Bias-Variance Tradeoffs in Program Analysis

Rahul Sharma
Stanford University
sharmar@cs.stanford.edu

Aditya V. Nori
Microsoft Research
adityan@microsoft.com

Alex Aiken
Stanford University
aiken@cs.stanford.edu

Abstract

It is often the case that increasing the precision of a program analysis leads to worse results. It is our thesis that this phenomenon is the result of fundamental limits on the ability to use precise abstract domains as the basis for inferring strong invariants of programs. We show that *bias-variance tradeoffs*, an idea from learning theory, can be used to explain why more precise abstractions do not necessarily lead to better results and also provides practical techniques for coping with such limitations. Learning theory captures precision using a combinatorial quantity called the *VC dimension*. We compute the VC dimension for different abstractions and report on its usefulness as a precision metric for program analyses. We evaluate *cross validation*, a technique for addressing bias-variance tradeoffs, on an industrial strength program verification tool called YOGI. The tool produced using cross validation has significantly better running time, finds new defects, and has fewer time-outs than the current production version. Finally, we make some recommendations for tackling bias-variance tradeoffs in program analysis.

Categories and Subject Descriptors D.2.4 [*Program Verification*]: Statistical methods; F.3.2 [*Semantics of Programming Languages*]: Program analysis; I.2.6 [*Learning*]: Parameter learning

Keywords Program Analysis; Machine Learning; Verification

1. Introduction

In program analysis, it is well understood that imprecise abstractions can lead to inferior results. However, what is not so well understood is that precise abstractions can also produce inferior results, in some cases even worse than very imprecise abstractions. We show that *bias-variance tradeoffs*, an idea from learning theory, can be used to explain how the quality of analysis results changes with precision. Learning theory quantifies precision using a combinatorial quantity called *Vapnik-Chervonenkis dimension* or *VC dimension*; we use VC dimension to analyze the behavior of a number of commonly used program analysis abstractions. In addition, using *cross validation*, a technique routinely applied in machine learning to address bias-variance tradeoffs, we are able to improve the performance of YOGI, an industrial strength program verification tool.

How can better precision adversely affect an analysis? Note that we are not talking about cost or efficiency—the question only concerns the quality of the results. The class of static analysis systems to which our results apply have two distinguishing characteristics. First, there is some domain of facts (e.g., an abstract domain) over which the analysis computes. Second, there is some step in the analysis that takes such facts and attempts to generalize them (e.g., to an invariant of the program). Many analysis frameworks such as abstract interpretation [17], counter-example guided abstraction refinement (CEGAR) [13], and various other inference techniques [10, 28, 49] have this structure. For example, abstract interpretation can use *widening* (Section 4.1) and CEGAR can use *interpolants* (Section 4.2) as a generalization step. There are also static analyses that do not have this structure and our results do not apply to them (see Section 6). Using results on bias-variance tradeoffs we show that increasing the precision of the underlying domain, at some point, may lead to worse results from the generalization step.

Consider the program in Figure 1, which we analyze using different program analyses with increasing precision. Each program analysis is a basic abstract interpreter over a different abstract domain. First, consider an interval analysis [16], which infers upper and lower bounds for numeric variables. Using intervals to analyze the program in Figure 1, we obtain the loop invariant $1 \leq i \leq 5 \wedge j \geq 0$ [36]. This result is perhaps the best invariant that one could expect with intervals. Intervals can only express facts about single variables, and hence the invariant has no details on any relationship between i and j. The problem is lack of precision: the abstract domain is not precise enough to express certain behaviors. We increase the precision to octagons [39], which can infer bounds on the sum or difference of pairs of variables. We obtain the following loop invariant [36]:

$$1 \leq i \leq 5 \wedge i - j \leq 1 \wedge i + j \geq 1 \wedge j \geq 0$$

This result is quite good, providing a useful relationship between i and j and good bounds. Next, we use the even more precise domain of polyhedra [19] which can express bounds on arbitrary linear combinations of program variables. We obtain the weak invariant $i \leq 5$ [36]. This result is not only worse than octagons, it is even worse than intervals. Increasing the precision of the analysis results in a decrease in the quality of the results.

Let us examine this outcome in some detail. The abstract interpreter generates some abstract states and then tries to generalize via joins and widening [17]. However, because polyhedra are so expressive, there are many incomparable polyhedra that perfectly describe any finite set of abstract states. It is difficult for the generalization step to pick the best one for this particular program from such a large candidate pool. For this example, we start with $i = 1 \wedge j = 0$ as the initial abstract state and we show the fixpoint iterations [17] published in [41]. First, we fit on $i = 1, j = 0$ and $i = 2, j = 1$ to obtain the polyhedra $-i + j \geq -1 \wedge i \geq 1$. Next, we obtain $i + j \geq -1 \wedge 7i - 4j \geq 7$ that fits $i = 1, j = 0$, $i = 2, j = 1$, and $i = 5, j = 7$. Note that there are an unbounded number of polyhedra, including the one discovered by octagons,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
POPL '14, January 22–24, 2014, San Diego, CA, USA.
Copyright © 2014 ACM 978-1-4503-2544-8/14/01...$15.00.
http://dx.doi.org/10.1145/2535838.2535853

```
1: int i = 1, j = 0;
2: while (i<=5) {
3:  j = j+i ;
4:  i = i+1;
5: }
```

Figure 1. Example program from [41].

that can fit these states. The domain of polyhedra is so precise that the generalization step has many valid choices and hence it can pick a hypothesis that fits these specific abstract states but does not hold in general. In the next iteration, we obtain $-i + j \geq -1$, and, finally, *true*. Using the loop guard and narrowing [17], we terminate with $i \leq 5$ as the invariant. One might be tempted to blame specific choices made in this analysis run for the result, but that misses the point that the phenomenon is general: any method that attempts to select the best generalization from a large set of equally viable but different candidates will run into the same problem on many programs. For example, other analyses such as [3] meet a similar fate on this example [41].

Very abstractly, a program has some behaviors, and if an analysis is not expressive enough to capture these behaviors, then we have *underfitting* and the results are poor. If the analysis engine is too expressive, then it can *overfit* the specific behaviors and fail to generalize. Currently, program analysis designers apply folk knowledge to avoid underfitting and overfitting. Our aim is to give a formal framework to understand these rules of thumb and use the foundations to obtain better tools.

In learning theory [34], underfitting is characterized by *bias* and overfitting is characterized by *variance* [21], and by varying the precision, one gets a bias-variance tradeoff. Low precision leads to high bias and low variance, while high precision leads to low bias but high variance. With an appropriate choice of precision, one can balance bias and variance and obtain good results. As an example, consider Figure 2. YOGI is a verification engine in Microsoft Windows SDV (Static Driver Verifier) toolkit that checks safety properties of Windows device drivers [25]. In Figure 2, increasing values on the x-axis indicate increasing precision (for details see [43] and Section 5). The y-axis shows the time taken by YOGI on 2490 verification tasks (a superset of the tasks reported in [25]). Higher analysis times are indicative of more time-outs and poorer quality of results. One observes the bias-variance tradeoff in Figure 2. At low precision the performance of the tool is poor. With increasing precision, the bias decreases and the performance of the tool improves. However, after a certain point, the variance starts increasing and the performance starts to degrade.

To develop a theory that can explain these empirical observations, we need a formal definition of generalization. Unfortunately, even though the term generalization frequently occurs in the program analysis literature, defining generalization precisely is difficult and there is no widely accepted definition. Just as complexity theory works with a model of machines and generates qualitative (as opposed to quantitative) results for comparing the efficiency of algorithms, we want a useful theory, perhaps working with a model, that generates useful qualitative feedback about bias-variance tradeoffs.

In a recent work, [47] applied the definition of generalization given by the PAC (*probably approximately correct*) learning framework [48] to prove that a verification algorithm for checking safety properties generalizes. According to this definition (which we give formally in Section 2), an algorithm generalizes if given enough samples of program states as input, it is likely to generate predicates that separate almost all the program's safe reachable states from erroneous program states. In this paper, we explore an alternative use of this framework, namely modeling bias-variance tradeoffs. We

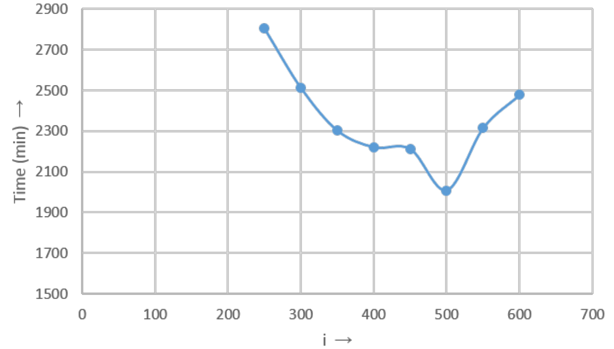

Figure 2. Result of running YOGI on 2490 driver-properties pairs. The best performance is achieved at $i = 500$.

do not claim that this definition is *the* correct definition of generalization. However, the motivation for our work is largely practical, and we show that this existing framework yields some immediately useful results. In the future, different definitions of generalization might be available and the framework developed here can be instantiated with the alternative definitions to derive other useful conclusions for the benefit of program analysis tools.

We believe that if program analysis tool designers are explicitly made aware of bias-variance tradeoffs and have mathematical tools to qualitatively reason about generalization, they can make better informed design decisions. In our framework, variance is proportional to VC dimension (Theorem 2.2). Hence, a high VC dimension is indicative of overfitting. We calculate the VC dimension of several abstractions used in program analysis (Section 3), including abstractions for numerical programs, array manipulating programs, and heap manipulating programs. We observe that more precise abstractions, that is, the abstractions with higher variance, have higher VC dimension. These proofs increase our confidence in the applicability of our framework to program analyses and provide evidence that VC dimension is a useful qualitative metric for characterizing the precision of abstractions.

Since our definition of generalization has been borrowed from learning theory, we can build on well-known techniques in the machine learning community for addressing bias-variance tradeoffs. Cross validation [2] is one of the simplest techniques for this purpose. Consider a fully automatic analysis tool that has a number of configuration parameters. How to set these parameters optimally is usually unclear, and the typical case is that such an optimal configuration might not even exist, although clearly some configurations are better than others. One logical candidate configuration is the one that performs best on a benchmark suite. However, we demonstrate that this strategy can overfit on the particular benchmark suite and significantly degrade the quality of the tool on new inputs (Section 5). Cross validation avoids overfitting, and we show that by applying cross validation to tune the configuration parameters of YOGI [25], we are able to significantly improve YOGI's running time while also reducing the number of timeouts and finding new defects.

Thus, overfitting can adversely impact program analysis in at least two ways. First, overly precise abstract domains can overfit and lead to poor generalization and hence inferior invariants. Second, tools can overfit their benchmark suite resulting in poor performance on new analysis tasks. We show that these seemingly different problems are both instances of bias-variance tradeoffs and hence the same principles (Theorem 2.2) apply to both. Learning

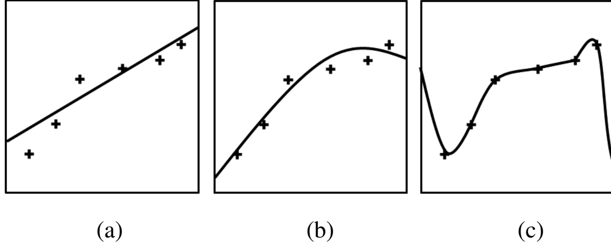

Figure 3. Regression example: underfit (a), good fit (b), overfit (c).

theory addresses bias-variance tradeoffs in general and can provide techniques to tackle both these problems.

To summarize, our contributions are as follows:

- We observe that improving precision does not necessarily lead to better results in program analysis. We connect this phenomenon to bias-variance tradeoffs in learning theory (Section 2).
- We show that VC dimension captures the precision of abstract domains used in program analysis. We calculate the VC dimension of a number of abstractions relevant to program analysis, including formulas over arrays and separation logic (Section 3).
- We explain several empirical observations in program analysis using bias-variance tradeoffs. For example, by incrementally increasing precision, one can balance bias and variance (Section 2), a strategy that is employed in several existing program analyses [32, 41, 49] (Section 4).
- Using bias-variance tradeoffs we show how overfitting to benchmark suites can result in unnecessarily poor performance on new inputs. Using cross validation to guide precision tuning, we are able to improve the performance of YOGI, a fully automatic, production quality program verification tool (Section 5).
- We make some specific recommendations for tackling bias-variance tradeoffs in program analysis (Section 6).

2. Preliminaries

We first review concepts from learning theory used in subsequent sections; readers already familiar with learning theory may safely skip this section. For more details, the reader is referred to the excellent textbook by Kearns and Vazirani [34].

2.1 Bias-Variance Tradeoffs in Regression

Bias-variance tradeoffs have been well-studied in machine learning and before introducing the formal definitions it is instructive to look at a machine learning example.

Consider the problem of regression [6]. We are given a set of input-output pairs (observations) $(x_1, y_1), \dots (x_n, y_n)$ and we want to predict the output y for additional but unknown input values x. In other words, we want to learn a function f, such that $y_i = f(x_i)$, $1 \leq i \leq n$. The standard way to solve this problem is to consider a template for f (a restricted class of functions from which the solution is chosen) and then fit the template to these observations.

Figure 3 fits three different templates to the same set of six observations. In Figure 3(a), we show the best fit line. Even though we have chosen the best of all possible lines, the fit is quite poor (meaning there are large errors even for the observed data points) and we do not expect the predictions obtained to be good. Figure 3(a) illustrates *underfitting*: lines are too imprecise to represent our observations. Next, we fit a quadratic curve (Figure 3(b)) and it seems to be a good fit: we expect it to produce good predictions. Figure 3(c) shows the fit of a polynomial of degree five: a fifth degree polynomial can interpolate between the six observations. The fit is extremely good for the actual observations, but there seems little reason for confidence in the predictions for very large or very small values of x given by this particular choice of function. Figure 3(c) illustrates *overfitting*.

The nature of the underfitting and the overfitting for the examples in Figure 1 and Figure 3 have similarities and differences. We observe that both excess precision and limited precision lead to bad results. But there are some obvious differences. The program analysis example in Figure 1 looks cleaner: the abstractions are over-approximating some of the loop behaviors, whereas in Figure 3(b) the quadratic model does not even agree with the observations. The difference is due to noise. In machine learning, the data is typically noisy, whereas programs are precise descriptions. Therefore, the definition of generalization we use and the development in this paper are for the noise-free case. Of course, programs can have bugs and these can be thought of as noise and learning theory has mechanisms for incorporating noise in generalization [34]. Defining and handling noise in program analysis is interesting future work.

2.2 Learning Theory Primer

Consider an *instance space* χ which is the set of all instances. Suppose each $x_i \in \chi$ is associated with a *label* $\ell(x_i)$ that belongs to a *label set* γ. Let $\mathcal{H}$ be a *hypothesis class*, that is, the set of all functions $h : \chi \to \gamma$ considered by the learning algorithm. The goal of a learning algorithm is to choose an $h \in \mathcal{H}$ such that for each $x_i \in \chi$, $h(x_i)$ is a good estimate of the label $\ell(x_i)$. For the example in Section 2.1, $\chi = \mathbb{R}$, $\gamma = \mathbb{R}$, and $\mathcal{H}$ is a set of polynomials. We are given a set $S = \{(x_i, y_i) : i = 1, \dots, m\} \subseteq \chi \times \gamma$ called the *training set*, and we want to find a hypothesis $h \in \mathcal{H}$ which generalizes over the whole instance space, that is, for all $x \in \chi$, $|h(x) - \ell(x)|$ is small.

The notion of instance space is general and can capture program states: it can be the collection of points in $\mathbb{R}^n$ for numerical programs, a collection of stack and heap pairs for heap manipulating programs, or a valuation of some numerical variables and arrays for array manipulating programs. One example of labels can be whether a state is reachable or unreachable and we might want our hypothesis to be a predicate which predicts for each state a label $true$ (denoting reachable) or $false$ (denoting unreachable), given some known reachable and unreachable states. For program analysis, we are interested in predicates as hypothesis classes. Hence, there are only two labels, *true* and *false*, and we limit our discussion to binary labels $\gamma = \{true, false\}$ unless stated otherwise.

Given a set of labeled instances called the training set $S \subseteq \chi \times \gamma$, one natural method to perform learning is *empirical risk minimization* (ERM): find a hypothesis $h \in \mathcal{H}$ such that the number of labeled instances $(x_i, y_i) \in S$ for which $h(x_i) \neq y_i$ is minimized. By finding a hypothesis that works well on the training set, we hope to find a hypothesis which works for the whole instance space χ.

However, it is not clear whether such an h generalizes. Next, we formally define the notion of generalization that we use. PAC learning [48] assumes that the training set S consists of m independent and identically distributed (iid) labeled instances drawn from an arbitrary but fixed distribution D over the instance and label space. If we draw a new labeled iid sample from D, then we are interested in the probability that the actual label agrees with the predicted label: if $(x, y) \sim D$ then what is $Pr\,[h(x) = y]$? So first we *train*, that is, generate a hypothesis using a training set, and then we *test*, that is, evaluate the performance of the hypothesis on the new samples. A *PAC learner* takes some samples from D as input and with high probability outputs a hypothesis that is approximately correct—if

one were to draw a new sample from D, then with high probability the predicted and the actual label agree. Since the output of the PAC learner predicts the labels for instances that it has not seen, we say that it *generalizes*.

Definition 2.1. *A learner **generalizes** if given m samples (determined by parameters δ, ϵ and the hypothesis space $\mathcal{H}$) from a distribution D, with probability $1-\delta$ it outputs a hypothesis $h \in \mathcal{H}$, such that $(x, y) \sim D \Rightarrow Pr[h(x) \neq y] \leq \epsilon$.*

Now, why does this definition make sense? Suppose the learner wants to convince an adversary that it can produce hypotheses that generalize. If an adversary controls the training set, then she can generate a very bad training set with no information about the structure of the problem. For example, the adversary can just duplicate a labeled instance an unbounded number of times and can claim to have generated a large or even an unbounded training set. Or she can generate samples in the training set that are related to some particular behaviors, and when testing the generalization properties of the generated hypothesis use completely different behaviors. The poor hypothesis generated using certain behaviors is bound to perform poorly on behaviors that it has no idea about. An algorithm that can succeed against such a powerful adversary seems unlikely and so it seems reasonable to weaken the adversary. First, to define generalization, the training set should have some guarantee of having a good coverage of behaviors; by selecting training inputs randomly, we ensure formally that our training set is not adversarially generated. In testing, the adversary might defeat the generated hypothesis by testing on very skewed inputs. By testing on iid samples, we also take this power away from the adversary.

2.2.1 Bias and Variance

First, we formally define ERM. The learner finds a hypothesis that minimizes the empirical error $\hat{\varepsilon}$ over a training set $S = \{(x_i, y_i) : 1 \leq i \leq n\}$:

$$\hat{\varepsilon}(h) = \frac{1}{m}\sum_{i=1}^{m} \mathbf{1}\{h(x_i) \neq y_i\} \tag{1}$$

where $\mathbf{1}$ is the indicator function: $\mathbf{1}\{b\} = 1$ if b is *true* and 0 if b is *false*. In *empirical risk minimization*, we try to find a hypothesis h that minimizes the empirical error $\hat{\varepsilon}(h)$ and hope that it generalizes. The generalization error $\varepsilon(h)$ for a hypothesis h is defined as follows:

$$\varepsilon(h) = Pr_{(x,y)\sim D}\left[h(x) \neq y\right] \tag{2}$$

The objective of a learning algorithm is to compute a hypothesis with low generalization error. By minimizing the empirical error, we hope to achieve this objective.

One of the fundamental theorems in machine learning is the following [34]:

Theorem 2.2. *For a hypothesis space $\mathcal{H}$, let $d = VC(\mathcal{H})$ (defined in Section 2.3). Then, given m samples, empirical risk minimization with high probability produces $\hat{h} \in \mathcal{H}$ such that:*

$$\varepsilon(\hat{h}) \leq \hat{\varepsilon}(\hat{h}) + \mathcal{O}\left(\sqrt{\frac{d}{m}}\right)$$

and also

$$\varepsilon(\hat{h}) \leq \varepsilon(h^*) + \mathcal{O}\left(\sqrt{\frac{d}{m}}\right)$$

where h^* is a hypothesis with the minimum generalization error in $\mathcal{H}$.

This theorem gives us a bound on the generalization error. In particular, the first part of the theorem bounds the generalization error using the empirical error. To generalize well or to have a low generalization error we want the bound to be small. This theorem says that we can produce a large generalization error for two reasons:

1. The term $\varepsilon(h^*)$, the *bias*, is the generalization error of the best hypothesis in $\mathcal{H}$, i.e., the one that minimizes the generalization error. If this value is large then the hypothesis class underfits: even if we select the best available hypothesis, we still have generalization errors.
2. The term $\mathcal{O}\left(\sqrt{\frac{d}{m}}\right)$, the *variance*, grows with the VC dimension or precision of the hypothesis class. If this value is large generalization errors occur from overfitting the training data.

Therefore, low precision causes generalization error due to bias and high precision causes generalization error due to variance, and this leads to the bias-variance tradeoff. A corollary of this theorem has been used by [47] for an alternative purpose: for a specific hypothesis class $\mathcal{H}$, using $VC(\mathcal{H})$ to bound the number of samples required to ensure that the generalization error is below a user-specified tolerance.

By trying multiple hypothesis classes in order of increasing precision, one can address bias-variance tradeoffs. One starts with an imprecise hypothesis class and gradually increases precision until the bounds start degrading. In the extreme, when we have an empty hypothesis class, then the bias is high and the variance is zero. At the other extreme, for very expressive classes, the bias can become zero and the variance is high. When the size of the hypothesis class increases, so that successive hypothesis classes include the previous hypothesis classes, then bias decreases monotonically and variance increases monotonically. By gradually increasing precision, we can find a "sweet spot" and achieve low bounds on generalization error [24].

2.3 VC dimension

The *Vapnik-Chervonenkis* or *VC dimension* [34] is a purely combinatorial quantity that measures the capacity of a hypothesis class.

Definition 2.3. *A hypothesis h satisfies a set of labeled instances (X, ℓ), iff $\forall x \in X$, $h(x) = \ell(x)$.*

Definition 2.4. *Given a set X of instances, a hypothesis class $\mathcal{H}$ shatters X if for any labeling ℓ there exists an $h \in \mathcal{H}$ s.t. h satisfies (X, ℓ).*

Definition 2.5. *$VC(\mathcal{H})$ is the cardinality of the largest set that $\mathcal{H}$ can shatter.*

In our setting, when a hypothesis satisfies a set of labeled instances, some of the instances are labeled $true$ and others are labeled $false$, and the hypothesis is a predicate containing all the points labeled $true$ and excluding the points labeled $false$. If a class is able to shatter large sets, then it has high precision, and it is precisely able to separate instances with different labels. The VC dimension is the largest number of points that one can shatter. If the VC dimension is too high then we can overfit, and ERM does not produce a hypothesis that generalizes. In the extreme case, if the VC dimension of a hypothesis class is infinite, then we cannot bound its generalization error (Theorem 2.2).

To prove that the VC dimension of a hypothesis class $\mathcal{H}$ is at least d, we need to show a set of d points in the instance space χ that $\mathcal{H}$ can shatter. To prove an upper bound u on VC dimension, we need to show that for *any* possible selection of u points from χ, $\mathcal{H}$ cannot shatter the u points. Since we want an upper bound on generalization error, it is generally sufficient to find upper bounds on VC dimension.

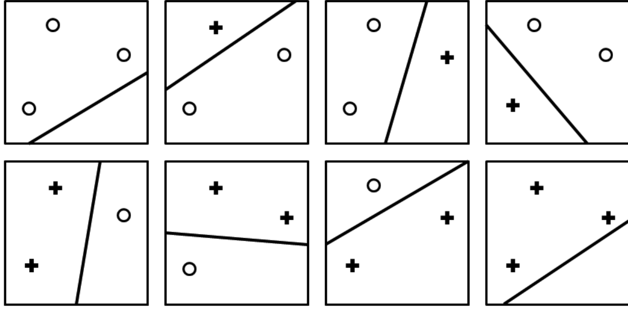

Figure 4. Shattering three points in two dimensions using one inequality.

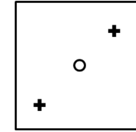

Figure 5. One inequality cannot shatter 3 collinear points.

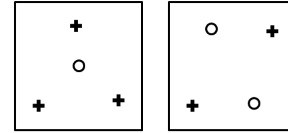

Figure 6. One inequality cannot shatter four points.

3. Abstract Domains

In this section, we calculate the VC dimension for several popular abstract domains and show that the VC dimension gives results which match our expectations. Our goal is not to calculate the VC dimension of every possible abstract domain. We consider some simple abstract domains and the techniques we develop are useful (but might not be sufficient) for computing the VC dimension of other more complicated abstract domains as well. The VC dimensions of the numerical domains we consider (Section 3.1, Section 3.2, and Section 3.3) follow from standard results in learning theory. We also consider predicates over arrays (Section 3.4) and separation logic (Section 3.5). We are unaware of any previous study of VC dimensions of these domains.

3.1 Standard numerical domains

Our main idea is to compute the VC dimension of different abstract domains in order to capture their precision. The formal proofs for the numerical domains discussed in this section are standard textbook material in machine learning [34]. The instance space χ we first consider is $\mathbb{R}^n$, which can represent the state of n-variable numerical programs with no arrays and no data structures. Arrays and data structures are discussed in Sections 3.4 and 3.5.

3.1.1 Single Inequality

First, let us consider the hypothesis class consisting of single linear inequalities. We select this class as it is one of the simplest hypothesis classes. The following result is known:

Theorem 3.1. *The VC dimension of the set of single inequalities in* n *dimensions is* $n + 1$.

Consider the 2-dimensional space $\mathbb{R}^2$. This theorem states that the VC dimension of the class of inequalities of the form $ax + by + c \geq 0$ is three. Figure 4 shows a set of three points that can be shattered using this hypothesis class. Circles are the points labeled *false* and crosses are the points labeled *true*.

Note that we cannot shatter some configurations of three points, but that does not matter for VC dimension (Definition 2.5). For instance, as shown in Figure 5, if the points are collinear, then there is a labeling that cannot be satisfied by a single inequality: there is no inequality that can include the two crosses and exclude the circle.

We cannot shatter any configuration of four points using a single inequality, and the canonical configurations that cannot be shattered are shown in Figure 6. In the first configuration of four points, we cannot satisfy the inner point with label *false* and the other three with label *true*. In the second configuration, we cannot satisfy the labeling in which diagonally opposite points have the same label and adjacent points have different labels.

To prove Theorem 3.1, observe that an inequality in $n - 1$ dimensions can be looked upon as $f(x) \geq 0$, where $f(x)$ is a plane in n dimensions passing through the origin. It is easy to see that we can shatter n points with such inequalities. Consider a set X of n points where the ith point has the coordinate 1 in the ith dimension and 0 otherwise. If $Z \subseteq X$ is the set of examples labeled $true$, then the inequality $f(x) \geq 0$, where $f(x)$ has the ith coordinate 1 if $x_i \in Z$ and -1 otherwise, satisfies X. So the VC dimension of an inequality in n dimensions is at least $n + 1$. To get the upper bound, we instantiate the following generalized lemma:

Lemma 3.2. *Let* $\mathcal{F}$ *be a function class containing functions* $f : \chi \mapsto \mathbb{R}$, *and let* $\mathcal{A} = \{\{x : f(x) \geq 0\} : f \in F\}$, *then* $VC(\mathcal{A}) \leq dimension(\mathcal{F})$.

Since planes in n dimensions passing through the origin are generated from n basis vectors, we conclude that the dimension of such planes is n, and Theorem 3.1 follows.

From this simple example of a VC dimension calculation, one can observe that computing the VC dimension of a hypothesis class can be a non-trivial task and can require reasoning about the mathematical structures underlying the instance space and the hypothesis class. Now we proceed to some of the more complicated hypothesis classes that are relevant for program analyses.

3.1.2 Intervals

Intervals or conjunctions of inequalities of the form $\pm x \geq c$ is a well-studied abstract domain [16]. It is also known for underfitting, as useful invariants often require relationships between multiple variables.

Theorem 3.3. *The VC dimension of intervals in* n *dimensions is* $2n$.

Proof. Consider the $2n$ points $X = \{x_{-n}, \ldots, x_{-1}, x_1, \ldots, x_n\}$, where x_i has $sign(i)$ as coordinate $|i|$ and the rest of the coordinates zero. We can shatter these $2n$ points using a construction that is a generalization of Figure 7. Moreover, intervals cannot shatter $2n + 1$ points. Any collection of $2n + 1$ points has at most $2n$ extreme points: an extreme point of a collection of points has the highest or the lowest coordinate along some dimension among the points in the collection. There is at least one non-extreme point as there are $2n$ extremes and $2n + 1$ total points. Since intervals are convex, if we consider any set of $2n + 1$ points, and assign the extreme points the label *true* and the non-extreme point(s) the label *false* then no interval can include all the *true* and exclude the *false* point(s). For example, if we consider $2n + 1$ points consisting of X and the origin, then the points in X are extreme points and an interval cannot satisfy the labeling, where points in X are labeled *true* and the origin is labeled *false*. $\square$

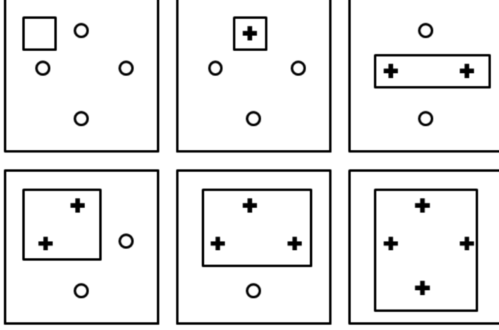

Figure 7. Shattering four points using intervals in 2 dimensions. Symmetric labelings have been omitted.

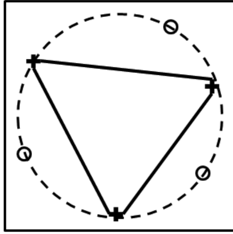

Figure 8. Using a polyhedron to satisfy a labeling.

We now move on to more expressive abstract domains and show that these have higher VC dimension than intervals.

3.1.3 Polyhedra

Given an arbitrary number of points on a circle, with arbitrary labelings, using convex hulls, polyhedra can separate the points labeled *true* from points labeled *false*. An example is shown in Figure 8. As a consequence, we have the following theorem:

Theorem 3.4. *The VC dimension of polyhedra is infinite.*

Using Theorem 2.2, we can conclude that when using polyhedra, it is not possible to bound the generalization error in our framework. However, many useful invariants require general inequalities. The fact that the VC dimension of polyhedra is infinite does not prevent us from handling general inequalities and these are addressed in the next section.

3.2 Templates

If we restrict polyhedra to k inequalities, then it turns out that the VC dimension is bounded above by $\mathcal{O}(kn\log(k))$. The proof of this fact relies on general composition theorems [8, 9] that relate how the VC dimension composes when complex hypothesis classes are obtained by composition of more primitive classes. Here we are composing k inequalities to generate a polyhedra and these theorems are applicable. To state these theorems, we require additional technical machinery that does not add to the development of the ideas in this paper and therefore we omit them.

In this case, as expected, the VC dimension is higher than intervals but seems manageable. If one can perform program analysis while keeping the number of inequalities fixed, then the generalization error is bounded in our framework and one might be able to combat overfitting. Template-based invariant inference engines perform analysis assuming the number of inequalities to be a fixed user-provided constant [15, 27, 28].

Now consider a given boolean combination instead of just conjunctions of inequalities. Approaches for performing abstract interpretation with a given fixed number of disjunctions are known [46]. In the case of template-based invariant inference, the recipe for finding the invariant is the same—one constructs a template and solves constraints to instantiate template parameters. The template can have only conjunctions [15] or a given boolean combination of inequalities [27]. Indeed, the VC dimension of a given boolean combination of k inequalities in n dimensions is also bounded by $\mathcal{O}(kn\log(k))$, which is the same as the VC dimension of conjunctions of k inequalities [9]. This observation suggests that one does not expect results to degrade significantly due to higher variance when using this more expressive template.

TCM (template constrained matrix) domains [45] provide a knob to vary precision. In TCM, one can provide linear expressions, and abstract interpretation infers lower and upper bounds for them. Intervals and octagons are special cases of TCM. As we increase the number of linear expressions, the VC dimension increases. In the limit, when we have an infinite number of linear expressions, TCM becomes polyhedra and the VC dimension is infinite. Here is an example of a family of abstract domains that can combat bias-variance tradeoffs. An intelligent template choice ensures that we neither underfit nor overfit and therefore leads to better results.

Moreover, by calculating the VC dimension, one can observe how the precision increases if additional linear expressions are added and thus can help an abstract interpretation designer make systematic decisions for choosing linear expressions by answering questions such as the expected increase in precision from adding a new linear expression.

3.3 Non-linear arithmetic

First, we compute an upper bound on the VC dimension of the hypothesis classes composed of polynomial inequalities. Recall that an upper bound on the VC dimension suffices for bounding the generalization error (see Theorem 2.2). We show that the bound depends on the degree of the polynomial. Suppose we are given that the invariant is composed of quadratic inequalities. Conceptually, we can create a new variable for every monomial up to degree two [1]. The quadratic invariant in the old set of variables becomes a linear invariant over the new set of variables. The calculations done above carry over, just with an increased dimension. Since the number of monomials increases with the degree under consideration, the bound also increases with the increasing polynomial degree.

In the extreme, when the degree is unbounded, we cannot bound the VC dimension. If there are other sources of non-linearities, similar arguments apply and we can add new variables so that non-linear invariants are reduced to linear invariants [26].

3.4 Arrays

Consider an array manipulating program. The program state is the valuation of integer variables and potentially unbounded arrays. The examples of useful invariants over arrays that occur in the program analysis literature are generally universally quantified predicates involving a small number (1 to 3) of arrays [7]. Likewise, if we consider the predicates of the form $\forall i\ .\ \pm a[i] \geq c$ as the hypothesis class $\mathcal{H}$, and the instance space χ as the valuation of an array denoted by a, then the VC dimension is 2, which is the same as an interval in one variable. Thus, such predicates can suffer from the same problem of underfitting as intervals.

Theorem 3.5. *The VC dimension of predicates $\forall i\ .\ \pm a[i] \geq c$ is 2.*

Proof. Consider three states, and let v_i denote the smallest value stored in the array of the ith state. Among the eight labelings that need to be satisfied, consider the configuration in which the states with the smallest and largest v_i are labeled *true* and the third state

is labeled *false*. This labeling cannot be satisfied by this class of predicates. Also, two distinct states, each consisting of only a single element in the array, are shattered by the predicates of the form $x \geq \pm c$. $\square$

We can extend the above result to a more general statement. Consider the instance space χ of points in $n+1$ dimensions, $\chi = \mathbb{R}^{n+1}$, that represents the values of $n+1$ numerical variables. Also consider the instance space χ^a that represents the values of an array a of unbounded size and n numerical variables, $\chi^a = \mathbb{R}^\omega \times \mathbb{R}^n$. We define three maps. The map $\mathcal{M}$ maps each predicate over χ to a predicate over χ^a, $\mathcal{M}(P(x, x_1, \ldots, x_n)) = \forall j.P(a[j], x_1, \ldots, x_n)$ and $j \notin \{x_1, \ldots, x_n\}$. We also use $\mathcal{M}$ to denote pointwise extension of this map to a set of predicates. Next, $f : \chi \rightarrow \chi^a$, is a map such that $f(c, c_1, \ldots, c_n)$ is an instance in χ^a where the array a has all elements equal to c and the ith numerical variable of χ^a is assigned c_i. Finally, $g : \chi^a \rightarrow \chi$ is a map where $g(a, c_1, \ldots, c_n)$ assigns the first numerical variable of χ an arbitrary element of a and the rest of the n variables are assigned values $c_1, \ldots, c_n$. If $\mathcal{H}$ is an arbitrary hypothesis class of predicates over χ then the following result holds:

Theorem 3.6. $VC(\mathcal{H}) = VC(\mathcal{M}(\mathcal{H}))$.

Proof. To prove $VC(\mathcal{H}) \leq VC(\mathcal{M}(\mathcal{H}))$, observe that if $\mathcal{H}$ can shatter m points $p_1, \ldots, p_m$ using predicates $P_1, \ldots P_{2^m}$ then $\mathcal{M}(\mathcal{H})$ can shatter $f(p_1), \ldots, f(p_m)$ using $\mathcal{M}(P_1), \ldots, \mathcal{M}(P_{2^m})$. The proof for the reverse inclusion, that is, $VC(\mathcal{H}) \geq VC(\mathcal{M}(\mathcal{H}))$ is similar and uses g and $\mathcal{M}^{-1}$ (which exists since $\mathcal{M}$ is one to one). $\square$

This result allows us to compute the VC dimension of richer hypothesis classes and boolean combinations of the same. For example, if we have a numerical variable z and an array a as our program variables, then by the results in Section 3.1 and Theorem 3.6 the VC dimension of the class of predicates $\forall j.a[j] + c_1 z \leq \pm c_2$ is three and for the class of conjunctions of predicates of the form $\forall j.a[j] \geq \pm c$ and $z \geq \pm c$ is four.

3.5 Separation Logic

For heap manipulating programs, separation logic has emerged as a successful approach [44]. We do not review all the details of separation logic here and keep the discussion at an abstract level. The predicates or the elements of the hypothesis class are formulas written in separation logic and elements of the instance space or the program states are pairs of a store s and a heap h. Since we focus on the heap, we keep the store fixed. Our instance space is composed of program states in which the store maps a single variable z to a heap location l.

In [10], a heuristic algorithm for inference with lists has been described. The related fragment shown below has an infinite VC dimension.

$$\begin{array}{lll} e & ::= & x \mid X \mid c \\ p & ::= & \texttt{emp} \mid e_1 \mapsto e_2, e_3 \mid p * q \mid \mathit{list}\ e \end{array}$$

An expression is either a variable x, or a logical variable X, or a constant c (e.g. `nil`) and an assertion says that the heap is either empty, or it contains one cons cell and e_1 is the address of the cons cell with contents e_2 and e_3, or the assertion has a separating conjunction ($p*q$) that decomposes the heap into two disjoint parts, one where p is true and one where q is true. A *list* denotes a `nil` terminated singly linked list. The logical variables in the assertions are implicitly existentially quantified.

Theorem 3.7. *VC dimension of the above class of predicates is infinite.*

Bench	PK/OCT			PK/BOX			OCT/BOX		
	$\subsetneq$	$\supsetneq$	unc.	$\subsetneq$	$\supsetneq$	unc.	$\subsetneq$	$\supsetneq$	unc.
a2ps	12.74	0.78	0	21.64	0	2.13	18.94	0	0.93
gawk	21.34	0	0	26.96	0	0	17.97	0	0
chess	5.99	5.78	2.47	12.67	3.68	2.24	14.87	0	0
gnugo	18.75	2.08	2.08	22.50	1.66	1.11	10.86	0	1.12
grep	3.30	0	0	8.26	0	0	8.26	0	0
gzip	21.16	2.18	0	32.84	0.72	1.45	26.27	0	0
lapack	11.84	5.67	0.85	78.96	2.16	2.99	85.03	0	0
make	6.50	4.00	5.50	6.52	4.34	5.97	11.94	0	0
tar	5.17	4.20	0	9.70	3.23	0.97	9.38	0	0

Figure 9. Results of comparing abstract domains published in [29].

Proof. We need to show that given any n, we can construct n heaps and shatter them using the predicates of our logic. We give a proof sketch. The intuition is that we can introduce an unbounded number of logical variables in the heap and lists can encode a boolean choice. Therefore, we can make an unbounded number of boolean choices and shatter unbounded sets. Suppose n is two. The construction below can be generalized to arbitrary n. Recall, the store maps z to l. Consider two heaps, $h_1 = [l \mapsto (l_1, \texttt{nil}), l_1 \mapsto (\texttt{nil}, \texttt{nil})]$ and $h_2 = [l \mapsto (\texttt{nil}, l_1), l_1 \mapsto (\texttt{nil}, \texttt{nil})]$. The predicates $z \mapsto (X, Y)$, $z \mapsto (X, Y) * \mathit{list}(X)$, $z \mapsto (X, Y) * \mathit{list}(Y)$, and $z \mapsto (X, Y) * \mathit{list}(X) * \mathit{list}(Y)$ shatter the four labelings, $(\mathit{false}, \mathit{false}), (\mathit{true}, \mathit{false}), (\mathit{false}, \mathit{true}), (\mathit{true}, \mathit{true})$, of (h_1, h_2) repectively. In general, by introducing n logical variables we can shatter n heaps obtained using this construction. $\square$

Similar to Section 3.2, we can restrict the size of the predicates to bound the VC dimension and hence the generalization error. We are unaware of any template-based analysis that restricts the structure of the predicates in separation logic, but given the success of template-based invariant inference for numerical programs, it seems to be a useful research direction to pursue in the future.

4. Discussion

In this section, we consider empirical results from various papers on program analysis and try to interpret these results using bias-variance tradeoffs. We do not claim to be exhaustive; the goal here is to show that a variety of useful techniques can be justified using our framework.

4.1 Abstract Interpretation

Consider the subset of results (Figure 9) obtained from the PAGAI static analyzer that were published in [29]. The columns in the table of Figure 9 compare the quality of invariants found by polyhedra (PK) and octagons (OCT), polyhedra (PK) and intervals (BOX), and octagons (OCT) and intervals (BOX) respectively. In each comparison there are three values: the first value is the percentage of invariants for which the results of the first abstract domain are logically stronger than the second; the second value is the percentage of invariants for which the results of the second domain are stronger than the first; the third is the percentage of incomparable invariants; the remaining percentage of invariants are logically identical.

It can be seen from Figure 9 that as precision increases the quality of the inferred invariants gets better: this result is expected as richer abstract domains can express invariants that weaker domains cannot. We share the observation with the authors that for a non-negligible percentage of invariants, polyhedra perform worse than intervals and octagons. Other evaluations in [29] (not studied here) show that the basic forward analysis of [19] can produce better results than the "path-focused" approach of [30, 40], even though the latter can produce more precise intermediate results. The authors explain these observation using the non-monotonicity of the

Outcome	SatAbs	Magic	Blast	Blast (new)
Verified	0	0	8	12
Refinement failed	13	13	0	0
Did not finish	0	0	5	1

Figure 10. Results for interpolation with incremental increase of precision published in [32]. SatAbs [14] and Magic [12] are based on weakest preconditions and Blast [31] uses interpolants.

widening operator. In abstract interpretation, the widening operator is responsible for generalization [17]. Widening is usually non-monotonic because widening more precise information can lead to worse generalizations [18]. For example, $[0, 1]\nabla[0, 2] = [0, \infty]$ and $[0, 2]\nabla[0, 2] = [0, 2]$. In this example, widening more precise intervals leads to less precise results.

Non-monotonicity of the widening operator is related to bias-variance tradeoffs. Intuitively, if there were a monotonic widening operator, then improving the precision of the underlying abstract domain would lead to better generalizations, consequently, there would be no bias-variance tradeoff. The tradeoff seems to be a fundamental limit on generalization, and therefore monotonic widening operators for sophisticated abstract domains seem unlikely.

4.2 Interpolants

Interpolant-based engines [31, 38] find simple proofs of infeasibility of a finite number of spurious counterexample paths and hope that because the proofs are simple the predicates used in the proof will generalize and refute all possible spurious counterexamples. Inability to find good predicates can result in divergence in CEGAR. We motivate the usefulness of simple proofs using bias-variance tradeoffs.

An interpolant is required to prove the infeasibility of a spurious counterexample. This requirement ensures that the interpolant is not too weak—it must be strong enough to prove a potentially useful fact. This requirement can also be seen as a means to tackle underfitting: we are imposing a lower bound on the precision of the language of interpolants. To avoid overfitting, we want the interpolant to be simple. The hope is that simple predicates will not overfit to a specific path and hence can avoid high variance. The definition of an interpolant ensures simplicity by restricting the variables that can occur in an interpolant, which corresponds to lowering the dimension of the instance space, and consequently the VC dimension as well.

However, it has been observed that only restricting the variables is not enough to avoid divergence. In [32], the language of interpolants or the hypothesis class is restricted to a finite set and this set is expanded gradually. If the hypothesis class is finite, then in our framework the following result is known:

Theorem 4.1. *If $\hat{h}$ is obtained from an ERM algorithm, $|\mathcal{H}| = k$, m is the size of the sample set, and δ is fixed, then with probability at least $1 - \delta$ we have*

$$\varepsilon(\hat{h}) \leq (min_{h\in\mathcal{H}}\varepsilon(h)) + 2\sqrt{\frac{1}{2m} log \frac{2k}{\delta}}$$

This theorem states that when the language is inexpressive, we have high bias, and as we increase the expressiveness of the language under consideration, $|\mathcal{H}|$ increases, and so does the variance. Similar to Theorem 2.2, by gradually increasing the size of the hypothesis class, the bounds on the generalization error can be minimized.

Consider the empirical results of Jhala and McMillan published in [32] and shown in Figure 10. SatAbs, Magic, and Blast are predicate abstraction based tools that use CEGAR [13] and add predicates during their analysis to perform refinement. A poor choice of refinement predicates can cause divergence. These predicates are obtained by refuting spurious counterexample paths. If the predicates overfit the paths, then they are typically not useful for finding an invariant. The last column shows the results obtained from the following strategy for generating predicates: use interpolants as refinement predicates and first restrict the interpolants to a finite language L_0. For example, L_0 can be the language of predicates that have their numeric constants restricted to either zero or $c \pm 0$ where c is a numerical constant statically occurring in the program. The language is incrementally expanded to L_1, L_2, and so on. The penultimate column just uses the interpolants in the language L_∞. We observe that by incrementally increasing the expressiveness of the language of interpolants, one can avoid the overfitting present in L_∞ and obtain better results, verifying 12 instead of 8 programs from the benchmark suite of 13 programs.

4.3 Incrementally Increasing Precision

In recent years, there has been a growing interest in exploring optimal abstractions [37, 49]. These papers argue for the most imprecise abstraction that is sufficient to prove the desired property of a program. (This line of work is different from the techniques that aim to compute the least fixed point [22, 23].) Generally, an increase in precision is associated with a decrease in efficiency. Imprecise abstractions are computationally cheap and hence are desirable. Liang et al. [37] show that quite imprecise abstractions can be sufficient to prove most properties of interest. Zhang et al. [49] have an abstraction refinement algorithm that successively tries abstractions of increasing cost. Stratified analysis [41] is another technique that incrementally increases precision: abstract interpretation is performed in a stratified fashion, running successive analyses of increasing precision by incrementally increasing the number of program variables under consideration. The later analyses use the results of the previous analyses and heuristics based on dataflow dependencies. By performing the stratified analysis, the authors obtain the same or better invariants than classical [19] or alternative widening schemes [3] for all the cases of their study [41]. All of these, including [32] (discussed in Section 4.2), can be seen as examples of addressing bias-variance tradeoffs by starting with low precision and moving to high precision. It is also possible to achieve the same effect by starting from high precision and moving towards lower precision (e.g., as done in [5]). In the next section, we explore whether the techniques for combating bias-variance tradeoffs in machine learning can also benefit program analysis tools.

5. Cross Validation

In this section, we describe our experience with applying techniques for addressing bias-variance tradeoffs to program analysis tools. In particular, we discuss the application of cross validation to Yogi, a verification engine in Windows SDV [25].

Machine learning tools generally have a number of precision knobs and finding a configuration of these knobs that does not overfit the training set is recognized as a problem [11, 21]. One widely used solution is cross validation [2]. While providing formal guarantees for cross validation algorithms is a topic of current research [2, 33], cross validation has been found to be extremely useful empirically and is standard practice in machine learning [21]. We study the simplest cross validation algorithm here. Evaluation of more sophisticated variants such as k-fold cross validation [2] is left as future work.

Cross validation partitions the training set into *training data* and *test data*. Next, multiple learning algorithms, called *learners*, are trained on the training data and tested on the test data. The learner that generates the hypothesis performing best on the test data is selected. For example, for the regression example of Section 2.1,

we would consider a subset of the observations as training data and the rest as test data (say a 70-30 split). Now consider different learning algorithms where each algorithm fits a polynomial of a different degree on the training data. We find the best fit line, quadratic, cubic, etc., for the training data and pick the degree which corresponds to the hypothesis that performs the best on our test data.

We observe that the learner obtained from cross validation generates a hypothesis using training data that generalizes to test data. If we train a learner on the full training set without performing cross validation, then we might overfit. In Figure 3, the curve that corresponds to the fifth degree polynomial curve best fits the observations (among linear, quadratic, and fifth degree polynomials) and thus performs the best on the observed data. During cross validation, the fifth degree polynomial overfits on the training data and shows large deviations from the observations in the test data. Thus, cross validation can serve as a guide which rejects fifth degree polynomials for this example and prevents overfitting.

Here is a full description of cross validation:

1. Randomly split the training set S into S_{train} and S_{test}.
2. For each learner M_i, train it on S_{train} to get a hypothesis h_i. Training can be just ERM over S_{train}.
3. Pick the learner M_k corresponding to h_k with smallest error on S_{test}. Train M_k on S to obtain the output hypothesis h.

Note that once we have selected the learner M_k that generalizes best to the test data, the final hypothesis is computed by training M_k on all of the available data. We remark that cross validation is not a silver bullet and if used improperly can itself start to overfit [42].

YOGI is an industrial strength tool for checking Windows device driver properties. YOGI has a benchmark suite of 2490 driver-property pairs and in the current production version its precision knobs have been tuned to perform the best on these benchmarks [43]. As discussed above, this process can lead to overfitting; in fact, we show that simply by using cross validation we can significantly improve YOGI's performance.

In our setting, the instances are input programs that YOGI analyzes and the hypothesis class consists of all the versions of Yogi that can be created by different choices of configuration parameters. We are given some benchmarks and these constitute our training set. The labels for these benchmarks, correct or buggy, are known. The goal of the learner is to find a hypothesis (which is a tool) that generalizes well, that is, even for new programs it has not seen, we want it to assign the correct label. For YOGI, the learner is simply ERM, which selects the best parameter configuration for YOGI on the training set.

We consider a sequence of tools $Yogi_i$ where i, the number of *test steps*, is one of the most important parameters in YOGI (see [43] for details). The actual details of how YOGI works are unimportant; what is pertinent here is that a higher i corresponds to increased precision. In Figure 2, the timing results are shown where each $Yogi_i$ has been trained on the full benchmark set: the parameters (other than i) have been tuned to obtain the best results. We observe the expected bias-variance tradeoff curve. As i is increasing, the precision is increasing, and the results first improve with increasing precision and then they degrade. From Figure 2, the best value of i for all 2490 driver-property pairs is 500. Note that [43] uses the total runtime as a performance metric. Hence, we also use total runtime for comparing performance of different configurations during cross validation. However, more general performance metrics are certainly possible. We observe that the authors of [43] have discovered a good configuration of the tool, which we call the old YOGI; old YOGI is $Yogi_{500}$ trained on the

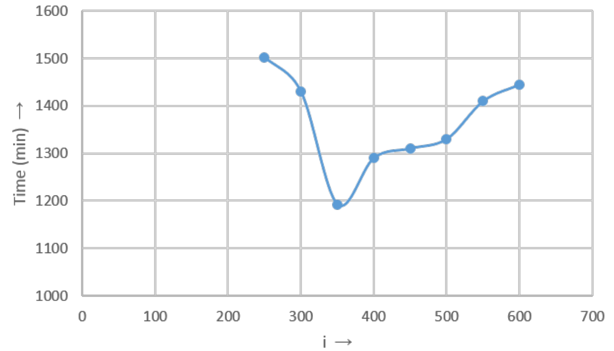

Figure 11. Result of running Yogi on 747 driver-properties test pairs with $i \in [250, 600]$.

Tool	Time (min)	#defects	#time-outs
$Yogi_{350}$	4704	189	21
$Yogi_{500}$	8279	183	66

Table 1. Performance of $Yogi_{350}$ and $Yogi_{500}$ on 2106 new driver-property pairs. The time-out was set to 30 minutes.

full set of 2490 driver-property pairs and is the current production version.

Now we consider the alternative, performing cross validation on YOGI. We split the 2490 driver-property pairs into 70% training data (consisting of 1743 driver-property pairs), and 30% test data (consisting of 747 driver-property pairs). We train each $Yogi_i$ (finding values for the parameters other than i) so that it performs as well as possible on the training data. Next, we pick the $Yogi_k$ that performs best on the test data (shown in Figure 11). From Figure 11, this best value occurs for $i = 350$. Next we train $Yogi_{350}$ on all 2490 driver-property pairs to obtain the new YOGI.

When verification is performed on new verification tasks (in this case, a new set of 2106 driver-property pairs), we observe that the new YOGI (with $i = 350$) shows better generalization properties than the old YOGI (with $i = 500$). Even though the new YOGI performs worse on the 2490 driver-property pairs (the old YOGI was the best configuration), it actually performs better on the new verification tasks, as shown in Table 1. We conclude that old YOGI was overfitted to the benchmarks. Here we have been able to reduce the run time by more than 40%, find new defects, and decrease the number of time-outs simply by using cross validation to set configuration parameters.

Here are some additional details about Table 1. The new YOGI and the old YOGI find 182 defects in common. The new YOGI timed out on one defect which the old YOGI could find. Also, there are 19 time-outs that are common between the old and the new YOGI. Such a result is expected as there is no best configuration. Using cross validation we are trying to find a configuration that avoids overfitting and is expected to work well in most cases.

To summarize, we empirically validate that YOGI has been overfitted to its benchmark suite and we remove this overfitting by cross validation to obtain a better tool. We believe other existing tools could be similarly improved with the straightforward application of cross validation.

6. Recommendations and Limitations

In this section, we consider the implications of bias-variance tradeoffs in the design of automatic program analyses. We also discuss

some limitations of our approach and make a number of specific recommendations.

Bias-variance tradeoffs can help in the selection of an abstract domain. One should avoid abstract domains with infinite VC dimension, otherwise we cannot bound the generalization error in our framework (Theorem 2.2) and we expect it to be large in practice. Ideally, one should use domains with finite VC dimension while ensuring the domain is rich enough to express the invariants of interest (to avoid large generalization error due to bias). One natural way to achieve this goal is to start with an imprecise domain and increase precision by gradually expanding the hypothesis class (Section 4).

We now discuss how bias-variance tradeoffs apply to fully automatic verification. Most practical tools have benchmarks and some parameters to tune. Take for instance the Astrée tool [20], which has more than a hundred parameters governing the abstract domains to enable, widening strategies, and so on. The best choice for the parameters is not usually clear. If one aims for a fully automatic tool like YOGI [25], then before shipping the tool one generally performs ERM: the parameters are tuned to give best results on a benchmark suite [43]. But if bias-variance tradeoffs exists, then this strategy is suboptimal. One can overfit to the benchmarks and unknowingly hamper the performance of the tool on new inputs (Section 5).

A common practice in the program analysis community is to tune to a benchmark suite and then report results [25, 43]. Given the results in this paper, it is clear that this approach provides no protection against overfitting and is therefore methodologically weak. As an extreme example, one can build a tool for any benchmark suite by simply creating a lookup table with the correct answer for each benchmark. This tool works perfectly in the reported experimental results and fails completely in practice. A more realistic scenario is that a tool builder discovers her tool gives poor results on some program P in her benchmark suite. She makes a change and as a result the tool works better on P. But has the tool really improved in general, or has the tool designer overfit to P? It might be that the tool's users were better off without the change. A tool that performs poorly on some of its benchmarks is not a bad thing if the alternative is overfitting.

The goal, of course, should be to build tools that work on previously unseen examples, not just on the benchmark suite. An experimental methodology that divides inputs into a training set that can influence the design of the tool and a separate test set that cannot inform the design of the tool is one way to validate that tools generalize beyond the benchmarks used to design them. Some researchers may be doing this already, but we are unaware of anyone mentioning this point explicitly in the literature. Note that this recommendation might not be directly applicable to tools that are not designed to be fully automatic. For example, Astrée [20] relies on user interaction to select the appropriate parameters for the program under analysis.

Despite our general critique of overfitting static analysis tools to benchmarks, there are situations in which that may be the right thing to do. When the goal is to analyze a specific program (e.g., the verification of seL4 [35]) rather than to build a tool that works for any program, it might make sense to overfit to the one important benchmark. The situation is also different for analysis algorithms that can be proven analytically to give an optimal or best result, such as certain classes of type inference, dataflow analysis, abstract interpretation over finite lattices, and the analysis of loop-free and recursion-free programs. Here the setting is sufficiently tractable that the generalization strategy is provably the best possible (or generalization is not needed at all) and reporting how well the algorithm works on a single full benchmark suite is a reasonable practice. Finally, if one had some guarantee that a benchmark suite was representative of all possible inputs there would be no need for a separate test suite—in this case improving performance on the training set guarantees improvement in general. However, it seems difficult to obtain or prove that a benchmark suite is representative, even for a particular domain. The drivers we consider as the training set in Section 5 are a superset of the benchmarks of [25] and from our results we can conclude that they are not representative.

The competition on software verification [4] was introduced as a common platform to compare tools. However, the current structure of the competition does not avoid overfitting. The benchmarks and the expected results of the competition are public and the benchmarks used to compare tools are a *subset* of the publicly released benchmarks. This contest design addresses a real concern, which is that because the semantics of C is underspecified and the benchmarks are C programs, publishing all benchmarks in advance helps ensure that the organizers and participants agree on the intended meaning of the programs. Our recommendation to organizers of competitions to evaluate software tools is that they should always evaluate the tools on some new, unseen programs as a check on overfitting. These unseen programs must be chosen carefully to ensure that any possible variations in interpretation are irrelevant to the verification task.

7. Conclusion

Because of bias-variance tradeoffs, increasing the precision of a program analysis can lead to a decrease in the quality of results. We have adapted the PAC learning framework to explain bias-variance tradeoffs in program analysis and used VC dimension as a measure of the precision of abstract domains. We have computed the VC dimension for some popular abstractions for numerical, array manipulating, and heap manipulating programs and we observe that more precise abstractions have higher VC dimension. We have also shown that standard techniques for addressing bias-variance tradeoffs, such as incrementally increasing precision and using cross validation in tuning parameters, are applicable to program analysis tools.

Acknowledgments

We thank Saurabh Gupta, Bharath Hariharan, Sriram Rajamani, and the anonymous reviewers for their constructive comments. This work was supported by NSF grant CCF-1160904. This material is also based on research sponsored by 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] G. Amato, M. Parton, and F. Scozzari. Discovering invariants via simple component analysis. *J. Symb. Comput.*, 47(12):1533–1560, 2012.

[2] S. Arlot and A. Celisse. A survey of cross-validation procedures for model selection. *Statistics Surveys*, 4:40–79, 2010.

[3] R. Bagnara, P. M. Hill, E. Ricci, and E. Zaffanella. Precise widening operators for convex polyhedra. *Sci. Comput. Program.*, 58(1-2):28–56, 2005.

[4] D. Beyer. Second competition on software verification - (summary of SV-COMP 2013). In *TACAS*, pages 594–609, 2013.

[5] D. Beyer, T. A. Henzinger, and G. Théoduloz. Program analysis with dynamic precision adjustment. In *ASE*, pages 29–38, 2008.

[6] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics).* Springer-Verlag New York, Inc., 2006. ISBN 0387310738.

[7] N. Bjørner, K. L. McMillan, and A. Rybalchenko. On solving universally quantified horn clauses. In *SAS*, pages 105–125, 2013.

[8] A. Blumer, A. Ehrenfeucht, D. Haussler, and M. K. Warmuth. Learnability and the Vapnik-Chervonenkis dimension. *J. ACM*, 36(4):929–965, 1989.

[9] N. H. Bshouty, S. A. Goldman, H. D. Mathias, S. Suri, and H. Tamaki. Noise-tolerant distribution-free learning of general geometric concepts. *J. ACM*, 45(5):863–890, 1998.

[10] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In *POPL*, pages 289–300, 2009.

[11] G. C. Cawley and N. L. C. Talbot. On over-fitting in model selection and subsequent selection bias in performance evaluation. *Journal of Machine Learning Research*, 11:2079–2107, 2010.

[12] S. Chaki, E. M. Clarke, A. Groce, and O. Strichman. Predicate abstraction with minimum predicates. In *CHARME*, pages 19–34, 2003.

[13] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV*, pages 154–169, 2000.

[14] E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate abstraction of ansi-c programs using sat. *Formal Methods in System Design*, 25(2-3):105–127, 2004.

[15] M. Colón, S. Sankaranarayanan, and H. Sipma. Linear invariant generation using non-linear constraint solving. In *CAV*, pages 420–432, 2003.

[16] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *ISOP*, pages 106–130, 1976.

[17] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[18] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In *PLILP*, pages 269–295, 1992.

[19] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, pages 84–96, 1978.

[20] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Why does Astrée scale up? *Formal Methods in System Design*, 35(3): 229–264, 2009.

[21] P. Domingos. A few useful things to know about machine learning. *Commun. ACM*, 55(10):78–87, 2012.

[22] T. Gawlitza and H. Seidl. Precise fixpoint computation through strategy iteration. In *ESOP*, pages 300–315, 2007.

[23] T. Gawlitza and H. Seidl. Precise relational invariants through strategy iteration. In *CSL*, pages 23–40, 2007.

[24] S. Geman, E. Bienenstock, and R. Doursat. Neural networks and the bias/variance dilemma. *Neural Computation*, 4(1):1–58, 1992.

[25] P. Godefroid, A. V. Nori, S. K. Rajamani, and S. Tetali. Compositional may-must program analysis: unleashing the power of alternation. In *POPL*, pages 43–56, 2010.

[26] B. S. Gulavani and S. Gulwani. A numerical abstract domain based on expression abstraction and max operator with application in timing analysis. In *CAV*, pages 370–384, 2008.

[27] S. Gulwani, S. Srivastava, and R. Venkatesan. Program analysis as constraint solving. In *PLDI*, pages 281–292, 2008.

[28] A. Gupta, R. Majumdar, and A. Rybalchenko. From tests to proofs. In *TACAS*, pages 262–276, 2009.

[29] J. Henry, D. Monniaux, and M. Moy. Pagai: A path sensitive static analyser. *Electr. Notes Theor. Comput. Sci.*, 289:15–25, 2012.

[30] J. Henry, D. Monniaux, and M. Moy. Succinct representations for abstract interpretation - combined analysis algorithms and experimental evaluation. In *SAS*, pages 283–299, 2012.

[31] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *POPL*, pages 232–244, 2004.

[32] R. Jhala and K. L. McMillan. A practical and complete approach to predicate refinement. In *TACAS*, pages 459–473, 2006.

[33] M. Kearns and D. Ron. Algorithmic stability and sanity-check bounds for leave-one-out cross-validation. *Neural Computation*, 11:152–162, 1997.

[34] M. J. Kearns and U. V. Vazirani. *An introduction to computational learning theory*. MIT Press, Cambridge, MA, USA, 1994. ISBN 0-262-11193-4.

[35] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: formal verification of an operating-system kernel. *Commun. ACM*, 53(6):107–115, 2010.

[36] G. Lalire, M. Argoud, and B. Jeannet. The Interproc Analyzer. http://pop-art.inrialpes.fr/people/bjeannet/bjeannet-forge/interproc/index.html.

[37] P. Liang, O. Tripp, and M. Naik. Learning minimal abstractions. In *POPL*, pages 31–42, 2011.

[38] K. L. McMillan. An interpolating theorem prover. *Theoretical Computer Science*, 345(1):101–121, 2005.

[39] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[40] D. Monniaux and L. Gonnord. Using bounded model checking to focus fixpoint iterations. In *SAS*, pages 369–385, 2011.

[41] D. Monniaux and J. L. Guen. Stratified static analysis based on variable dependencies. *Electr. Notes Theor. Comput. Sci.*, 288:61–74, 2012.

[42] A. Y. Ng. Preventing "overfitting" of cross-validation data. In *ICML*, pages 245–253, 1997.

[43] A. V. Nori and S. K. Rajamani. An empirical study of optimizations in YOGI. In *ICSE (1)*, pages 355–364, 2010.

[44] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74, 2002.

[45] S. Sankaranarayanan, H. B. Sipma, and Z. Manna. Scalable analysis of linear systems using mathematical programming. In *VMCAI*, pages 25–41, 2005.

[46] S. Sankaranarayanan, F. Ivancic, I. Shlyakhter, and A. Gupta. Static analysis in disjunctive numerical domains. In *SAS*, pages 3–17, 2006.

[47] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, and A. V. Nori. Verification as learning geometric concepts. In *SAS*, pages 388–411, 2013.

[48] L. G. Valiant. A theory of the learnable. *Commun. ACM*, 27(11): 1134–1142, 1984.

[49] X. Zhang, M. Naik, and H. Yang. Finding optimum abstractions in parametric dataflow analysis. In *PLDI*, pages 365–376, 2013.