Validating Programs with Examples
A Theoretical Exploration

Philip Zigoris*
CMPS203 Final Project
University of California, Santa Cruz
December 2, 2004

Abstract

A major issue in software engineering is how one can demonstrate the correctness of a program. Formal methods exist but we often opt for a more informal approach: testing examples. This paper offers some theoretical analysis of this approach showing that in some situation it is appropriate, and others it is seemingly futile. In particular we analyze programming languages that include basic arithmetic, boolean, and imperative expressions.

1 Introduction

The aim of this project is to theoretically quantify the assurance that software testing gives us about program correctness. If we know our program performs correctly on some set of inputs, what can be said about the correctness of the program on all inputs? And how much does this depend on the programming language itself?

The approach here is to first formalize the complexity of a program, using the VC dimension. Then we can apply results from about uniform convergence to yield probabilistic statements about the program’s correctness.

We consider a number of simple variations of the IMP language, which include arithmetic, boolean, and imperative commands and give bounds on their complexity.

2 Preliminaries

In this section we introduce a well studied notion of function class complexity, the Vapnik-Chervonenkis Dimension and describe its characteristic properties.

2.1 VC Dimension

Let’s begin this discussion by thinking about programs that take two real values and return a boolean. To visualize the behavior of a particular program of this form we might color in the Cartesian plane according to which points will return a true and false. We might think about translating the notion of language complexity to the complexity with which that language, not just a specific program, can ‘color’ the input space. This is, in a high level geometric sense, what the Vapnik-Chervonenkis Dimension (VCD) captures.

Consider a set of functions, $H$, that map some set $X$ (our input space) to $\{0, 1\}$. The set of functions representable by our programming language is one such family. We might also think of $H$ as the indicator functions for a family of subsets of $X$; the two definitions are interchangeable. The literature will often refer to $H$ as a hypothesis or concept class. To begin to formalize our notion of complexity we give the following definition.

Definition The growth function of a hypothesis class $H \subseteq 2^X$ is

$$\Pi_H(m) = \max |H_S| : S \subseteq X \text{ and } |S| = m$$

where $H_S = \{h \cap S : h \in H\}$ for a set $S \in X$.
The growth function tells us the maximum number of ways you can label a sample of $X$ of size $m$. We say that a set $S$ is shattered by $H$ if $|H_S| = 2^{|S|}$ and clearly $\forall m \Pi_H(m) \leq 2^m$. But it is easy to think of simple hypothesis classes where this upper bound is overzealous. For example, if our input space was $\mathbb{R}$ and our hypothesis class consisted of closed intervals on the real line. A point is labeled 1 if it is in the interval, 0 if not. For any three real numbers $a < b < c$ there does not exist an interval $I$ such that $a, c \in I$ and $b \notin I$ and consequently $\Pi(3) = 7$.

We can now define the VCD of a hypothesis class.

**Definition** The Vapnik-Chervonenkis Dimension (VCD) of a hypothesis class $H \subseteq 2^X$ is the largest $m$ such that $\Pi_H(m) = 2^m$. If no such maximum exists then we say that the $VCD(H) = \infty$.

This is a well studied notion of complexity and yields strong results about the convergence of empirical and true error. Before going on to discuss those results some examples are given to further illustrate the concept:

- The hypothesis class of intervals on the real line, described above, has $VCDIM(\text{INTERVAL}) = 2$.

- Consider the hypothesis class $\text{CIRCLE}$. Let $h_{x,y,r} \in \text{CIRCLE}$ have the following behavior: $h_{x,y,r}(a,b) = 1$ if $(a,b)$ is contained in the circle of radius $r$ centered at $(x,y)$, 0 otherwise. $VCD(\text{CIRCLE}) = 3$.

- Consider the hypothesis class $\cup \text{CIRCLE}$ which allows the regions bounded by any number of circles to be unioned: $VCDIM(\cup \text{CIRCLE}) = \infty$.

We provide one more result due to Sauer [7] that polynomially bounds the growth function in terms of the VCD.

**Theorem 2.1** If $H$ has finite VC dimension of $d$ then $\Pi_H(m) \leq m^d$.

### 2.2 Uniform convergence

Suppose we are trying to estimate some function $\hat{h}$. Given some sample of input/output pairs we might guess some function $\hat{h}$. We can calculate the error of $\hat{h}$ on the provided sample and ask how well this sample error converges on the true error, over all of $X$. The reason we chose to use the VC dimension in this project is it precisely characterizes this convergence.

The first theorem is originally due to Vapnik [8]. We present an interpretation of the form given by Anthony and Bartlett [2]

**Theorem 2.2** Given a hypothesis class $H \subseteq 2^X$ and some size $m$ sample $S_f = \{(x,y) : x \in X, f(x) = y\}$, labeled according to some $\{0,1\}$ valued function $f$, drawn according some distribution $P$ on $X$, $0 < \epsilon < 1$

$$P(|er^f_P(\hat{h}) - er_{S_f}(\hat{h})| \geq \epsilon, \hat{h} \in H) \leq 4\Pi_H(2m)e^{-\frac{2m}{\epsilon^2}}$$

where

$$er^f_P(\hat{h}) = Pr_P(\hat{h}(x) \neq f(x))$$

$$er_{S_f}(\hat{h}) = \frac{1}{m} \sum_{x \in S} I_{\{h(x) \neq f(x)\}}$$

which correspond, respectively, to the true and sample error of $\hat{h}$.

This theorem tells us that for an arbitrary degree of precision $\epsilon$ we can bound the probability that the true and sample error of our guess differ by more than $\epsilon$. Theorem 2.1 gives us a polynomial bound on $\Pi_H$ which means that as we test our hypothesis on more and more samples, this bound decreases exponentially fast to zero.

The following theorems, based on results presented by Anthony and Bartlett [2], demonstrate how the VC dimension characterizes uniform convergence. They tell us how many samples are needed to guarantee with $(1 - \delta)$ confidence that our estimation of the error does not deviate from the true error by more than $\epsilon$.

**Theorem 2.3** If $H$ has finite VC dimension $d$ then for any distribution $P$ on $X$, $f : X \leftarrow \{0,1\}$, $\hat{h} \in H$, there exists a function $\epsilon(m, \delta, d) \in \Theta(\sqrt{\frac{1}{m} \ln \frac{1}{\delta}})$ such that for a size $m$ random sample $S_f$:

$$P(|er^f_P(\hat{h}) - er_{S_f}(\hat{h})| \leq \epsilon(m, \delta)) \geq 1 - \delta$$
We can give more specific bounds on the number of samples required for precision $\varepsilon$ based on the VC dimension of $H$. In particular, if $m \geq d/2$ then

$$
\varepsilon(m, \delta, d) = \left( \frac{32}{m} \left( \frac{d}{\ln \left( \frac{2em}{d} \right)} + \ln \left( \frac{4}{\delta} \right) \right) \right)^{1/2}
$$

A geometric interpretation of these results is shown in Figure 1. The shaded regions represented those points that our hypothesis $h$ differs from the actual function $f$. If we choose a random test point $x \in X$ the probability that $h(x) \neq f(x)$ corresponds to the area in the shaded region. The results above tell us that this region can be made arbitrarily small with arbitrary confidence. The reason we cannot offer a result with 100% confidence is that there is always a chance of getting an unrepresentative sample. For instance, if we were to randomly draw the points labeled in Figure 1 then our sample error of 0 would not agree with the true error.

3 Complexity of Languages

Let us take a moment and translate the results of the previous section to the context of programming languages. The programmer aims to represent some function $f$ in some language. After some effort, a guess (albeit a very intelligent one) $\hat{h}$ is produced. The program running successfully on $m$ examples corresponds to a sample error of 0. Now, let’s assume that we know the VC dimension of the language used is $d$. Although we cannot give a result of the form “The probability that the program is correct is ...” we can give the following theorem.

**Theorem 3.1** Suppose program $A$ written in language $L$ is tested on $m$ examples and $VCD(L) = d$, then the probability that

$$
Pr(A \text{ fails on } r \text{ runs}) \leq 1 - (1 - \varepsilon(m, \delta, d))^r
$$

is at least $1 - \delta$

**Proof** From Theorem 2.3 we know that, with confidence $1 - \delta$, the probability that our program is wrong for some arbitrarily picked input is less than $\varepsilon(m, \delta, d)$. The probability of picking $r$ inputs that $A$ gets correct is therefore greater than $(1 - \varepsilon(m, \delta, d))^m$. So the probability that it gets at least one wrong is $1 - (1 - \varepsilon(m, \delta, d))^m$.

Whether or not this corresponds with program validity is a matter of debate. I would argue that such qualities are a necessary condition for being able to validate programs by example. Here we focus on characterizing languages with the VC dimension but other notions of complexity do exist and may provide better results where this analysis falls short.

We now turn to the task of finding the VC dimension of a language. First, we provide a more formal definition.

**Definition** For a language $L$, $VCD(L) = VCD(H)$ where $H = \{h : h \text{ can be represented by } L\}$

A disappointing observation that can be made immediately is that even a simple language such as IMP has infinite VC dimension. Given an arbitrary set of points of $m$ inputs we can construct a program, of length $O(m)$, that is essentially an exhaustive ‘switch’ statement. To handle this, we can constrain the “length” of our programs and then ask about the VC dimension of this class of programs.

We can then ask: what is an appropriate notion of “program length”? We could simply use the length of the string literal of our program. This
approach is appealing in that there is only a finite number of programs of length \( n \), providing an easy upper bound on the VCD. However, assuming that VCD grows with program size, the VCD of our program will represent less the inherent complexity of the language than, say, the precision with which we specify the numbers in our program.

Instead we identify the size of a program, Size(\( P \)), with the number of internal nodes in its abstract syntax tree (AST). While this seems more appropriate, we no longer have a finite number of programs of length \( n \). For instance, a subclass of length 2 programs are those of the form \( c + x \leq 0 \) where \( c \) is some arbitrary number and \( x \) is our input. There are infinitely many programs of this form since \( c \) can take on infinitely many values. But the VC dimension of this subclass is most certainly finite; it is, in fact, 1.

A fact from graph theory is that the number of internal nodes \( N(T) \) of a binary tree is greater than or equal to the number of leaves \( L(T) \) minus 1. Equality holds if the tree is full, but because we might have unary operators such as \(-\), \(=\), \(\odot\) = we will proceed using the inequality. In the AST for the languages we consider, internal nodes represent operations such as \(+\) and \texttt{if/then} whereas leaves represent references to stored variables, \texttt{parameters} (constants), and input values. From this we see that the number of parameters is always less than the program size. \(^1\) This is important for translating certain results to the context of this project. We could in a similar manner bound the number of inputs to our program but instead we allow it, along with program size, parameterize our language classes.

The rest of this section is devoted to characterizing the VC dimension of a number of simple languages.

3.1 Arithmetic Operations

The first language we consider is one containing simple arithmetic operations\((+,-,x,/)\), variable assignment and a simple return statement. Formally our language \texttt{ARITH}\(\( a := \ n \)

\[
\begin{align*}
  e_a & := n \\
  x & \\
  e_a + e_a & \\
  e_a * e_a & \\
  e_a - e_a & \\
  e_a/e_a & \\
  e_c & := e_c; e_c \\
  x & = e_a
\end{align*}
\]

We assume that the input to the program is specified in the initial state so we don’t give any special syntax for input. We also assume the semantics for the language are naturally defined.

What mathematical functions can we compute with this language? It is clear that we can compute any polynomial function and decide whether points lie in the region above and below its graph. We call these \texttt{polynomial threshold} functions. With some thought we see that, in fact, every program written in this language represents a polynomial threshold function. This is to our advantage because the VC dimension of such functions have been explicitly studied \cite{golberg-jerrum}. In particular, we cite a result given by Golberg, Jerrum \cite{golberg-jerrum}.

**Theorem 3.2** The VCD of a polynomial threshold function of degree \( d \) and parameterized by \( p \) values is less than or equal to \( 2p \log(8d) \).

Consider a subset of our language \texttt{ARITH}\(\( a := \ n \)

\[
\begin{align*}
  \text{program} & := e_c; \text{ return } e_a > 0
\end{align*}
\]

\( \text{Program of the form } (x * x)^n, \text{ where } x \text{ is an input value, effectively maximizes the degree of the associated polynomial, which is of degree } 2^n. \]

**Proof** A program of the form \( (x * x)^n \), where \( x \) is an input value, effectively maximizes the degree of the associated polynomial, which is of degree \( 2^n \).

The following theorem very simply follows from Lemma 3.3, Theorem 3.2, and the previously mentioned fact that a program of size \( n \) is parameterized by \( p < n \) values.
Theorem 3.4 The VCD(\text{ARITH}(n,k) \leq 2n(n + \log(8e))) and is therefore $O(n^2)$.

Referring back to 1 we can see that as we allow larger programs the number of samples necessary to achieve the same level of precision grows polynomially.

3.2 Boolean Operations

Naturally, the next step is add boolean expressions to our language. We extend \text{ARITH} to include the following

$$e_b := \begin{cases} e_a > e_a \\ e_a = e_a \\ e_b \land e_b \\ e_b \lor e_b \\ \neg e_b \end{cases}$$

$$e_c := \begin{cases} e_c; e_c \\ x = e_a \\ \text{return } e_b \end{cases}$$

$$\text{program} := e_c;$$

and call this language \text{BOOL}. Note that these boolean expressions effectively include \text{if/then/else} in our language.

Now our language is clearly capable of expressing something other than a polynomial threshold function but how does this affect the VC dimension. Again, we give a result due to Goldberg and Jerrum [6]

Theorem 3.5 Let $C(n,k)$ be a concept class expressible by a boolean combination of $s$ polynomial threshold functions, each of degree at most $d$, parameterized by at most $n$ values, and acting on inputs of dimension $k$. Then $VCD(C(n,k)) \leq 2n \log(8 \text{eds})$.

Using this results we offer the following

Theorem 3.6 The $VCD(\text{BOOL}(n,k)) \leq 2n(n + \log(8ne))$ and therefore is $O(n^2)$.

Proof We begin by accepting the claim that the programs in \text{BOOL} are expressible as boolean combinations of polynomial threshold functions. Lemma 3.3 gives the maximum degree of any such polynomial. And since there are at most $n$ operations in a program, we have that there at most $n$ of these functions being combined with boolean operators. Plugging $s \leq n$ and $d \leq 2^d$ into Theorem 3.5 completes the proof. 

So in an asymptotic sense, boolean values do not increase the complexity of our language.

3.3 IMP

The next logical step is for our language to allow looping. We modify the command expressions of \text{BOOL} to also include the statement

$$\text{while}(e_b)(e_c)$$

and, for completeness

$$\text{if}(e_b) \text{ then } (e_c) \text{ else } (e_c)$$

Now it seems the previous results do not apply. For instance, the program $f = 2; \text{while}(x > 0)(f = f + f; x = x - 1); \text{return } y > f$ is not a polynomial threshold function.

In our previous languages, the number of operations our program could perform directly corresponded to its size. Here, we can in some sense perform an arbitrary number of operations. Does this give us infinite VCD? I argue that it does, even though our program size is constrained and we therefore cannot simply offer the extended switch statements argument discussed earlier.

Consider a recursive checkerboard, illustrated in Figure 2. The following program tests if a point $(x,y)$ is in a shaded region of such a design when it is calculated to some depth $d$.

```plaintext
left=n1;
right=n2;
top=n3;
bottom=n4;
depth=n5;
while(depth>0)
    midpoint=left+(right-left)/2;
    if(x>midpoint) then
        left=midpoint;
```
Figure 2: The recursive checkerboard. Dark regions represent points that evaluate to 1.

def recursive_checkerboard(top, left, right, depth):
    if depth > 0:
        # Make a decision
        if ...:  # Add your decision-making logic here
            # Recursively call the function
            ...  # Add your recursive call here
    else:
        # Base case
        return ...

# Example usage
recursive_checkerboard(top, left, right, depth=3)

else
    right=midpoint;
    midpoint=bottom+(top-bottom)/2;
    if(y>midpoint) then
        bottom=midpoint;
    else
        top=midpoint;
        d=d-1;
    horizontal_midpoint=left+(right-left)/2;
    vertical_midpoint=bottom+(top-bottom)/2;
    if(x>horizontal_midpoint) then
        return y>vertical_midpoint
    else
        return y<vertical_midpoint

This program has 5 parameters, takes 2 input values, and is of fixed length 40. But I contend, with a firm waving of my hand, that this program can shatter an arbitrarily large set of inputs. We can do this because there is no bound on parameter n_s.

If this is not satisfying then consider a language that treats sin(x) as a constant time operation. Anthony and Bartlett prove that this class of functions has infinite VC dimension [2] (see Lemma 7.2). So while ARITH and BOOL offered some reasonably satisfying results, we see that simple operations one would expect in any practical programming language put that language outside the realm of our analysis.

There are two ways to respond to this dilemma. One is to consider a notion of complexity other than the VCD. For instance, if we consider the size of a program to be simply the length of its literal string representation then, as discussed before, we in some sense are bounding the size of the program’s parameters. In the recursive checkerboard example we then have a limit on the recursive depth of the program. We also mention margin based or scale sensitive notions of complexity, popular in neural network and support-vector machine theory (REFERENCES), which seems a more fruitful approach. [3]

Another way to respond is to ask what happens if we bound the number of times a while loop can iterate. Then we can essentially unravel any while statement to a bounded number of if statements and we can refer back to earlier results. And, in fact, if we upper bound the number of operations performed during the execution of our programs by t then we have that the VC dimension of our language is O(nt) [6].

4 Learnability

An interesting corollary to the analysis given is that languages like ARITH and BOOL are learnable [5]. Finiteness of VCD is a necessary and sufficient condition for learnability under the PAC-Theoretic model. Essentially this translates to: given a set of input/output pairs for a desired function, there exists some algorithm that can produce a program which approximates the desired function to an arbitrarily high degree of accuracy.

5 Conclusion

The aim here was to justify the simplest approach to software validation: testing on examples. We considered a range of programs and tried to characterize their complexity in terms of the VC dimension. We saw that for arithmetic and boolean expressions there is in fact some theoretical justification for software testing. Because the VC dimension of these languages grew reasonably well with respect to program size, we were able to give promising statements about the probability of program correctness.
6 Future Work?

A natural extension of this work is to consider programs that can return real values or even multidimensional values. I have encountered measures of complexity, e.g. pseudo-dimension, fat-shattering dimension [4], appropriate for this type of analysis but they are mathematically much more demanding.

7 Acknowledgments

Thanks to Prof. Cormac Flanagan for letting me pursue this project and to Prof. David Helmbold for technical insight and recommending references.

References


