}{2}, \frac{7}{2}, \frac{11}{2}, \frac{15}{2}, \ldots, \frac{27}{2}\}\}$.

From DTrace$^{\#}$ to DTrace$^{\#}_{\mathbb{R}}$

The formalization for the abstract case is more involved than the concrete case: (*i*) we will similarly modify bestSplit$^{\#}$, but also (*ii*) we will change our abstract domain over predicates. This change to the predicate domain means we will have to make largely superficial adjustments to the many of the other operations, as well.

bestSplit$^{\#}_{\mathbb{R}}$, the real-valued version of bestSplit$^{\#}$, is responsible for selecting a finite set of predicates that it will consider in its computation. This motivates the second point, which we will discuss first: because we don't know *which* values could be missing from $\langle T, n\rangle$, we might naively consider a value of τ for every such possible combination of missing values. If we did so, and if bestSplit$_{\mathbb{R}}$T considers m predicates, then bestSplit$^{\#}_{\mathbb{R}}(\langle T, n\rangle)$ might consider up to $\approx m^n$ predicates. For efficiency, we will instead create a finite set of m-many *symbolic predicates* that overapproximates these possibilities.

Definition A.2 (Symbolic Real-Valued Predicate)**.** *For a real-valued feature at* x_i*, a* symbolic predicate ρ *over* x_i *takes the form* $\lambda x_i.\, x_i \leqslant [a, b)$ *for real values* a *and* b*. The semantics of a symbolic predicate is three-valued, which we denote as follows:*

$$\rho(x) \coloneqq \begin{cases} \textit{true} & x_i \leqslant a \\ \textit{maybe} & a < x_i < b \\ \textit{false} & b \leqslant x_i \end{cases}$$

Each symbolic predicate $\rho = \lambda x_i.\, x_i \leqslant [a, b)$ *has the concretization* $\gamma(\rho) \coloneqq \{\lambda x_i.\, x_i \leqslant \tau \mid \tau \in [a, b)\}$.

Without loss of generality, we focus on the single disjunct case. We previously stated DTrace$^{\#}$ operates over a state $(\langle T, n\rangle, \Psi)$ where Ψ is some finite set of predicates; now we let the state of DTrace$^{\#}_{\mathbb{R}}$ be $(\langle T, n\rangle, \Psi^{\#})$ where $\Psi^{\#}$ is represented as a finite set of *symbolic predicates*. The join remains a

simple set union $\Psi_1^\# \sqcup \Psi_2^\# \coloneqq \Psi_1^\# \cup \Psi_2^\#$. The concretization captures an infinite set of (non-symbolic) predicates: $\gamma(\Psi^\#) \coloneqq \bigcup_{\rho \in \Psi^\#} \gamma(\rho)$.

Symbolic Predicates in Auxiliary Operators. The definition of $\langle T, n\rangle\downarrow_\rho^\#$ changes slightly, since we must include the *maybe* case to be sound. This complicates the computation of the poisoning amount n, since we have two sources of uncertainty: we must account for elements that could be missing because either (*i*) they are missing from some particular $T' \in \gamma(\langle T, n\rangle)$, or (*ii*) ρ evaluates to *maybe*. Fortunately, we will be able to succinctly encompass these possibilities using our existing lower-level operations.

We define $\langle T, n\rangle\downarrow_\rho^\#$ as follows: suppose ρ is of the form $\lambda x_i.\, x_i \leqslant [a, b)$, let $\varphi_a = \lambda x_i.\, x_i \leqslant a$ and let $\varphi_b = \lambda x_i.\, x_i < b$. Because φ_a and φ_b are concrete predicates, we can use Equation 2.1 (Section 2.3) to compute their abstract semantics. Then

$$\langle T, n\rangle\downarrow_\rho^\# \coloneqq \langle T, n\rangle\downarrow_{\varphi_a}^\# \sqcup \langle T, n\rangle\downarrow_{\varphi_b}^\#.$$

Proposition A.3. *Let* $T' \in \gamma(\langle T, n\rangle)$ *and* $\varphi' \in \gamma(\rho)$. *Then,*

$$T'\downarrow_{\varphi'} \in \gamma(\langle T, n\rangle\downarrow_\rho^\#).$$

Proof. Denote $\langle S_a, m_a\rangle = \langle T, n\rangle\downarrow_{\varphi_a}^\#$ and $\langle S_b, m_b\rangle = \langle T, n\rangle\downarrow_{\varphi_b}^\#$. We will show that $T'\downarrow_{\varphi'} \in \gamma(\langle S_a, m_a\rangle \sqcup \langle S_b, m_b\rangle)$. This join has nice structure since $a \leqslant b$ and thus $S_a \subseteq S_b$.

We break into two cases, since $m_b = \min(n, |S_b|)$. First, suppose $m_b = |S_b|$. Then $\langle S_a, m_a\rangle \sqcup \langle S_b, m_b\rangle = \langle S_b, |S_b|\rangle$, where $\gamma(\langle S_b, |S_b|\rangle) = \mathcal{P}(T\downarrow_{\varphi_b})$. Because $\varphi' \in \gamma(\rho)$, we then have $T'\downarrow_{\varphi'} \subseteq T'\downarrow_{\varphi_b} \in \mathcal{P}(T\downarrow_{\varphi_b})$, and we are done.

Otherwise, we have $m_b = n$. Here, $\langle S_a, m_a\rangle \sqcup \langle S_b, m_b\rangle = \langle S_b, |S_b \setminus S_a| + n\rangle$ (unless $m_a = |S_a|$, in which case we immediately collapse to the previous case). Again, because $\varphi' \in \gamma(\rho)$, we immediately have that $T'\downarrow_{\varphi'} \subseteq S_b$; it remains to show that $|S_b \setminus T'\downarrow_{\varphi'}| \leqslant |S_b \setminus S_a| + n$. Observe

that (*i*) $T{\downarrow_{\varphi'}} \setminus T'{\downarrow_{\varphi'}} \subseteq T \setminus T'$ and thus $|T{\downarrow_{\varphi'}} \setminus T'{\downarrow_{\varphi'}}| \leqslant n$, and (*ii*) since $S_a \subseteq S_b$ and $\varphi' \in \gamma(\rho)$, we have that $|S_b \setminus T{\downarrow_{\varphi'}}| \leqslant |S_b \setminus S_a|$. Combined, we know that $|S_b \setminus T'{\downarrow_{\varphi'}}| = |S_b \setminus T{\downarrow_{\varphi'}}| + |T{\downarrow_{\varphi'}} \setminus T'{\downarrow_{\varphi'}}| \leqslant |S_b \setminus S_a| + n$.

Symbolic Predicates in $\mathsf{filter}^{\#}_{\mathbb{R}}$**.** In the original definition of $\mathsf{filter}^{\#}$ we separated Ψ into two sets Ψ_x and $\Psi_{\neg x}$ because exclusively either $\varphi \models x$ or $\neg\varphi \models x$. In this new three-valued symbolic predicate case, we appropriately over-approximate the $\rho(x) = \textit{maybe}$ possibility.

Let us denote the following:

$$\Psi^{\#}_x = \{\rho \in \Psi^{\#} \mid \rho(x) \in \{\textit{true}, \textit{maybe}\}\}$$
$$\Psi^{\#}_{\neg x} = \{\rho \in \Psi^{\#} \mid \rho(x) \in \{\textit{maybe}, \textit{false}\}\}$$

and define

$$\mathsf{filter}^{\#}_{\mathbb{R}}(\langle T, n\rangle, \Psi^{\#}, x) \coloneqq \left(\bigsqcup_{\rho \in \Psi^{\#}_x} \langle T, n\rangle{\downarrow^{\#}_{\rho}}\right) \sqcup \left(\bigsqcup_{\rho \in \Psi^{\#}_{\neg x}} \langle T, n\rangle{\downarrow^{\#}_{\neg\rho}}\right)$$

Proposition A.4. *Let* $T' \in \gamma(\langle T, n\rangle)$ *and let* $\varphi' \in \gamma(\Psi^{\#})$*. Then,*

$$\mathsf{filter}_{\mathbb{R}}(T', \varphi', x) \in \gamma\big(\mathsf{filter}^{\#}_{\mathbb{R}}(\langle T, n\rangle, \Psi^{\#}, x)\big)$$

Proof. Because $\varphi' \in \gamma(\Psi^{\#})$, we know there is some $\rho \in \Psi^{\#}$ such that $\varphi' \in \gamma(\rho)$. Once again, soundness now follows from the soundness of the join and the soundness of $\downarrow^{\#}_{\rho}$.

Symbolic Predicates in $\mathsf{bestSplit}^{\#}_{\mathbb{R}}$**.** Finally, we discuss the treatment of our favorite complicated instruction. Perhaps surprisingly, very little has to change. Indeed, $\mathsf{bestSplit}^{\#}_{\mathbb{R}}(\langle T, n\rangle)$ begins by creating a finite set of symbolic predicates $\Phi^{\#}$ and then proceeds exactly as its original definition.

The construction of $\Phi^\#$ is very simple. In the concrete case we considered $\lambda x_i.\, x_i \leqslant \frac{a+b}{2}$ for all adjacent pairs (a, b) of the feature values that occur in T; here, we will consider $\lambda x_i.\, x_i \leqslant [a, b)$ for all such pairs.

It should be surprising that this computation is sound. After all, it involves (effectively) computing the argmin over a set of predicates; here, we're *over-approximating* that set, and we ought to be concerned that our over-approximation could include extraneous predicates whose valuation is so small that they occlude the feasible minimizing predicates. Serendipitously, the choice of $\Phi^\#$ prevents this from happening.

Lemma A.5. *Let* $T' \in \gamma(\langle T, n\rangle)$. *Then,*

$$\mathsf{bestSplit}_{\mathbb{R}}(T') \in \gamma\big(\mathsf{bestSplit}^{\#}_{\mathbb{R}}(\langle T, n\rangle)\big)$$

Proof. We begin by observing two facts about the $\Phi^\#$ constructed in $\mathsf{bestSplit}^{\#}_{\mathbb{R}}$: (*i*) For every $\varphi \in \Phi$ built during the non-symbolic $\mathsf{bestSplit}_{\mathbb{R}}(T')$, there exists some $\rho \in \Phi^\#$ such that $\varphi \in \gamma(\rho)$. (*ii*) For every $\rho \in \Phi^\#$, there exists some $T'' \in \gamma(\langle T, n\rangle)$ and $\varphi'' \in \Phi$ from $\mathsf{bestSplit}_{\mathbb{R}}(T'')$ (φ'' is not necessarily returned as optimal, just constructed for consideration) such that $\varphi'' \in \gamma(\rho)$.

Let $\varphi' = \mathsf{bestSplit}_{\mathbb{R}}(T')$. The rest of the proof is exactly the same as the soundness proof for $\mathsf{bestSplit}^{\#}$ (the previous two observations ensure that $\Phi^{\#}_{\forall}$ and $\Phi^{\#}_{\exists}$ preserve the properties necessary for those arguments), except for establishing that our target φ' is in the concretization of some ρ' returned in the then-branch of the definition. Take any $\varphi^* \in \gamma(\rho^*)$ where ρ^* is the minimizer in $\mathsf{lub}_{\Phi^\#}$. Observe the following for any $\rho' \in \gamma(\Phi^\#)$

such that $\varphi' \in \gamma(\rho')$:

$$\begin{aligned}
\mathrm{lb}(\mathsf{score}^{\#}(\langle \mathsf{T}, \mathfrak{n} \rangle, \rho')) &\leqslant \mathrm{lb}(\mathsf{score}^{\#}(\langle \mathsf{T}, \mathfrak{n} \rangle, \varphi')) \\
&\leqslant \mathsf{score}(\mathsf{T}', \varphi') \\
&\leqslant \mathsf{score}(\mathsf{T}', \varphi^*) \\
&\leqslant \mathrm{ub}(\mathsf{score}^{\#}(\langle \mathsf{T}, \mathfrak{n} \rangle, \varphi^*)) \\
&\leqslant \mathrm{ub}(\mathsf{score}^{\#}(\langle \mathsf{T}, \mathfrak{n} \rangle, \rho^*)) \\
&= \mathrm{lub}_{\Phi^{\#}}
\end{aligned}$$

and thus φ' would be included.

This Appendix contains various proofs omitted from the main text.

B.1 Proofs for Antidote

Proof of Proposition 2.7. Suppose $T \in \gamma(\langle T_1, n_1 \rangle) \cup \gamma(\langle T_2, n_2 \rangle)$. Without loss of generality, assume $T \in \gamma(\langle T_1, n_1 \rangle)$. Certainly $T \subseteq T_1 \cup T_2$; furthermore, observe that T can be formed by first removing $|T_2 \setminus T_1|$-many elements from $T_1 \cup T_2$ to recover T_1 and then removing $\leqslant n_1$ remaining elements: this gives us $|(T_1 \cup T_2) \setminus T| = |(T_2 \setminus T_1) \cup (T_1 \setminus T)| \leqslant |T_2 \setminus T_1| + |T_1 \setminus T| \leqslant |T_2 \setminus T_1| + n_1$. This matches the definition of $\sqcup$.

Proof of Proposition 2.9. Let $\langle S, m \rangle = \gamma(\langle T, n \rangle \downarrow_\varphi^\#)$ (so we will show $T' \downarrow_\varphi \in \gamma(\langle S, m \rangle)$). Since $T' \subseteq T$, we have that $T' \downarrow_\varphi \subseteq \{(x, y) \in T : x \models \varphi\} = S$, thus $T' \downarrow_\varphi \subseteq S$. Additionally, $S \setminus T' \downarrow_\varphi = S \setminus T' \subseteq T \setminus T'$; certainly, then, $|S \setminus T' \downarrow_\varphi| \leqslant |T \setminus T'|$ and thus $\leqslant n$. Recall there are two cases, $m = n$ or $m = |S|$: for the latter, we have trivially that $|S \setminus T' \downarrow_\varphi| \leqslant |S| \leqslant m$. Therefore $|S \setminus T' \downarrow_\varphi| \leqslant \min(n, m)$.

Proof of Proposition 2.10. When $n < |T|$, this follows from the soundness of interval arithmetic. In our corner case for $n = |T|$, the concrete cprob is undefined behavior, so we encompass every well-formed categorical probability distribution by allowing each component to take any $[0, 1]$ value.

Proof of Proposition 2.12. This is an immediate consequence of the soundness of $\langle T, n \rangle \downarrow_{\varphi'}^\#$ and the soundness of the join.

Proof of Lemma 2.15. First, observe that $\text{score}^\#$ is sound since it involves composing sound interval arithmetic with other sound operations. Let $\varphi' = \text{bestSplit}(T')$.

- Case $\varphi' = \diamond$: By definition of bestSplit, this occurs when T' is such that every predicate $\varphi \in \Phi$ results in trivial splits. Soundness of $\downarrow^{\#}$ then gives us that for each $\varphi \in \Phi$, either $\emptyset \in \gamma(\langle T, n\rangle\downarrow^{\#}_{\varphi})$ or $\emptyset \in \gamma(\langle T, n\rangle\downarrow^{\#}_{\neg\varphi})$, thus $\Phi_{\forall} = \emptyset$. We return using the then-branch, which includes $\diamond$.

- Case $\varphi' \neq \diamond$: By definition of bestSplit, (*i*) φ' non-trivially splits T', and furthermore, (*ii*) for all other ψ that non-trivially split T', we have that $\mathsf{score}(T', \varphi') \leqslant \mathsf{score}(T', \psi)$. (*i*) gives us that $\varphi' \in \Phi_{\exists}$, therefore if $\Phi_{\forall} = \emptyset$, then φ' is included in the return value.

  Otherwise, when $\Phi_{\forall} \neq \emptyset$, we return using the then-branch. Let ψ^* minimize $\mathrm{lub}_{\Phi_{\forall}}$: since $\psi^* \in \Phi_{\forall}$, (*ii*) gives us that $\mathsf{score}(T', \varphi') \leqslant \mathsf{score}(T', \psi^*)$, and thus we have $\mathrm{lb}(\mathsf{score}^{\#}(\langle T, n\rangle, \varphi')) \leqslant \mathrm{ub}(\mathsf{score}^{\#}(\langle T, n\rangle, \psi^*)) = \mathrm{lub}_{\Phi_{\forall}}$. Therefore φ' is included in the return.

Proof of Theorem 2.16. $\mathsf{DTrace}^{\#}$ applies a sequence of operations; throughout the section, we state the soundness of each of these operations. The soundness of $\mathsf{DTrace}^{\#}$ follows from taking their composition.

B.2 Proofs for FairSquare

Proof of Theorem 3.7. **Soundness:** Suppose $m \models \square_{\varphi}$. By definition of $\square_{\varphi}$, the following formula is valid

$$\bigwedge_{x \in X_{\varphi}} m(l_x) \leqslant x \leqslant m(u_x) \Rightarrow \varphi$$

Therefore $H^m \Rightarrow \varphi$, by definition of H^m.

Completeness: Let H be a hyperrectangle such that $H \Rightarrow \varphi$. By definition, H is of the form $\bigwedge_{x \in X_{\varphi}} c_x \leqslant x \leqslant c'_x$. It immediately follows that the model where $l_x = c_x$ and $u_x = c'_x$, for every $x \in X_{\varphi}$, satisfies $\square_{\varphi}$, since $c'_x > c_x$ (satisfying the first conjunct of $\square_{\varphi}$), and $\forall X_{\varphi}. H \Rightarrow \varphi$ (satisfying the second conjunct of $\square_{\varphi}$).

Proof of Theorem 3.8. **Soundness:** Suppose $H^{m_1} \wedge H^{m_2}$ is satisfiable. By definition of a hyperrectangle, this means that for all variables $x \in X_\varphi$, we have that the intervals $[H_l^{m_1}(x), H_u^{m_1}(x)]$ and $[H_l^{m_2}(x), H_u^{m_2}(x)]$ overlap, i.e., at least are equal on one of the extremes. Therefore, $m_2 \not\models \Psi \wedge \mathit{block}(H^{m_1})$, since *block* does not admit any model m where, for all $x \in X_\varphi$, $[m(l_x), m(u_x)]$ overlaps with $[H_l^{m_1}(x), H_u^{m_1}(x)]$.

Completeness: Suppose $H^{m_1} \wedge H^{m_2}$ is unsatisfiable. By definition of a hyperrectangle, there is at least one $x \in X_\varphi$ where $[H_l^{m_1}(x), H_u^{m_1}(x)]$ and $[H_l^{m_2}(x), H_u^{m_2}(x)]$ do not overlap. Therefore, if $m_2 \models \Psi$, then $m_2 \models \Psi \wedge \mathit{block}(H^{m_1})$, since *block* explicitly states that for at least one variable x, $[m(l_x), m(u_x)]$ should not overlap with $[H_l^{m_1}(x), H_u^{m_1}(x)]$.

Proof of Theorem 3.13. **Soundness:** Suppose $m \models \mathit{step}^\Phi(x)$. Then,

$$m(\delta_x) = \sum_{i=1}^{n} c_i \cdot \left|[a_i, b_i] \cap [m(l_x), m(u_x)]\right|$$

By definition of the area under a positive step function, we have

$$m(\delta_x) = \int_{m(l_x)}^{m(u_x)} \mathit{step}(x)\ dx$$

Completeness: Completeness easily follows from correctness of the encoding of integrals over step functions as sums.

B.3 Proofs for DIGITS

Proof of Lemma 4.4. We cite the seminal result from Blumer et al. (1989) 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

[1] See Blumer et al. (1989, Appendix 1) for a discussion of well-behaved concept classes.

concept $c^* \in C$, any concept $c \in C$ consistent with those samples has $error_{\mathcal{D}}(c) \leqslant \varepsilon$ with probability at least $1 - \delta$. Here, $error_{\mathcal{D}}(c)$ is the probability that a sample drawn from $\mathcal{D}$ is classified differently by the two concepts c^* and 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 $P \in \mathcal{P}$, some labeling of the $\geqslant \frac{4}{\varepsilon}\log_2\frac{2}{\delta} + \frac{8k}{\varepsilon}\log_2\frac{13}{\varepsilon}$ samples is consistent with P, and therefore the theorem from Blumer et al. (1989) applies.

Proof of Theorem 4.6. By Lemma 4.4, we know that with probability $\geqslant 1 - \delta$, one of the candidate programs P' enumerated by DIGITS will have $d_{\mathbb{D}}(P, P;) \leqslant \varepsilon$. Since P is α-robust and $\varepsilon \leqslant \alpha$, then $P' \in B_\alpha(P)$ and thus $[\![P']\!](\mathbb{D}) \models \mathit{post}$.

Proof of Corollary 4.7. This follows immediately from Theorem 4.6 by letting $P = P^*$.

Proof of Corollary 4.8. Observe that $d_{\mathbb{D}}$ respects the triangle inequality, i.e. $d_{\mathbb{D}}(P_1, P_2) \leqslant d_{\mathbb{D}}(P_1, P_3) + d_{\mathbb{D}}(P_3, P_2)$.

$$\begin{aligned}
d_{\mathbb{D}}(P_1, P_2) &= \Pr[P_1 \neq P_2] \\
&= \Pr[P_1 \neq P_2 \wedge P_1 \neq P_3] + \Pr[P_1 \neq P_2 \wedge P_3 = P_1] \\
&= \Pr[P_1 \neq P_2 \wedge P_1 \neq P_3] + \Pr[P_1 \neq P_2 \wedge P_3 \neq P_2] \\
&\leqslant \Pr[P_1 \neq P_3] + \Pr[P_3 \neq P_2] \\
&= d_{\mathbb{D}}(P_1, P_3) + d_{\mathbb{D}}(P_3, P_2)
\end{aligned}$$

Thus if P is α-robust and $d_{\mathbb{D}}(P^*, P) = \Delta$, we know P' has $d_{\mathbb{D}}(P, P') \leqslant \varepsilon$ by

Theorem 4.6, and the triangle inequality gives us that

$$\begin{aligned}\mathrm{Er}(P') &= d_{\mathbb{D}}(\hat{P}, P') \\ &\leqslant d_{\mathbb{D}}(\hat{P}, P^*) + d_{\mathbb{D}}(P^*, P) + d_{\mathbb{D}}(P, P') \\ &\leqslant \mathrm{Er}(P^*) + \Delta + \varepsilon\end{aligned}$$

Proof of Theorem 4.15. Let S be the set of samples, with $|S| = m$. By Lemma 4.13, the number of queries is at most $|S||\Pi_{\mathcal{P}}(S)|$, which is in turn at most $m\hat{\Pi}_{\mathcal{P}}(m)$. Applying Lemma 4.14 immediately gives us the $O(m^{d+1})$ bound.

Proof of Theorem 4.24. By construction, PVC soundly converts the repair model into a logical formula over hole-input-output tuples consistent with the repair model; this is a standard encoding used in Sketch (Solar-Lezama, 2008), for example. Specifically, substituting concrete input-output pairs for the appropriate variables results in a formula where satisfying assignments are values of the holes that result in a program consistent with the samples. Therefore, the soundness and completeness of our $\mathcal{O}_{\text{syn}}$ implementation follows from the soundness and completeness of SMT solvers for existentially quantified LRA formulae.

C.1 Full Benchmarks for Antidote

Here we present figures analagous to Figure 2.6 for the remaining datasets in Table 2.1, which were otherwise ommited from Section 2.5. Figures C.1, C.2, C.3, and C.4 show these detailed performance metrics. (Note that Iris is is the only benchmark for which we do not use log-log plots, since the numbers are generally small.)

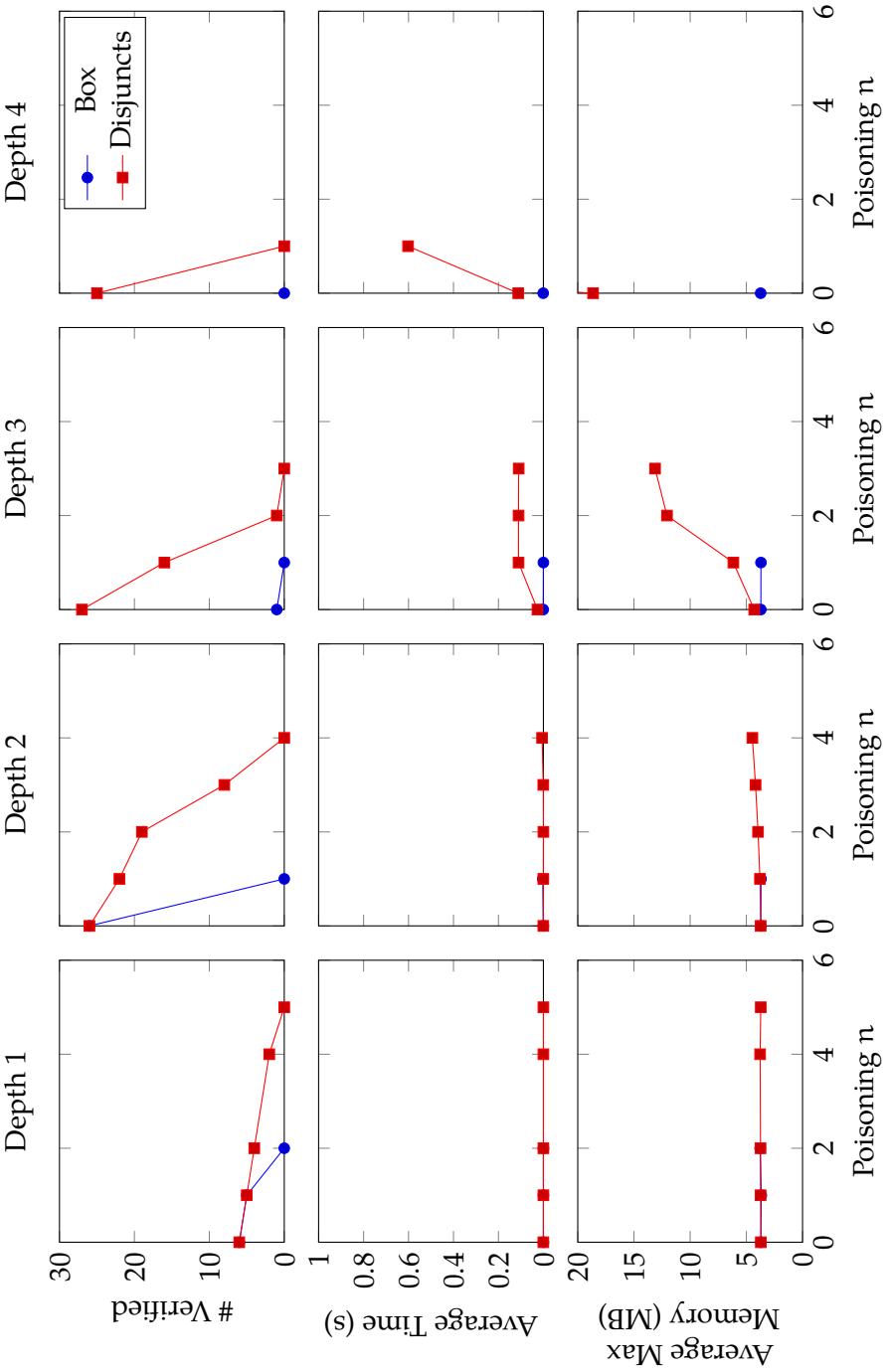


Figure C.1: Iris

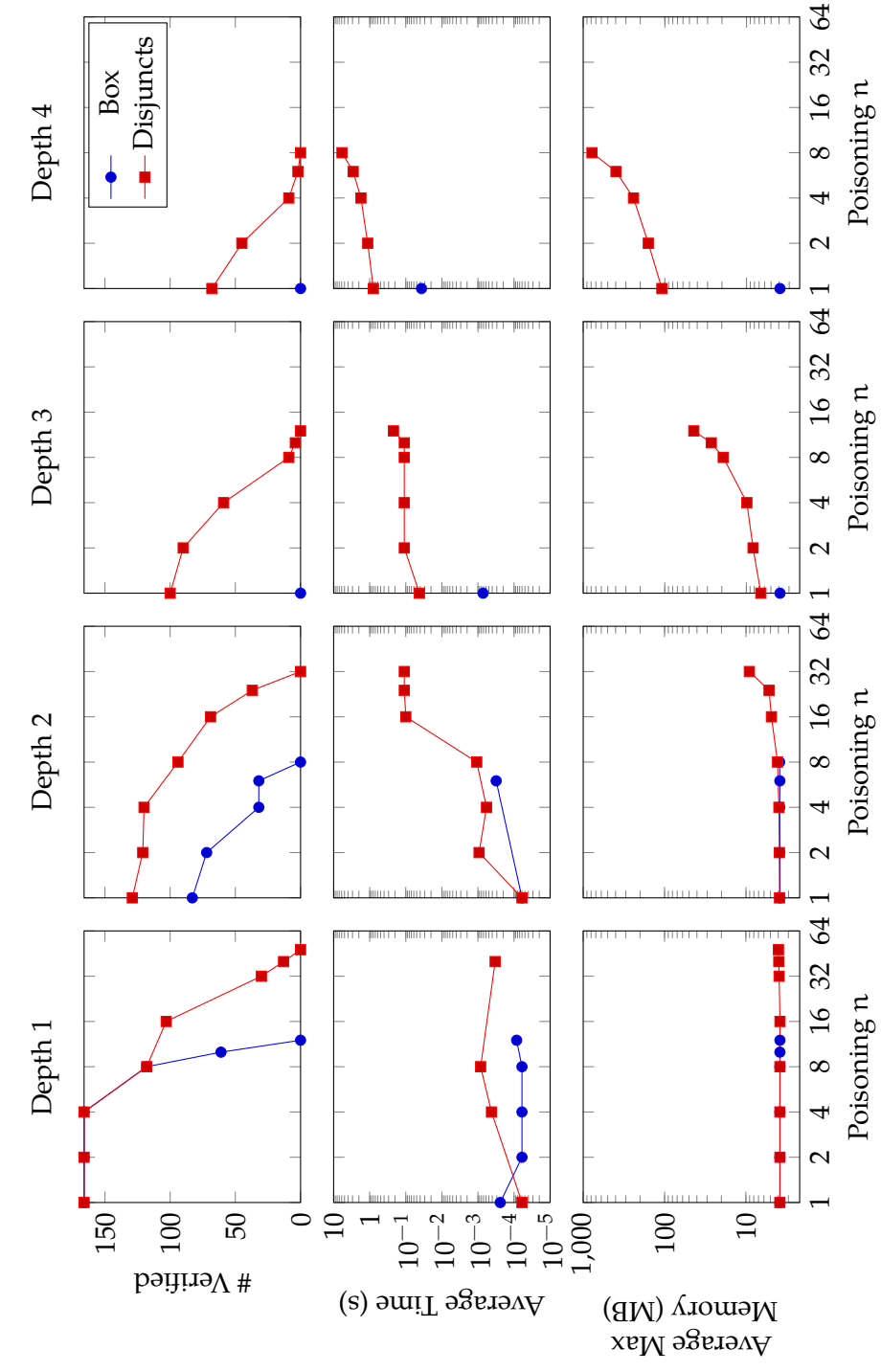


Figure C.2: Mammographic Masses

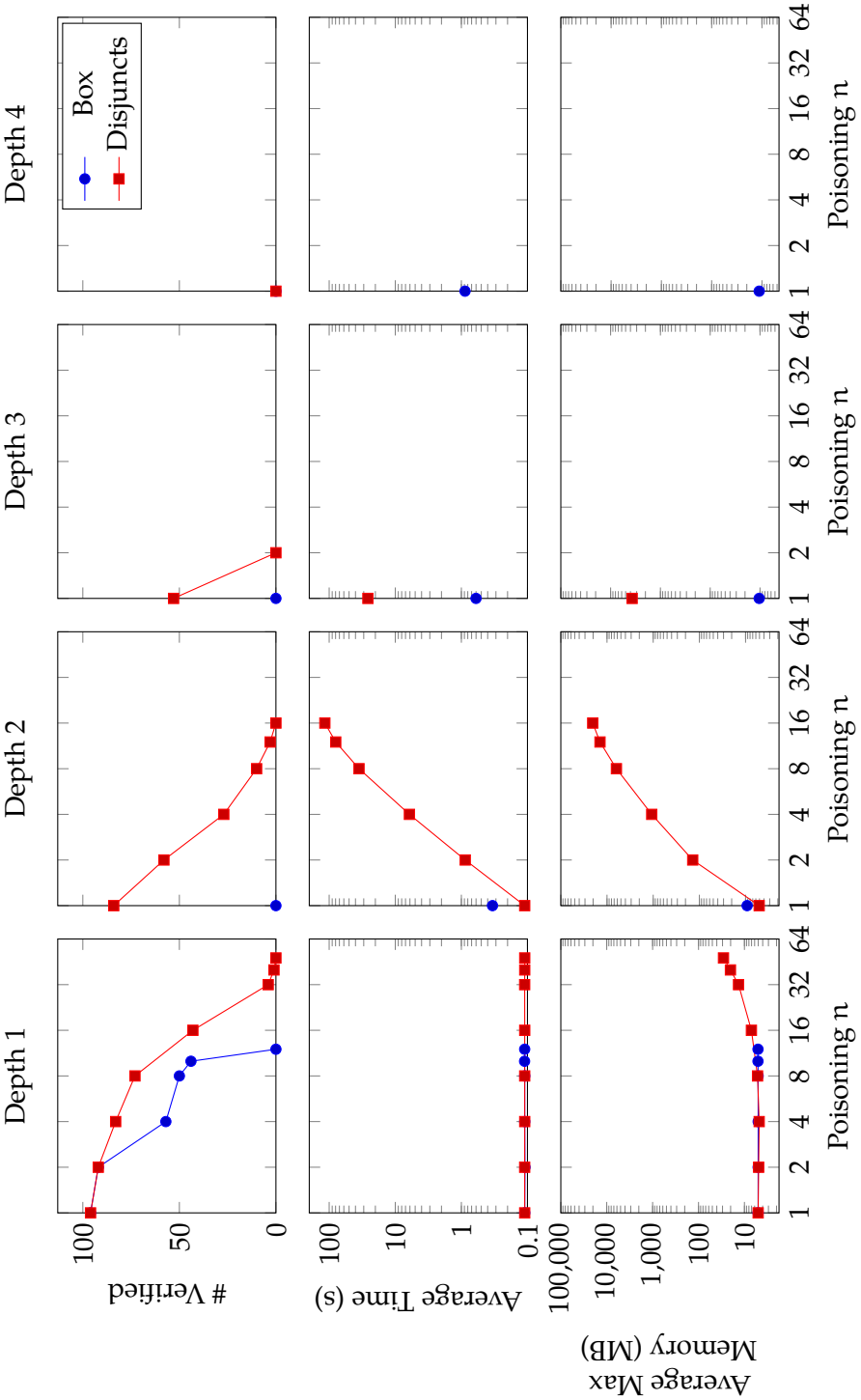


Figure C.3: Wisconsin Diagnostic Breast Cancer

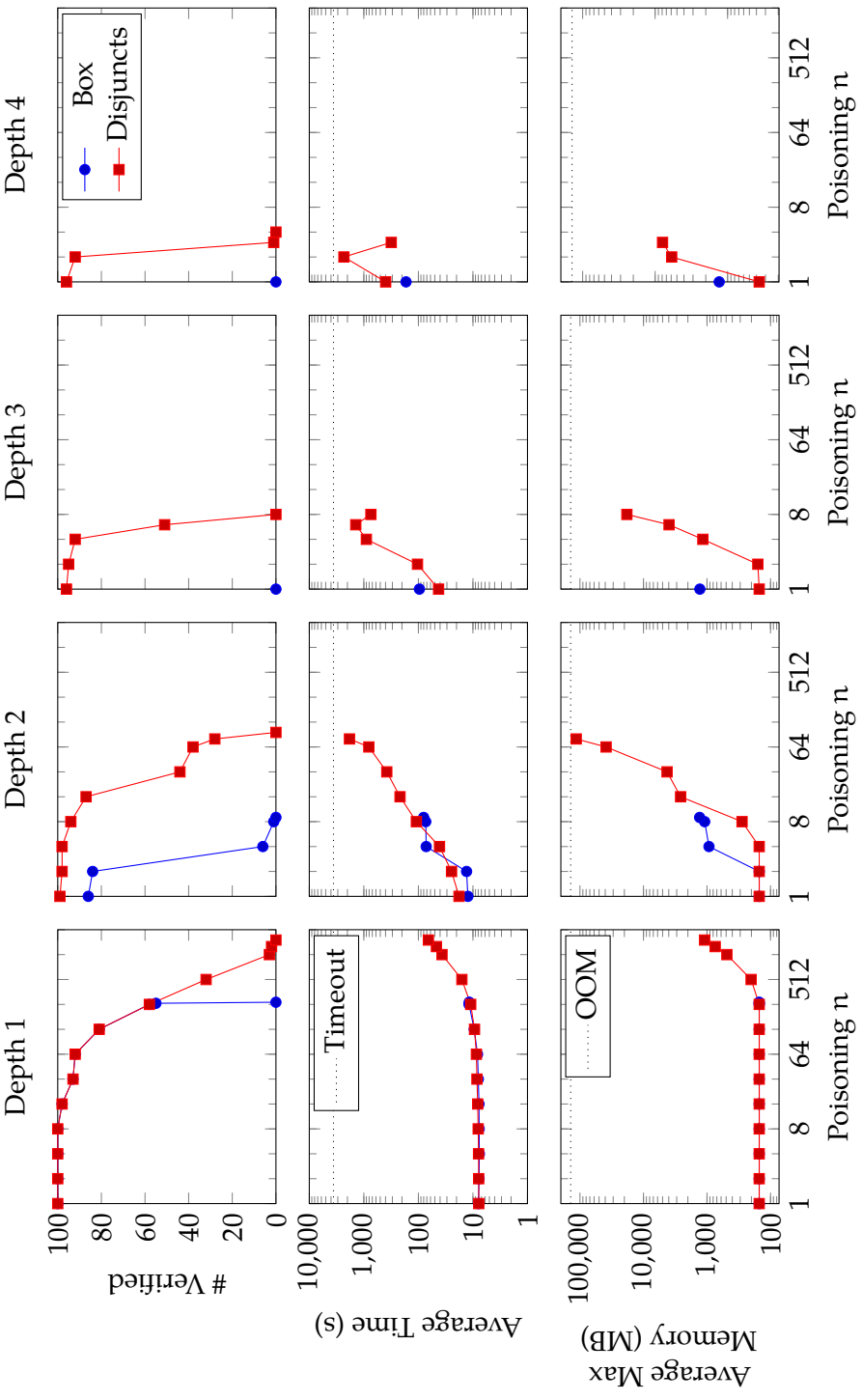


Figure C.4: MNIST-1-7-Real

C.2 Full Synthetic Benchmarks for DIGITS

In this section, we provide the complete description of the synthetic benchmarks used in Section 4.6 and present the complete plots of our evaluation.

We consider a class of hyperrectangle programs for which we can compute the optimal solution exactly; this lets us compare the results of our implementation to an ideal baseline. Here, the concept class $\mathcal{P}$ (i.e., the set of programs) is defined as the set of axis-aligned hyperrectangles within $[-1,1]^d$, and the input distribution $\mathbb{D}$ is such that inputs are distributed uniformly over $[-1,1]^d$. We fix some probability mass b and aim to synthesize a program that is close to a functional specification of the form $0 \leqslant x_1 \leqslant 2b \wedge \bigwedge_{i\in\{2,\ldots,d\}} -1 \leqslant x_i \leqslant 1$, which only returns 1 for points whose first coordinate is positive and at most $2b$. We fix the following postcondition:

$$\Pr[P(\mathbf{x}) = 1 \mid x_1 \leqslant 0] \geqslant \Pr[P(\mathbf{x}) = 1 \mid x_1 \geqslant 0] \wedge \Pr[P(\mathbf{x}) = 1] \geqslant b.$$

In other words, a correct hyperrectangle must include as much probability mass of points whose first coordinate is negative as it does for those with a positive first coordinate, and additionally it must include at least as much probability mass as the original hyperrectangle. Observe that independent of d, the best error for a correct solution is exactly b (and there exist dense regions of α-robust programs that have error $b + \alpha$).

We consider problem instances formed from combinations of $d \in \{1,2,3\}$ and $b \in \{0.05, 0.1, 0.2\}$. As d increases, the set of programs increases in complexity (in fact, it has VC dimension $2d$) and the synthesis queries become more expensive. As b increases, the threshold used by the optimization cannot be as small, so we expect the search to benefit less from our optimizations. We run our implementation using thresholds $\tau \in \{0.07, 0.15, 0.3, 0.5, 1\}$, omitting those values for which $\tau < b$; additionally, we also consider an adaptive run where τ is initialized as the value 1,

and whenever a new best solution is enumerated with error k we update $\tau \leftarrow k$.

Each combination of parameters was run for a period of 2 minutes. Figure C.5 shows each of the following as a function of time: (*i*) the depth completed by the search (i.e. the current size of the sample set), and (*ii*) the best solution found by the search.

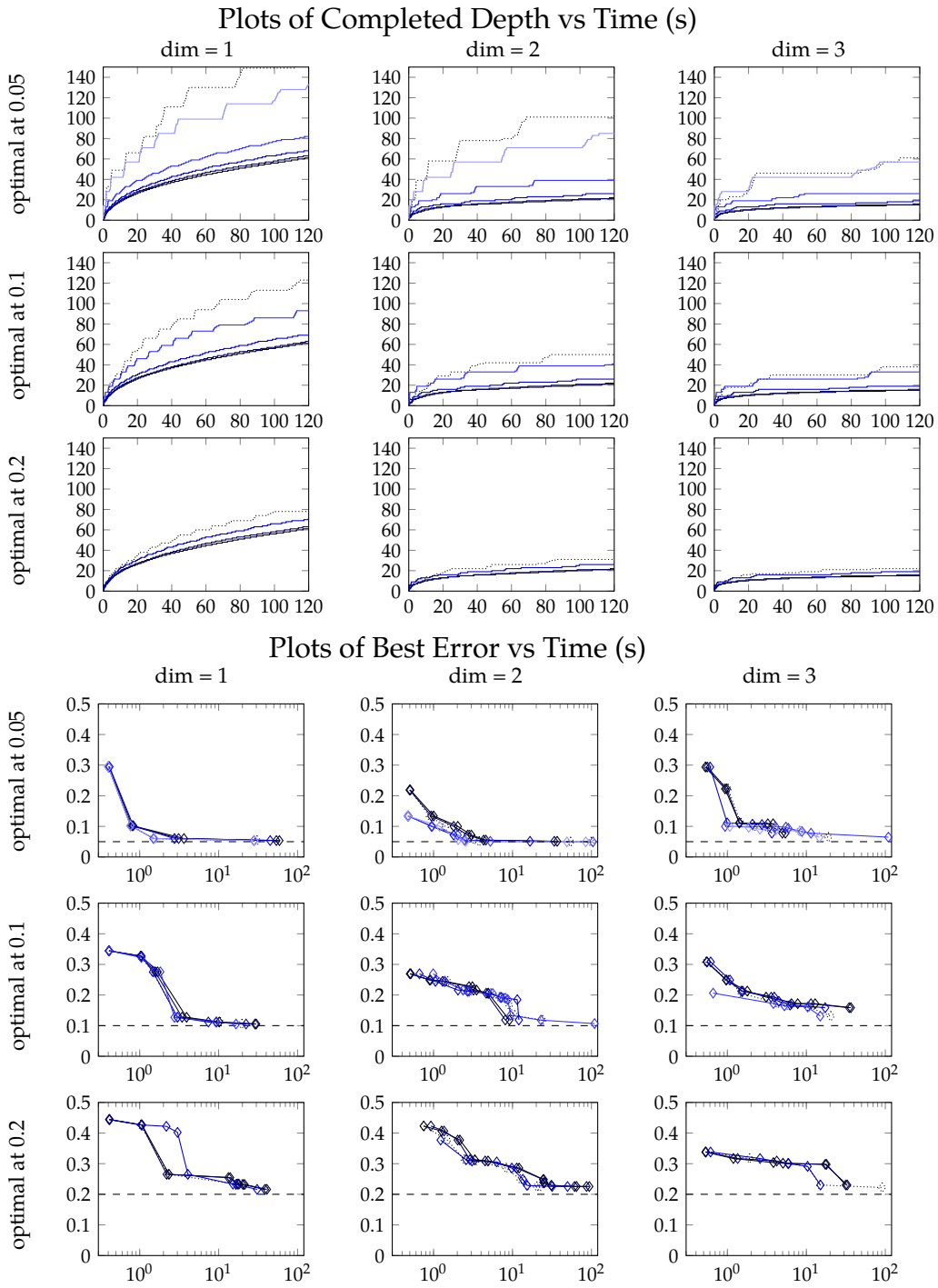


Figure C.5: Performance of τ-DIGITS with $\tau \in \{1, 0.5, 0.3, 0.15, 0.07\}$ on synthetic hyperrectangle examples with varying parameters