Verification as Learning Geometric Concepts

Rahul Sharma[1], Saurabh Gupta[2], Bharath Hariharan[2], Alex Aiken[1], and Aditya V. Nori[3]

[1] Stanford University, {sharmar,aiken}@cs.stanford.edu
[2] University of California at Berkeley, {sgupta,bharath2}@eecs.berkeley.edu
[3] Microsoft Research India, adityan@microsoft.com

Abstract. We formalize the problem of program verification as a learning problem, showing that invariants in program verification can be regarded as geometric concepts in machine learning. Safety properties define *bad states*: states a program should not reach. Program verification explains why a program's set of reachable states is disjoint from the set of bad states. In Hoare Logic, these explanations are predicates that form inductive assertions. Using samples for reachable and bad states and by applying well known machine learning algorithms for classification, we are able to generate inductive assertions. By relaxing the search for an exact proof to classifiers, we obtain complexity theoretic improvements. Further, we extend the learning algorithm to obtain a sound procedure that can generate proofs containing invariants that are arbitrary boolean combinations of polynomial inequalities. We have evaluated our approach on a number of challenging benchmarks and the results are promising.

Keywords: loop invariants, verification, machine learning

1 Introduction

We formalize the problem of verification as a learning problem, showing that loop invariants can be regarded as geometric concepts in machine learning. Informally, an invariant is a predicate that separates good and bad program states and once we have obtained strong invariants for all the loops, standard techniques can be used to generate program proofs. The motivation for using machine learning for invariant inference is twofold: guarantees and expressiveness.

Standard verification algorithms observe some small number of behaviors of the program under consideration and extrapolate this information to (hopefully) get a proof for all possible behaviors of the program. The extrapolation is a heuristic and systematic ways of performing extrapolation are unknown, except for the cases where they have been carefully designed for a particular class of programs. SLAM [6] generates new predicates from infeasible counterexample traces. Interpolant based techniques [37] extrapolate the information obtained from proving the correctness of finite unwindings of loops. In abstract interpretation [21], fixpoint iterations are performed for a few iterations of the loop and this information is extrapolated using a widening operator. In any of these heuristics, and others, there is no formal characterization of how well the output of the extrapolation strategy approximates the true invariants.

Extrapolation is the fundamental problem attacked by machine learning: A learning algorithm has some finite training data and the goal is to learn a function that generalizes for the infinite set of possible inputs. For classification, the learner is given some examples of good and bad states and the goal is to learn a predicate that separates all the good states from all the bad states. Unlike standard verification approaches that have no guarantees on extrapolation, learning theory provides formal generalization guarantees for learning algorithms. These guarantees are provided in learning models that assume certain oracles. However, it is well known in the machine learning community that extrapolation engines that have learning guarantees in the theoretical models tend to have good performance empirically. The algorithms have been applied in diverse areas such as finance, biology, and vision: we apply learning algorithms to the task of invariant inference.

Standard invariant generation techniques find invariants of a restricted form: there are restrictions on expressiveness that are not due to efficiency considerations but instead due to fundamental limitations. These techniques especially have trouble with disjunctions and non-linearities. Predicate abstraction restricts invariants to a boolean combination of a given set of predicates. Existing interpolation engines cannot generate non-linear predicates [1]. Template based approaches for linear invariants like [35] require a template that fixes the boolean form of the invariant and approaches for non-linear invariants [53] can only find conjunctions of polynomial equalities. Abstract interpretation over convex hulls [23] handles neither disjunctions nor non-linearities. Disjunctions can be obtained by performing disjunctive completion [22, 26], but widening [3] places an ad hoc restriction on the number of disjuncts. Our learning algorithm is strictly more expressive than these previous approaches: It can generate arbitrary boolean combinations of polynomial inequalities (of a given degree). Hence there are no restrictions on the number of disjuncts and we go beyond linear inequalities and polynomial equalities.

Unsurprisingly, our learning algorithm, with such expressive power, has high computational complexity. Next, we show how to trade expressiveness for computational speedups. We construct efficient machine learning algorithms, with formal generalization guarantees, for generating arbitrary boolean combinations of constituent predicates when these predicates come from a given set of predicates (predicate abstraction), when the size of integer constants in the predicates are bounded, or from a given abstract domain (such as boxes or octagons). Note that these efficient algorithms with reduced expressiveness still generate arbitrary boolean combinations of predicates.

Our main insight is to view invariants as geometric concepts separating good and bad states. This view allows us to make the following contributions:

- We show how to use a well known learning algorithm [13] for the purpose of computing candidate invariants. This algorithm is a PAC learner: it has generalization guarantees in the PAC (probably approximately correct) learning model. The learning algorithm makes no assumption about the syntax of the program and outputs a candidate invariant that is as expressive as arbitrary boolean combinations of linear inequalities.

- The algorithm of [13] is impractical. We parametrize the algorithm of [13] by the abstract domain in which the linear inequalities constituting the invariants lie, allowing us to obtain candidates that are arbitrary boolean combinations of linear inequalities belonging to the given abstract domain. We obtain efficient PAC learning algorithms for generating such candidates for abstract domains requiring few variables, such as boxes or octagons and finite domains such as predicate abstraction.
- We augment our learning algorithms with a theorem prover to obtain a sound procedure for computing invariants. This idea of combining procedures for generating likely invariants with verification engines has been previously explored in [49, 55, 54] (see Section 6). We evaluate the performance of this procedure on challenging benchmarks for invariant generation from the literature. We are able to generate invariants, using a small amount of data, in a few seconds per loop on these benchmarks.

The rest of the paper is organized as follows: We informally introduce our technique using an example in Section 2. We then describe necessary background material, including the learning algorithm of [13] (Section 3). Section 4 describes the main results of our work. We first give an efficient algorithm for obtaining likely invariants from candidate predicates (Section 4.1). Next, in Section 4.1, we obtain efficient algorithms for the case when the linear inequalities constituting the invariant lie in a given abstract domain. In Section 4.2, we extend [13] to generate candidates that are arbitrary boolean combinations of polynomial inequalities. Finally, Section 4.3 describes our sound procedure for generating invariants. Section 5 describes our implementation and experiments. We discuss related work in Section 6 and conclude in Section 7.

2 Overview of the Technique

```
1: x := i; y := j;
2: while (x != 0) { x--; y--; }
3: if (i == j) assert (y == 0);
```

Fig. 1. Motivating example.

Consider the program in Figure 1 [37]. To prove that the assertion in line 3 is never violated, we need to prove the following Hoare triple:

$$\{x = i \wedge y = j\}\texttt{while (x != 0) do x--; y--}\{i = j \Rightarrow y = 0\}$$

In general, to prove $\{P\}$ `while` E `do` S $\{Q\}$, where E is the loop condition and S is the loop body, we need to find a *loop invariant* I satisfying $P \Rightarrow I$, $\{I \wedge E\}S\{I\}$, and $I \wedge \neg E \Rightarrow Q$. Thus, to verify that the program in Figure 1 does not violate the assertion, we need a loop invariant I such that $(x = i \wedge y = j) \Rightarrow I$, $\{I \wedge x \neq 0\}S\{I\}$, and $I \wedge x = 0 \Rightarrow (i = j \Rightarrow y = 0)$. The predicate $I \equiv i = j \Rightarrow x = y$ is one such invariant [37].

There is another way to view loop invariants. For simplicity of exposition, we restrict our attention to *correct* programs that never violate assertions (e.g., Figure 1). A *state* is a valuation of the program variables, for example $(i, j, x, y) = (1, 0, 1, 0)$. Consider the set of states at the loop head (the `while` statement of Figure 1) when the program is executed. All such states are *good* states, that is, states that a correct program can reach. A *bad* state is one that would cause an assertion violation. For example, if we are in the state $(i, j, x, y) = (1, 1, 0, 1)$ at the loop head, then execution does not enter the loop and violates the assertion.

An invariant strong enough to prove the program correct is true for all good states and false for all bad states. Therefore, if one can compute the good states and the bad states, an invariant will be a predicate that *separates* the good states from the bad states. Of course, in general we cannot compute the set of all good states and the set of all bad states. But we can always compute some good and bad states by sampling the program.

To generate samples of good states, we simply run the program on some inputs. If we run the program in Figure 1 with the initial state $(1, 0, 1, 0)$, we obtain the good samples $(1, 0, 1, 0)$ and $(1, 0, 0, -1)$. To compute bad states, we can sample from predicates under-approximating the set of all bad states. For Figure 1, $(x = 0 \wedge i = j \wedge y \neq 0)$ is the set of bad states that do not enter the loop body and violate the assertion, and $(x = 1 \wedge i = j \wedge y \neq 1)$ is the set of bad states that execute the loop body once and then violate the assertion. Note that such predicates can be obtained from the program using a standard weakest precondition computation. Finally, we find a predicate separating the good and bad samples.

But how can we guarantee that a predicate separating the good samples from the bad samples also separates *all* good states from *all* bad states? In machine learning, formal guarantees are obtained by showing that the algorithm generating these predicates *learns* in some learning model. There are several learning models and in this paper we use Valiant's PAC (probably approximately correct) model [56]. An algorithm that learns in the PAC model has the guarantee that if it is given enough independent samples then with a high probability it will come up with a predicate that will separate almost all the good states from the bad states. Hence, under the assumptions of the PAC model, we are guaranteed to find good candidate invariants with high probability. However, just like any other theoretical learning model, the assumptions of PAC model are generally impossible or at least very difficult to realize in practice. We emphasize that in the variety of applications in which PAC learning algorithms are applied, the assumptions of the PAC model are seldom met. Hence, the question whether generalization guarantees in a learning model are relevant in practice is an empirical one. PAC has intimate connections with complexity theory and cryptography and is one of the most widely used models. We demonstrate empirically in Section 5 that PAC learning algorithms successfully infer invariants.

Bshouty et al. [13] presented a PAC learning algorithm for geometric concepts (see Section 3.2). This algorithm can produce predicates as expressive as arbitrary boolean combinations of linear inequalities. In particular, the invariant required for Figure 1 is expressible using this approach. However, this expressiveness has a cost: the algorithm of [13] is exponential in the number of program variables. To obtain polynomial time algorithms in the number of samples and program variables we must restrict the expressiveness. Assume, for example, that we knew the invariant for the program in Figure 1 is a boolean combination of octagons (which it is). For octagons, the linear inequalities are of the form $\pm x \pm y \leq c$, where x and y are program variables and c is a constant (Section 4.1). We extend [13] to obtain a PAC learning algorithm for obtaining a predicate, separating good and bad samples, that is an arbitrary boolean combination of linear inequalities belonging to a given abstract domain. The time complexity of our algorithm increases gracefully with the expressiveness of the chosen abstract domain (Section 4.1). For example, the complexity for octagons is higher than that for boxes.

We augment our learning algorithm with a theorem prover (Section 4.3), obtaining a sound algorithm for program verification. Empirically, we show that the predicates discovered by our approach are provably invariants using standard verification engines (Section 5).

2.1 Finding Invariants for the Example

We now explain how our sound algorithm (Section 4.3) for program verification (parametrized by octagons) proves the correctness of the program in Figure 1. To sample the good states, assume we run the program on inputs where $i, j \in \{0, 1, 2\}$. As suggested above, we obtain bad states by sampling the predicate representing violations of the assertion after going through at most one loop iteration: $x = 0 \wedge i = j \wedge y \neq 0 \vee x = 1 \wedge i = j \wedge y \neq 1$. In total, for this example, we generated 18 good samples and 24 bad samples. The algorithm of [13] first generates a large set of candidate hyperplanes representing all linear inequalities possibly occurring in the output predicate. We build this set by constructing all possible hyperplanes (of the form $\pm x \pm y = c$) passing through every state. For instance, the state $(2, 2, 0, 0)$ generates twenty four hyperplanes: $x = 0$, $x = y$, $i \pm x = 2, \ldots$. Section 4.1 justifies this choice of the set of candidates.

From this large set of hyperplanes, we pick a subset that successfully separates the good and bad samples. Note that every good sample must be separated from every bad sample. Several algorithms can be used to solve this problem. We describe how a standard greedy approach would work. We keep track of the pairs of samples, one good and the other bad, that have not yet been separated by any hyperplane, and repeatedly select from the set of candidate hyperplanes the one that separates the maximum number of remaining unseparated pairs, repeating until no unseparated pairs remain.

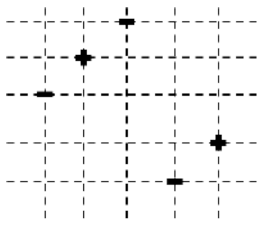
Fig. 2. Candidate inequalities passing through all states.

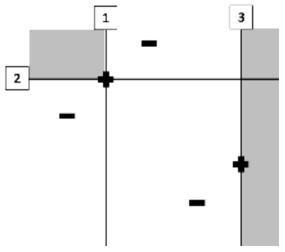

Fig. 3. Separating good states and bad states using boxes.

We illustrate this process in Figures 2 and 3. The $+$'s are the good states, and the $-$'s are the bad states. Assume that our abstract domain is the box or interval domain, that is, the predicates are inequalities of the form $\pm x \leq c$. We first generate our candidates, that is, hyperplanes of the form $x = c$ passing through all the good and bad states. These corresponds to all possible horizontal and vertical lines passing through all the $+$ and $-$ states as shown in Figure 2. Next, from this set of candidate lines, we initially select line 3, separating one good state from three bad states, which is the maximum number of pairs separated by any of the lines. Next, we select line 1 because it separates one good state from two bad states. Finally, we select line 2, separating the final pair of one good state and one bad state. The lines tesselate the space into cells, where each cell is a conjunction of boxes bounding the cell and no cell contains both a good and a bad state. Each shaded cell in Figure 3 represents a conjunction of boxes that includes only the good states. The returned predicate is the set of all shaded cells in Figure 3, which is a disjunction of boxes.

By a similar process, for the 42 states generated from Figure 1 and using the octagon domain, our tool infers the predicate $I \equiv i \leq j + 1 \vee j \leq i + 1 \vee x = y$ in 0.06 seconds. We annotated the loop of Figure 1 with this predicate as a candidate loop invariant and gave it to the Boogie [7] program checker. Boogie was successfully able to prove that I was indeed a loop invariant and was able to show that the assertion holds. As another example, on parametrizing with the Octahedron [16] abstract domain, our technique discovers the simpler conjunctive loop invariant: $i + y = x + j$ in $0.09s$.

3 Preliminaries

This section presents necessary background material, including the learning algorithm of [13]. Our goal is to verify a *Hoare triple* $\{P\}\mathcal{S}\{Q\}$ for the simple language of *while programs* defined as follows:

$$\mathcal{S} ::= \; x{:=}M \mid \mathcal{S};\mathcal{S} \mid \texttt{if } E \texttt{ then } \mathcal{S} \texttt{ else } \mathcal{S} \texttt{ fi} \mid \texttt{while } E \texttt{ do } \mathcal{S}$$

The while program $\mathcal{S}$ is defined over integer variables, and we want to check whether, for all states s in the precondition P, executing $\mathcal{S}$ with initial state s results in a state satisfying the postcondition Q. In particular, if $L \equiv \texttt{while}\ E\ \texttt{do}\ S$ is a while program, then to check $\{P\}L\{Q\}$, Hoare logic tells us that we need a predicate I such that $P \Rightarrow I$, $\{I \wedge E\}S\{I\}$, and $I \wedge \neg E \Rightarrow Q$. Such a predicate I is called an *inductive invariant* or simply an *invariant* of the loop L. Once we have obtained invariants for all the loops, then standard techniques can generate program proofs [7]. We first focus our attention on invariants in the theory of *linear arithmetic*:

$$\phi ::= w^T x + d \geq 0 \mid \mathit{true} \mid \mathit{false} \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg \phi$$

where $w = (w_1, \ldots, w_n)^T \in \mathbb{Q}^n$ is a *point*, an n-dimensional vector of rational number constants. The vector $x = (x_1, \ldots, x_n)^T$ is an n-dimensional vector of variables. The *inner product* $\langle w, x \rangle$ of w and x is $w^T x = w_1 x_1 + \ldots + w_n x_n$. The equation $w^T x + d = 0$ is a *hyperplane* in n dimensions with *slope* w and *bias* d. Each hyperplane *corresponds* to an intersection of two *half-spaces*: $w^T x + d \geq 0$ and $w^T x + d \leq 0$. For instance, $x - y = 0$ is a 2-dimensional hyperplane, $x - y + 2z = 0$ is a 3-dimensional hyperplane, and $x \geq y$ and $x \leq y$ are half-spaces corresponding to the hyperplane $x = y$.

3.1 Invariants and Binary Classification

Assume that the Hoare triple $\{P\}\texttt{while}\ E\ \texttt{do}\ S\{Q\}$ is valid. Let the loop L have n variables $x = \{x_1, \ldots, x_n\}$. Therefore, the precondition $P(x)$ and postcondition $Q(x)$ are predicates over x. If the loop execution is started in a state satisfying P and control flow reaches the loop head after zero or more iterations, then the resulting state is said be *reachable* at the loop head. Denote the set of all reachable states at the loop head by $\mathcal{R}$. Since the Hoare triple is valid, all the reachable states are *good* states. On the other hand, if we execute the loop from a state y satisfying $\neg E \wedge \neg Q$, then we will reach a state at the end of the loop that violates the postcondition, that is, y satisfies $\neg Q$. We call such a state a *bad state*. Denote the set of all bad states by $\mathcal{B}$. Observe that for a correct program, $\mathcal{R} \Rightarrow \neg \mathcal{B}$. Otherwise, any state satisfying $\mathcal{R} \wedge \mathcal{B}$ is a reachable bad state. $\mathcal{R}$ is the strongest invariant, while $\neg \mathcal{B}$ is the weakest invariant sufficient to prove the Hoare triple. Any inductive predicate $\mathcal{I}$ satisfying $\mathcal{R} \Rightarrow \mathcal{I}$ and $\mathcal{I} \Rightarrow \neg \mathcal{B}$ suffices for the proof: $\mathcal{I}$ contains all the good states and does not contain any bad state. Therefore, $\mathcal{I}$ *separates* the good states from the bad states, and thus the problem of computing an invariant can be formulated as finding a separator between $\mathcal{R}$ and $\mathcal{B}$. In general, we do not know $\mathcal{R}$ and $\mathcal{B}$ – our objective is to compute a separator $\mathcal{I}$ from under-approximations of $\mathcal{R}$ and $\mathcal{B}$. For the Hoare triple $\{P\}\texttt{while}\ E\ \texttt{do}\ S\{Q\}$, any subset of states reachable from P is an under-approximation of $\mathcal{R}$, while any subset of states satisfying, but not limited to, the predicate $\neg E \wedge \neg Q$ is an under-approximation of $\mathcal{B}$.

Computing separators between sets of points is a well-studied problem in machine learning and goes under the name *binary classification*. The input to the binary classification problem is a set of points with labels from $\{1, 0\}$. Given points and their labels, the goal of the binary classification is to find a *classifier*

$C : \mathit{points} \to \{\mathit{true}, \mathit{false}\}$, such that $C(a) = \mathit{true}$, for every point a with label 1, and $C(b) = \mathit{false}$ for every point b with label 0. This process is called *training* a classifier, and the set of labeled points is called the *training data*.

The goal of classification is not to just classify the training points correctly but also to be able to predict the labels of previously unseen points. In particular, even if we are given a new labeled point w, with label l, not contained in the training data, then it should be very likely that $C(w)$ is *true* if and only if $l = 1$. This property is called *generalization*, and an algorithm that computes classifiers that are likely to perform well on unseen points is said to *generalize* well.

If C lies in linear arithmetic, that is, it is an arbitrary boolean combination of half-spaces, then we call such a C a *geometric concept*. Our goal is to apply machine learning algorithms for learning geometric concepts to obtain invariants. The good states, obtained by sampling from $\mathcal{R}$, will be labeled 1 and the bad states, obtained by sampling from $\mathcal{B}$, will be labeled 0. We want to use these labeled points to train a classifier that is likely to be an invariant, separating all the good states $\mathcal{R}$ from all the bad states $\mathcal{B}$. In other words, we would like to compute a classifier that generalizes well enough to be an invariant.

3.2 Learning Geometric Concepts

Let R and B be under-approximations of the good states $\mathcal{R}$ and the bad states $\mathcal{B}$, respectively, at a loop head. The classifier $\vee_{r \in R} x = r$ trivially separates R from B. However, this classifier has a large generalization error. In particular, it will *misclassify* every state in $\mathcal{R} \setminus R$; a candidate invariant misclassifies a good state r when $I(r) = \mathit{false}$ and a bad state b when $I(b) = \mathit{true}$. It can be shown if a predicate or classifier grows linearly with the size of training data ($\vee_{r \in R} x = r$ being such a predicate), then such a classifier cannot generalize well. On the other hand, a predicate that is independent of the size of training data can be proven to generalize well [11].

To reduce the size of the predicates, Bshouty et al. [13] frame the problem of learning a general geometric concept as a *set cover* problem. Let X be a set of n points. We are given a set $F \subseteq 2^X$ with k elements such that each element $F_i \in F$ is a subset of X. We say that an element $x \in X$ is *covered* by the set F_i if $x \in F_i$. The goal is to select the minimum number of sets F_i such that each element of X is covered by at least one set. For example, if $X = \{1, 2, 3\}$ and $F = \{\{1, 2\}, \{2, 3\}, \{1, 3\}\}$, then $\{\{1, 2\}, \{2, 3\}\}$ is a solution, and this minimum set cover has a size of two. The set cover problem is NP-complete and we have to be satisfied with approximation algorithms [12, 15]. Bshouty et al. [13] formalize learning of geometric concepts as a set cover problem, solve it using [12], and show that the resulting algorithm PAC learns. Note that experiments of [12] show that the performance of the naive greedy algorithm [15] is similar to the algorithm of [12] in practice. Hence, we use the simple to implement greedy set cover for our implementation (Section 5).

We are given a set of samples $V = \{x_i\}_{i=1,\ldots m}$, some of which are good and some bad. We create a bipartite graph $\mathcal{U}$ where each sample is a node and there is an edge between nodes x_+ and x_- for every good sample x_+ and every bad

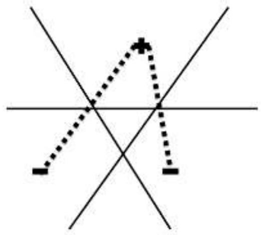

Fig. 4. Separating three points in two dimensions. The solid lines tessellate $\mathbb{R}^2$ into seven cells. The $-$'s are the bad states and the $+$'s are the good states. The dotted lines are the edges to be cut.

sample x_-. In Figure 4, there is one good state, two bad states, and dotted lines represent edges of $\mathcal{U}$. Next, we look for hyperplanes that cut the edges of the graph $\mathcal{U}$. A hyperplane cuts an edge if the two endpoints of the edge lie on different sides of the hyperplane. Note that for every solution, each good sample needs to be separated from every bad sample. This implies that we will need to "cut" every edge in graph $\mathcal{U}$. Intuitively, once we have collected a set S of hyperplanes such that every edge in graph $\mathcal{U}$ is cut by at least one hyperplane in S we can perfectly separate the good and bad samples. The hyperplanes in S tessellate $\mathbb{R}^d$ into a number of cells. (In Figure 4, the three solid lines tessellate $\mathbb{R}^2$ into seven cells.) No cell contains both a good sample and a bad sample – if it does, then the edge between a good sample and a bad sample in the cell is not cut by any hyperplane in S. Thus, each cell contains only good samples, or only bad samples, or no samples at all. We can therefore label each cell, as "good" in the first case, "bad" in the second case, and with an arbitrary "don't care" label in the last case.

Each cell is bounded by a set of hyperplanes, and therefore corresponds to an intersection of half-spaces. The "good" region of $\mathbb{R}^d$ (where d is the number of variables in the program) is then a union of cells labeled "good", and hence a union of intersections of half-spaces, that we output. Thus, the union of intersections of half-spaces we output contains all the good samples, no bad samples, and separates all the good from all the bad samples.

This discussion shows that all we need to do is to come up with the set S of hyperplanes that together cut every edge of graph $\mathcal{U}$. To achieve this goal, we consider a universal set of hyperplanes $\mathcal{F}$ corresponding to all possible partitions of states. Every hyperplane defines a partition of states: some states lie above the plane and some lie below it. $\mathcal{F}$ contains one hyperplane for every possible partition. By Sauer's lemma, such a set $\mathcal{F}$ has cardinality $\mathcal{O}(m^d)$ [13]. We say that an edge is covered by a hyperplane from $\mathcal{F}$ if the hyperplane cuts it. We want to cover all edges of graph $\mathcal{U}$ by these hyperplanes. This set cover problem can be solved in several ways that have comparable performance in practice [15, 12]. The simplest solution is to greedily select the hyperplane from $\mathcal{F}$ that covers the maximum number of uncovered edges of graph $\mathcal{U}$, and repeating the greedy selection until all edges in $\mathcal{U}$ are cut. For Figure 4, $\mathcal{F}$ contains three hyperplanes, and graph $\mathcal{U}$ has two edges (edges between $-$'s and $+$'s.). The horizontal plane cuts both the edges and divides the space into two cells: one above and one below. Since the cell above the horizontal plane contains a '$+$', we will label it "good". Similarly, the cell below is labeled "bad". The output predicate is the half-space above the horizontal hyperplane. If the good and bad samples, total m in number, require a minimum number of s hyperplanes to separate them, then the greedy approach has the guarantee that will compute a predicate that uses $\mathcal{O}(s \log m)$ hyperplanes. Using [12], we can obtain a predicate using $\mathcal{O}(sd \log sd)$ hyperplanes. This implies that the number of inequalities of the classifier approximates the number of the inequalities of the simplest true invariant by a logarithmic factor. Such a relationship between candidate and true invariants appears to be new in the context of invariant inference.

3.3 PAC Learning

By enumerating a plane for each partition and performing a set cover, the algorithm of [13] finds a geometric concept that separates the good samples from the bad samples. But how well does it generalize? Bshouty et al. [13] showed that under the assumptions of the PAC model [56] this process is likely to produce a geometric concept that will separate all the good states from all the bad states with high probability. The major assumption of the PAC model is that there is an oracle that knows the true classifier and it generates training data by drawing independent and identically distributed samples from a distribution and assigning them labels, either good or bad, using the true classifier.

Independent samples are theoretically justified as otherwise one can construct data with an arbitrary number of samples by duplicating one sample an arbitrary number of times and then the term "amount of training data" is not well defined. Practically, if one draws a sample randomly from some distribution, then deciding whether it is good or bad is undecidable. Hence such an oracle cannot be implemented and in our experiments we make do with a simple technique for obtaining samples, where the samples are not necessarily independent.

The proof of PAC learning in [13] uses the following result from the seminal paper of Blumer et al. [11].

Theorem 1. *If an algorithm outputs f consistent with a sample of size max $\left(\frac{4}{\epsilon}\log\frac{2}{\delta}, \frac{8VC}{\epsilon}\log\frac{13}{\epsilon}\right)$ then f has error at most ϵ with probability at least $1-\delta$.*

Intuitively, this theorem states that if an algorithm can separate a large number of good and bad samples then the classifier has a low probability of misclassifying a new sample. Here VC is the Vapnik-Chervonenkis dimension, a quantity determined by the number of hyperplanes in the geometric concepts we are learning and the number of variables. In [13], by using algorithms for set cover that have a good approximation factor [12], Bshouty et al. are able to bound the number of planes in the output predicate f, and hence the quantity VC. Since the output of [13] is consistent with all good and bad samples, given enough samples the algorithm outputs a predicate that is very likely to separate all the good states from all the bad states. For the full proof the reader is referred to [13].

Hence, [13] can produce predicates that are likely to separate all good states and bad states, under PAC assumptions. This is a formal guarantee on the extrapolation we have performed using some good and bad samples, that is, using some finite behaviors of the program. Although this guarantee is in a model, we are unaware of any previous program verification engine with any guarantee, in a model or otherwise, on the heuristic extrapolation they perform. Even though this guarantee is not the best possible guarantee that one would desire, the undecidability of program verification prevents strong results for the problem we consider. It is well known that the PAC learners tend to have good performance in practice for a variety of learning tasks. Our experiments show that the PAC learners we construct have good performance for the task of invariant inference. We believe that by finding candidate invariants separating all good samples from all bad samples and misclassifying unseen points with low probability leads our technique to produce true invariants.

3.4 Complexity

If we have m states in d dimensions, then we need to cover $\mathcal{O}(m^2)$ edges of graph $\mathcal{U}$ using $\mathcal{O}(m^d)$ hyperplanes of $\mathcal{F}$. Greedy set cover has a time complexity of $\mathcal{O}(m^2|\mathcal{F}|)$. Considering $\mathcal{O}(m^d)$ hyperplanes is, however, impractical. With a thousand samples for a four variable program, we will need to enumerate 10^{12} planes. Hence this algorithm has a very high space complexity and will run out of memory on most benchmarks of Section 5.

Suppose the invariant has s hyperplanes. Hence the good states and bad states can be separated by s hyperplanes. To achieve learning, we require that $\mathcal{F}$ should contain s hyperplanes that separate the good samples and the bad samples – since the planes constituting the invariant could be any arbitrary set, in general we need to select a lot of candidates to ensure this. By adding assumptions about the invariant, the size of $\mathcal{F}$ can be reduced. Say for octagons, for thousand samples and four variables, the algorithm of Section 4.1 considers 24000 candidates.

3.5 Logic Minimization

The output of the algorithm of Section 3.2 is a set S of hyperplanes separating every good sample from every bad sample. As described previously, these hyperplanes tessellate $\mathbb{R}^d$ into cells. Recall that S has the property that no cell contains both a good state and a bad state.

Now we must construct a predicate containing all good samples and excluding all bad samples. One obvious option is the union of cells labeled "good". But this might result in a huge predicate since each cell is an intersection of half-spaces. Our goal is to compute a predicate with the smallest number of boolean operators such that it contains all the "good" cells and no "bad" cells. Let $\mathcal{H}$ be the set of half-spaces constituting the "good" cells. Define a boolean matrix M with m rows and $|\mathcal{H}|$ columns, and an m-dimensional vector y as follows.

$$M(i,j) = \mathit{true} \Leftrightarrow \{i^{th} \mathit{state} \in j^{th} \text{ half-space of } \mathcal{H}\}$$
$$y(i) = \mathit{true} \Leftrightarrow \{i^{th} \mathit{state} \text{ is a good state}\}$$

This matrix M together with the vector y resembles a partial truth table – the i^{th} row of M identifies the cell in which the i^{th} state lies and $y(i)$ (the label of the i^{th} state) gives the label for the cell (whether it is a cell containing only good states or only bad states). Now, we want to learn the simplest boolean function (in terms of the number of boolean operators) $f : \{true, false\}^{|\mathcal{H}|} \rightarrow \{true, false\}$, such that $f(M_i) = y(i)$ (M_i is the i^{th} row of M). This problem is called logic minimization and is NP-complete. Empirically, however, S has a small number of hyperplanes, at most eight in our experiments, and we are able to use standard exponential time algorithms like the Quine-McCluskey algorithm [44] to get a small classifier.

In summary, we use set covering for learning geometric concepts (Section 3.2) to compute predicates with a small number of hyperplanes. Combining this with logic minimization, we compute a predicate with a small number of boolean connectives. Empirically, we find that these predicates are actual invariants for *all* the benchmarks that have an arbitrary boolean combination of linear inequalities as an invariant.

4 Practical Algorithms

The algorithm discussed in Section 3.2, although of considerable interest, has limited practical applicability because its space and time complexity is exponential in the dimension, which in our case, is the number of program variables (Section 3.4). This complexity is not too surprising since, for example, abstract interpretation over the abstract domain of convex hulls [23] is also exponential in the number of variables. In this paper, we make the common assumption that the invariants come from a restricted class, which amounts to reducing the number of candidate sets for covering in our set cover algorithm. Therefore, we are able to obtain polynomial time algorithms in the number of samples and the dimension to generate classifiers under mild restrictions (Section 4.1).

4.1 Restricting Generality

Let s denote the number of hyperplanes in the invariant. Then for PAC learning, we say the set $\mathcal{F}$ of candidate hyperplanes is *adequate* if it contains s hyperplanes that completely separate the good samples from the bad samples. Recall that the complexity of the procedure of Section 3.2 is $\mathcal{O}(m^2|\mathcal{F}|)$, and therefore a polynomial size set $\mathcal{F}$ makes the algorithm polynomial time. In addition, the set covering step can be parallelized for efficiency [8].

In the following two sections we will give two PAC learning algorithms. The formal proofs that these algorithms learn in the PAC model are beyond the scope of this paper and are similar to the proofs in [13]. However, we do show the construction of adequate sets $\mathcal{F}$ that coupled with a good approximation factor of set cover [12] give us PAC learning guarantees.

Predicate Abstraction Suppose we are given a set of predicates $\mathcal{P}$ where each predicate is a half-space. Assume that the invariant is a boolean combination of predicates in $\mathcal{P}$, and checking whether a given candidate I is an invariant is co-NP-complete. If the invariant is an intersection or disjunction of predicates in $\mathcal{P}$, then Houdini [27] can find the invariant in time P^{NP} (that is, it makes a polynomial number of calls to an oracle that can solve NP problems). When the predicates are arbitrary boolean combinations of half-spaces from $\mathcal{P}$, then the problem of finding the invariant is much harder, NP^{NP}-complete [39]. We are not aware of any previous approach that solves this problem.

Now suppose that instead of an exact invariant, we want to find a PAC classifier to separate the good states from the bad states. If the set of candidates $\mathcal{F}$ is $\mathcal{P}$, then this trivially guarantees that there are s hyperplanes in $\mathcal{F}$ that do separate all the good states from the bad states – all we need to do now to obtain a PAC algorithm is to solve a set cover problem [12]. This observation allows us to obtain a practical algorithm. By using the greedy algorithm on m samples, we can find a classifier in time $\mathcal{O}(m^2|\mathcal{P}|)$. Therefore, by relaxing our problem to finding a classifier that separates good samples from bad samples, rather than finding an exact invariant, we are able to solve a NP^{NP} complete problem in time $\mathcal{O}(m^2|\mathcal{P}|)$ time, a very significant improvement in time complexity.

Abstract Interpretation Simple predicate abstraction can be restrictive because the set of predicates is fixed and finite. Abstract interpretation is another approach to finding invariants that can deal with infinite sets of predicates. For scalable analyses, abstract interpretation assumes that invariants come from restricted abstract domains. Two of the most common abstract domains are boxes and octagons. In boxes, the predicates are of the form $\pm x + c \geq 0$, where x is a program variable and c is a constant. In octagons, the predicates are of the form $\pm x \pm y + c \geq 0$. Note that, by varying c, these form an infinite family of predicates. These restricted abstract domains amount to fixing the set of possible slopes w of the constituent half-spaces $w^T x + b \geq 0$ (the bias b that corresponds to c, is however free).

Suppose now that we are given a finite set of slopes, that is, we are given a finite set of weight vectors $\Sigma = \{w_i \mid i = 1, \ldots, |\Sigma|\}$, such that the invariant only involves hyperplanes with these slopes. In this case, we observe that we can restrict our attention to hyperplanes that pass through one of the samples, because any hyperplane in the invariant that does not pass through any sample can be translated until it passes through one of the samples and the resulting predicate will still separate all the good samples from the bad samples. In this case, the set $\mathcal{F}$ is defined as follows:

$$\mathcal{F} = \{(w, b) \mid w \in \Sigma \text{ and } w^T x_i + b = 0 \text{ for some sample } x_i \in V\} \qquad (1)$$

The size of $\mathcal{F}$ is $|\Sigma|m$. Again, this set contains s hyperplanes that separate all the good samples from all the bad samples (the s hyperplanes of the invariant, translated to where they pass through one of the samples), and therefore this set is adequate and coupled with set covering [12] gives us a PAC learning algorithm.

The time complexity for greedy set cover in this case also includes the time taken to compute the bias for each hyperplane in $\mathcal{F}$. There are $|\mathcal{F}| = |\Sigma|m$ such hyperplanes, and finding the bias for each hyperplane takes $\mathcal{O}(d)$ time. The time complexity is therefore $\mathcal{O}(m^2|\mathcal{F}| + d|\mathcal{F}|) = \mathcal{O}(m^3|\Sigma|)$.

If we want to find classifiers over abstract domains such as boxes and octagons, then we can work with the appropriate slopes. For boxes $|\Sigma|$ is $\mathcal{O}(d)$ and for octagons $|\Sigma|$ is $\mathcal{O}(d^2)$. Interestingly, the increase in complexity when learning classifiers as we go from boxes to octagons mirrors the increase in complexity of the abstract interpretation. By adding more slopes we can move to more expressive abstract domains. Also note that the abstract domain over which we compute classifiers is much richer than the corresponding abstract interpretation. Conventional efficient abstract interpretation can only find invariants that are conjunctions of predicates, but we learn arbitrary boolean combinations of half-spaces, that allows us to learn arbitrary boolean combinations of predicates in abstract domains.

Again, we observe that by relaxing the requirement from an invariant to a classifier that separates good and bad samples, we are able to obtain predicates in polynomial time that are richer than any existing symbolic program verification tool we are familiar with.

4.2 Non-linear Invariants

Our geometric method of extracting likely invariants carries over to polynomial inequalities. Assume we are given a fixed bound k on the degree of the polynomials. Consider a d-dimensional point $\vec{x} = (x_1, \ldots, x_d)$. We can map $\vec{x}$ to a $\binom{d+k-1}{k}$-dimensional space by considering every possible monomial involving the components of $\vec{x}$ of maximum degree k as a separate dimension. Thus,

$$\phi(\vec{x}) = (x_1^{\alpha_1} x_2^{\alpha_2} \ldots x_d^{\alpha_d} \mid \sum_i \alpha_i \leq k, \alpha_i \in \mathbb{N}) \qquad (2)$$

Using the mapping ϕ, we can transform every point $\vec{x}$ into a higher dimensional space. In this space, polynomial inequalities of degree k are linear half-spaces, and so the entire machinery above carries through without any changes. In the general case, when we have no information about the invariant then we will take time exponential in d. When we know the slopes or the predicates constituting the invariants then we can get efficient algorithms by following the approach of Section 4.1. Therefore, we can infer likely invariants that are arbitrary boolean combinations of polynomial inequalities of a given degree.

4.3 Recovering Soundness

Once we obtain a classifier, we want to use it to construct proofs for programs. But the classifier is not guaranteed to be an invariant. To obtain soundness, we augment our learning algorithm with a theorem prover using a standard *guess-and-check* loop [55, 54]. We sample, perform learning, and propose a candidate invariant using the set cover approach for learning geometric concepts as described in Section 3.2 (the guess step). We then ask a theorem prover to check whether the candidate invariant is indeed an invariant (the check step). If the check succeeds we are done. Otherwise, the candidate invariant is not an invariant and we sample more states and guess again. When we terminate successfully, we have computed a sound invariant. For a candidate invariant I, we make the following queries:

1. The candidate invariant is weaker than the pre-condition $P \Rightarrow I$.
2. The candidate invariant implies the post-condition $I \wedge \neg E \Rightarrow Q$.
3. The candidate invariant is inductive $\{I \wedge E\}S\{I\}$.

If all three queries succeed, then we have found an invariant. Note that since we are working with samples, I is neither an under-approximation nor an over-approximation of the actual invariant. If the first constraint fails, then a counter-example is a good state that I classifies as bad. If the second constraint fails, then a counter-example is a bad state that I classifies as good. If the third constraint, representing inductiveness, fails then we get a pair of states (x, y) such that I classifies x as good, y as bad, and if the loop body starts its execution from state x then it can terminate in state y. Hence if x is good then so is y and (x, y) refutes the candidate I. However, x is unlabelled, i.e., we do not know whether it is a good state or not and we cannot add x and y to samples directly.

Now, we want our learning algorithm to generate a classifier that respects the pair (x, y) of counter-example states: if the classifier includes x then it also includes y. If the invariant has s hyperplanes then the greedy set cover can be extended to generate a separator between good and bad samples that respects such pairs. The basic idea is to greedily select the hyperplanes which make the most number of pairs consistent. Moreover the number of hyperplanes in the output is guaranteed to be $\mathcal{O}(s(\log m)^2)$: the size of the predicate can increase linearly with the number of pairs. This algorithm can be used to guide our learning algorithm in the case it finds an invariant that is not inductive. Note that the need for this extension did not arise in our experiments. Using a small amount of data, greedy set cover was sufficient to find an invariant. For buggy programs, a good state g, a bad state b, and a sequence of pairs $(x_1, x_2), (x_2, x_3), \ldots, (x_{k-1}, x_k)$ such that $g = x_1$ and $b = x_k$ is an error trace, i.e., certificate for a bug.

When we applied guess-and-check in our previous work [55, 54] to infer relevant predicates for verification, we checked for only two out of the three constraints listed above (Section 6). Hence, these predicates did not prove any program property and moreover they were of limited expressiveness (no disjunctions among other restrictions). Checking fewer constraints coupled with reduced expressiveness made it straightforward to incorporate counter-examples. In contrast, we now must deal with the kinds of counter-examples (good, bad, and unlabeled) for an expressive class of predicates. Handling all three kinds is necessary to guarantee progress, ensuring that an incorrect candidate invariant is never proposed again. However, if the candidates are inadequate then the guess-and-check procedure will loop forever: Inadequacy results in candidate invariants that grow linearly with the number of samples.

If we want to analyze a single procedure program with multiple loops, then we process the loops beginning with the last, innermost loop and working outwards and upward to the first, outermost loop. The invariants of the processed loops become assertions or postconditions for the to-be-processed loops. While checking the candidate invariants, the condition that the candidate invariant should be weaker than the pre-condition is only checked for the topmost outermost loop L and not for others. If this check generates a counter-example then the program is executed from the head of L with the variables initialized using the counter-example. This execution generates new good states for the loops it reaches and invariant computation is repeated for these loops.

5 Experimental Evaluation

We have implemented and evaluated our approach on a number of challenging C benchmarks. Greedy set cover is implemented in one hundred lines of MATLAB code. We use Havoc [5] to generate BoogiePL programs from C programs annotated with candidate invariants. Next, Boogie [7] verification condition generator operates on the BoogiePL programs to check the candidate invariants by passing the verification conditions to Z3 theorem prover [45]. All experiments were performed on a 2.67GHz Intel Xeon processor system with 8 GB RAM running Windows 7 and MATLAB R2010b.

Implementation notes Our implementation analyzes single procedure C programs with integer variables and assertions. Since all these programs contain loops, we need to compute invariants that are strong enough to prove the assertions. For every loop, our technique works as follows: first, we instrument the loop head to log the values of the variables in scope. Next, we run the program till termination on some test inputs to generate data. All internal non-deterministic choices, such as non-deterministic tests on branches, are randomly selected. All states reaching the loop head are stored in a matrix `good`. We then compute the null space of `good` to get the sub-space J in which the good states lie: J represents the equality relationships that the good states satisfy. Working in the lower dimensional sub-space J improves the performance of our algorithms by effectively reducing d, the number of independent variables.

Next, from the loop body, we statically identify the predicate B representing the states that will violate some assertion after at most one iteration of the loop. We then sample the bad states from the predicate $B \wedge J$. The good and bad samples are then used to generate the set of candidate hyperplanes $\mathcal{F}$ using the specified slopes – octagons are sufficient for all programs except `seq-len`.

We perform another optimization: we restrict the candidates to just the octagons passing through the good states, thus reducing the number of candidates. Note that this optimization still leads to an adequate set of candidates and we

Table 1. `Program` is the name, `LOC` is lines, `#Loops` is the number of loops, and `#Vars` is the number of variables in the benchmark. `#Good` is the maximum number of good states, `#Bad` is the maximum number of bad states, and `Learn` is the maximum time of the learning routine over all loops of the program. `Check` is time by BOOGIE for proving the correctness of the whole program and `Result` is the verdict: OK is verified, FAIL is failure of our learning technique, and PRE is verified but under certain pre-conditions.

`Program`	`LOC`	`#Loops`	`#Vars`	`#Good`	`#Bad`	`Learn`(s)	`Check`(s)	`Result`
`fig6` [31]	16	1	2	3	0	0.030	1.04	OK
`fig9` [31]	10	1	2	1	0	0.030	0.99	OK
`prog2` [31]	19	1	2	10	0	0.034	1.00	OK
`prog3` [31]	29	1	4	8	126	0.106	1.05	OK
`test` [31]	30	1	4	20	0	0.162	1.00	OK
`ex23` [36]	20	1	2	111	0	0.045	1.05	OK
`sas07` [29]	20	1	2	103	6112	2.64	1.02	OK
`popl07` [32]	20	1	2	101	10000	2.85	0.99	OK
`get-tag` [35]	120	2	2	6	28	0.092	1.04	OK
`hsort` [35]	47	2	5	15	435	0.19	1.05	OK
`maill-qp` [35]	92	1	3	9	253	0.11	1.05	OK
`msort` [35]	73	6	10	9	77	0.093	1.12	OK
`nested` [35]	21	3	4	49	392	0.24	0.99	OK
`seq-len1` [35]	44	6	5	36	1029	0.32	1.04	PRE
`seq-len` [35]	44	6	5	224	3822	4.39	1.04	OK
`spam` [35]	57	2	5	11	147	1.01	1.05	OK
`svd` [35]	50	5	5	150	1708	4.92	0.99	OK
`split`	20	1	5	36	4851	FAIL	NA	FAIL
`div` [53]	28	2	6	343	248	2.03	1.04	OK

retain our learning guarantees. Next, using the greedy algorithm, we select the hyperplanes that separate the good from the bad states, and return a set of half-spaces $\mathcal{H}$ and a partial boolean function f: $f(b_1, \dots, b_{|\mathcal{H}|})$ that represents the label of the cell that lies inside the half-spaces for which b_i's are *true* and outside the half-space for which b_i is *false*. This algorithm is linear in the number of bad states and its complexity is governed almost entirely by the number of good states. For our benchmarks, $|\mathcal{H}|$ was at most 8. We use the Quine-McCluskey algorithm for logic minimization (Section 3.5) that returns the smallest total boolean function g that agrees with f. Conjoining the predicate obtained using g and $\mathcal{H}$ with J yields a candidate invariant. This invariant is added as an annotation to the original program that is checked with BOOGIE for assertion violations.

Evaluation An important empirical question is how much data is sufficient to obtain a sound invariant. To answer this question, we adopt the following method for generating data: we run the programs on all possible inputs s.t. all input variables have their values between $[-1, N]$ where N is initially zero. This process generates good states at the loop head. Next we generate bad states and check whether our first guess is an invariant. If not then we continue generating more bad states and checking if the guess is an invariant. If we have generated 10,000 bad states and still have not found an invariant then we increment N by one and repeat the process. We are able to obtain a sound invariant within four iterations of this process for our linear benchmarks; `div` needs ten iterations: it needs more data as the (non-linear) invariant is found in a higher dimensional space.

Now we explain our approach of sampling bad states given a set of good states. Each variable x at the loop head takes values in some range $[L_x, M_x]$ for the good states. To sample the bad states, we exhaustively enumerate states (in the subspace in which the good states lie) where the value of each variable x varies over the range $[L_x, M_x]$. For deterministic programs with finite number of reachable states, any enumerated state that is unreachable is labeled bad. For others, bad states are generated by identifying the enumerated states satisfying the predicate B representing bad states. Because this process can enumerate a very large number of states unless the range or number of variables is small, we incrementally enumerate the states until we generate 10,000 bad states. The results in Table 1 show the number of good states (column 5) and bad states (column 6) that yield a sound invariant.

We observe that only a few good states are required for these benchmarks, which leads us to believe that existing test suites of programs should be sufficient for generating sound invariants. We observe that our sampling strategy based on enumeration generates many bad states that are not useful for the algorithm. The candidate invariant is mainly determined by the bad states that are close to the good states and not those that are further away and play no role in determining the good state/bad state boundary. The complexity of our algorithm is governed mainly by the good states, due to our optimizations, and hence generating superfluous bad states is not an issue for these benchmarks. Since the candidate inequalities are determined by the good and bad states, the good and bad samples should be generated with the goal of including the inequalities of the invariants in the set of candidates. Note that we use a naive strategy for sampling. Better strategies directed towards the above goal are certainly possible and may work better.

The benchmarks that we used for evaluating our technique are shown in the first column (labeled `Program`) of Table 1. Lee-Yannakakis partition refinement algorithm [42] does not work well on `fig6`; Synergy [31] fails to terminate on `fig9`; `prog2` has a loop with a large constant number of iterations and predicate abstraction based tools like Slam take time proportional to the number of loop iterations. The program `prog3` requires a disjunctive invariant. For `test` we find the invariant $y = x + \mathit{lock}$: Slam finds the disjunctive invariant $(x = y \Rightarrow \mathit{lock} = 0 \wedge x \neq y \Rightarrow \mathit{lock} = 1)$. For `ex23`, we discovered the invariant $z = \mathit{counter} + 36y$. This is possible because the size of constants are bounded only for computing inequalities: the equalities in J have no restriction on the size of constants. Such relationships are beyond the scope of tools performing abstract interpretation over octagons [40]. The equalities in J are sufficient to verify the correctness of the benchmarks containing a zero in column `#Bad` of Table 1. The programs `sas07` and `popl07` are deterministic programs requiring disjunctive invariants. We handle these without using any templates [35]. The programs `get-tag` through `svd` are the benchmarks used to evaluate the template based invariant generation tool InvGen [35]. As seen from Table 1, we are faster than InvGen on half of these programs, and slower on the other half.

We modify `seq-len` to obtain the benchmark `seq-len1`; the program `seq-len1` assumes that all inputs are positive. We are able to find strong invariants for the loops, using octagons for slopes, that are sufficient to prove the correctness of this program. These invariants include sophisticated equalities like $i + k = n0 + n1 + n2$. Since we proved the correctness by assuming a pre-condition on inputs, the `Result` column says PRE. Next, we analyze `seq-len`, that has no pre-conditions on inputs, using octagons as slopes. We obtain a separator that has as many linear inequalities as the number of input states; such a predicate will not generalize. For this example, there is no separator small in size if we restrict the domain of our slopes to octagons. Therefore, we add slopes of hyperplanes that constitute invariants of `seq-len1` and similar slopes to our bag of slopes. We are then able to prove `seq-len` correct by discovering invariants like $i + k \geq n0 + n1 + n2$. This demonstrates how we can find logically stronger invariants in specialized contexts.

The `split` program requires an invariant that uses an interpreted function *iseven*. Our approach fails on this program as the desired invariant cannot be expressed as an arbitrary boolean combinations of half-spaces. For the `div` program, the objective is to verify that the computed remainder is less than the divisor and the quotient times divisor plus remainder is equal to dividend. Using the technique described in Section 4.2 with a degree bound of 2, we are able to infer a invariant that proves the specification. We are unaware of any previous technique that can prove the specification of this benchmark.

6 Related Work

In this section, we compare our approach with existing techniques for linear and non-linear invariant generation. Since the literature on invariant inference is rich, we only discuss the techniques closest to our work.

6.1 Comparison with Linear Invariant Generation

Invariant generation tools that are based on either abstract interpretation [23, 21], or constraint solving [19, 35], or their combination [18], cannot handle arbitrary boolean combinations of half-spaces. Similar to us, Clousot [41] improves its performance by conjoining equalities and inequalities over boxes. Some approaches like [25, 26, 52, 32, 34, 29, 43] can handle disjunctions, but they restrict the number of disjunctions by widening, manual input, or trace based heuristics. In contrast, [28] handles disjunctions of a specific form.

Predicate abstraction based tools are geared towards computing arbitrary boolean combinations of predicates [6, 9, 31, 1, 10, 30]. Among these, YOGI [31] uses test cases to determine where to refine its abstraction. However, just like [47], it uses the trace and not the concrete states generated by a test. INVGEN [35] uses test cases for constraint simplification, but does not generalize from them with provable generalization guarantees. Amato et al. [2] analyze data from program executions to tune their abstract interpretation. Recently, we ran support vector machines [20], a widely used machine learning algorithm, in a guess-and-check loop to obtain a sound interpolation procedure [55]. However, [55] cannot handle disjunctions and computed interpolants need not be inductive.

Daikon [24] is a tool for generating likely invariants using tests. Candidate invariants are generated using templates, and candidates that violate some test case are removed. Since the invariants are based on templates, Daikon is less expressive than our approach. It is interesting to note that our empirical results are consistent with those reported in [49]: a small number of states can cover most program behaviors. Random interpretation [33] trade-offs complexity of program analysis for a small probability of unsoundness. In contrast, our guarantees are sound and we trade expressiveness for efficiency.

6.2 Comparison with Tools for Non-linear Invariants

Existing sound tools for non-linear invariant generation can produce invariants that are conjunctions of polynomial equalities [51, 38, 50, 14, 46, 53, 17]. However, by imposing strict restrictions on syntax (such as no nested loops) [51, 38] do not need to assume the degree of polynomials as the input. Bagnara et al. [4] introduce new variables for monomials and generate linear invariants over them by abstract interpretation over convex polyhedra. Our domain is more expressive: arbitrary boolean combinations of polynomial inequalities.

Nguyen et al. [48] give an unsound algorithm for generation of likely invariants that are conjunctions of polynomial equalities or inequalities. For equalities, they compute the null space of good samples (obtained from tests) in the higher dimensional space described in Section 4.2, that is also one of the steps of our technique. For generation of candidate polynomial inequalities they find the convex hull of the good samples in the higher dimensional space. In addition to limiting the expressiveness to just conjunction of polynomial inequalities, this step is computationally very expensive. In a related work, we ran [48] in a guess-and-check loop to obtain an algorithm [54], with soundness and termination guarantees, for generating polynomial equalities as invariants. A termination proof was possible as [54] can return the trivial invariant *true*: it is not required to find invariants strong enough to prove some property of interest. This technique can handle only the benchmarks that require zero bad states in Table 1, whereas our current technique can handle all the benchmarks of [54].

7 Conclusion

We have presented a machine learning perspective to verifying safety properties of programs and demonstrated how it helps us achieve guarantees and expressiveness. The learning algorithm performs a set cover and given an adequate set of candidate inequalities, it has the guarantee that the output candidate invariant uses at most a logarithmic number of inequalities more than the simplest true invariant. Hence the algorithm is biased towards simple invariants and hence parsimonious proofs. The PAC learning guarantees for this algorithm formally capture the generalization properties of the candidate invariants. Disjunctions and non-linearities are handled naturally with no a priori bound on the number of disjunctions. We trade expressiveness for efficiency by changing the abstract domains and demonstrate our approach on challenging benchmarks. The literature on classification algorithms is rich and it will be interesting to see how different classification algorithms perform on the task of invariant inference. Learning algorithms for data structures manipulating programs are left as future work.

Acknowledgements We thank Hongseok Yang and the anonymous reviewers for their constructive comments. Praneeth Netrapalli, Divya Gupta, and Prateek Jain helped in extending the classification algorithm to handle pairs. Sharma performed part of the work reported here during a summer internship at Microsoft Research India. This material is based on research sponsored by NSF grant CCF-0915766 and the Air Force Research Laboratory, under agreement number FA8750-12-2-0020. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

1. Albarghouthi, A., Gurfinkel, A., Chechik, M.: Craig interpretation. In: SAS. pp. 300–316 (2012)
2. Amato, G., Parton, M., Scozzari, F.: Discovering invariants via simple component analysis. J. Symb. Comput. 47(12), 1533–1560 (2012)
3. Bagnara, R., Hill, P.M., Zaffanella, E.: Widening operators for powerset domains. STTT 9(3-4) (2007)
4. Bagnara, R., Rodríguez-Carbonell, E., Zaffanella, E.: Generation of basic semi-algebraic invariants using convex polyhedra. In: SAS. pp. 19–34 (2005)
5. Ball, T., Hackett, B., Lahiri, S.K., Qadeer, S., Vanegue, J.: Towards scalable modular checking of user-defined properties. In: VSTTE. pp. 1–24 (2010)
6. Ball, T., Rajamani, S.K.: The SLAM toolkit. In: CAV. pp. 260–264 (2001)
7. Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: FMCO. pp. 364–387 (2005)
8. Berger, B., Rompel, J., Shor, P.W.: Efficient NC algorithms for set cover with applications to learning and geometry. J. Comput. Syst. Sci. 49(3), 454–477 (1994)

9. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker BLAST. STTT 9(5-6), 505–525 (2007)
10. Beyer, D., Henzinger, T.A., Majumdar, R., Rybalchenko, A.: Path invariants. In: PLDI. pp. 300–309 (2007)
11. Blumer, A., Ehrenfeucht, A., Haussler, D., Warmuth, M.K.: Learnability and the Vapnik-Chervonenkis dimension. JACM 36(4), 929–965 (October 1989)
12. Brönnimann, H., Goodrich, M.T.: Almost optimal set covers in finite VC-dimension. In: SoCG. pp. 293–302 (1994)
13. Bshouty, N.H., Goldman, S.A., Mathias, H.D., Suri, S., Tamaki, H.: Noise-tolerant distribution-free learning of general geometric concepts. In: STOC. pp. 151–160 (1996)
14. Cachera, D., Jensen, T.P., Jobin, A., Kirchner, F.: Inference of polynomial invariants for imperative programs: A farewell to Gröbner bases. In: SAS. pp. 58–74 (2012)
15. Chvatal, V.: A greedy heuristic for the set-covering problem. Mathematics of Operations Research 4(3), 233–235 (1979)
16. Clarisó, R., Cortadella, J.: The octahedron abstract domain. In: SAS. pp. 312–327 (2004)
17. Colón, M.: Approximating the algebraic relational semantics of imperative programs. In: Static Analysis Symposium (SAS). pp. 296–311 (2004)
18. Colón, M., Sankaranarayanan, S.: Generalizing the template polyhedral domain. In: ESOP. pp. 176–195 (2011)
19. Colón, M., Sankaranarayanan, S., Sipma, H.: Linear invariant generation using non-linear constraint solving. In: CAV. pp. 420–432 (2003)
20. Cortes, C., Vapnik, V.: Support-vector networks. Machine Learning 20(3), 273–297 (1995)
21. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL. pp. 238–252 (1977)
22. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: POPL. pp. 269–282 (1979)
23. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL. pp. 84–96 (1978)
24. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The Daikon system for dynamic detection of likely invariants. Sci. Comput. Program. 69(1–3), 35–45 (2007)
25. Fähndrich, M., Logozzo, F.: Static contract checking with abstract interpretation. In: FoVeOOS. pp. 10–30 (2010)
26. Filé, G., Ranzato, F.: Improving abstract interpretations by systematic lifting to the powerset. In: GULP-PRODE (1). pp. 357–371 (1994)
27. Flanagan, C., Leino, K.R.M.: Houdini, an annotation assistant for ESC/Java. In: FME. pp. 500–517 (2001)
28. Ghorbal, K., Ivancic, F., Balakrishnan, G., Maeda, N., Gupta, A.: Donut domains: Efficient non-convex domains for abstract interpretation. In: VMCAI. pp. 235–250 (2012)
29. Gopan, D., Reps, T.W.: Guided static analysis. In: SAS. pp. 349–365 (2007)
30. Gulavani, B.S., Chakraborty, S., Nori, A.V., Rajamani, S.K.: Automatically refining abstract interpretations. In: TACAS. pp. 443–458 (2008)
31. Gulavani, B.S., Henzinger, T.A., Kannan, Y., Nori, A.V., Rajamani, S.K.: Synergy: a new algorithm for property checking. In: FSE. pp. 117–127 (2006)

32. Gulwani, S., Jojic, N.: Program verification as probabilistic inference. In: POPL. pp. 277–289 (2007)
33. Gulwani, S., Necula, G.C.: Discovering affine equalities using random interpretation. In: POPL. pp. 74–84 (2003)
34. Gulwani, S., Srivastava, S., Venkatesan, R.: Program analysis as constraint solving. In: PLDI. pp. 281–292 (2008)
35. Gupta, A., Majumdar, R., Rybalchenko, A.: From tests to proofs. In: TACAS. pp. 262–276 (2009)
36. Ivancic, F., Sankaranarayanan, S.: NECLA Static Analysis Benchmarks `http://www.nec-labs.com/research/system/systems_SAV-website/small_static_bench-v1.1.tar.gz`
37. Jhala, R., McMillan, K.L.: A practical and complete approach to predicate refinement. In: TACAS. pp. 459–473 (2006)
38. Kovács, L.: A complete invariant generation approach for P-solvable loops. In: Ershov Memorial Conference. pp. 242–256 (2009)
39. Lahiri, S.K., Qadeer, S.: Complexity and algorithms for monomial and clausal predicate abstraction. In: CADE. pp. 214–229 (2009)
40. Lalire, G., Argoud, M., Jeannet, B.: The Interproc Analyzer. `http://pop-art.inrialpes.fr/people/bjeannet/bjeannet-forge/interproc/index.html`
41. Laviron, V., Logozzo, F.: Subpolyhedra: a family of numerical abstract domains for the (more) scalable inference of linear inequalities. STTT 13(6), 585–601 (2011)
42. Lee, D., Yannakakis, M.: Online minimization of transition systems (extended abstract). In: STOC. pp. 264–274 (1992)
43. Mauborgne, L., Rival, X.: Trace partitioning in abstract interpretation based static analyzers. In: ESOP. pp. 5–20 (2005)
44. McCluskey, E.J.: Minimization of boolean functions. Bell Systems Technical Journal 35(6), 1417–1444 (1956)
45. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS. pp. 337–340 (2008)
46. Müller-Olm, M., Seidl, H.: Computing polynomial program invariants. Information Processing Letters 91(5), 233–244 (2004)
47. Naik, M., Yang, H., Castelnuovo, G., Sagiv, M.: Abstractions from tests. In: POPL. pp. 373–386 (2012)
48. Nguyen, T., Kapur, D., Weimer, W., Forrest, S.: Using dynamic analysis to discover polynomial and array invariants. In: ICSE (2012)
49. Nimmer, J.W., Ernst, M.D.: Automatic generation of program specifications. In: ISSTA. pp. 229–239 (2002)
50. Rodríguez-Carbonell, E., Kapur, D.: Automatic generation of polynomial invariants of bounded degree using abstract interpretation. Sci. Comput. Program. 64(1), 54–75 (2007)
51. Rodríguez-Carbonell, E., Kapur, D.: Generating all polynomial invariants in simple loops. J. Symb. Comput. 42(4), 443–476 (2007)
52. Sankaranarayanan, S., Ivancic, F., Shlyakhter, I., Gupta, A.: Static analysis in disjunctive numerical domains. In: SAS. pp. 3–17 (2006)
53. Sankaranarayanan, S., Sipma, H., Manna, Z.: Non-linear loop invariant generation using Gröbner bases. In: POPL. pp. 318–329 (2004)
54. Sharma, R., Gupta, S., Hariharan, B., Aiken, A., Liang, P., Nori, A.V.: A data driven approach for algebraic loop invariants. In: ESOP. pp. 574–592 (2013)
55. Sharma, R., Nori, A., Aiken, A.: Interpolants as classifiers. In: CAV. pp. 71–87 (2012)
56. Valiant, L.G.: A theory of the learnable. Commun. ACM 27(11), 1134–1142 (1984)