Interpolants as Classifiers*

Rahul Sharma[1], Aditya V. Nori[2], and Alex Aiken[1]

[1] Stanford University, {sharmar, aiken}@stanford.edu
[2] Microsoft Research India, adityan@microsoft.com

Abstract. We show how interpolants can be viewed as classifiers in supervised machine learning. This view has several advantages: First, we are able to use off-the-shelf classification techniques, in particular support vector machines (SVMs), for interpolation. Second, we show that SVMs can find relevant predicates for a number of benchmarks. Since classification algorithms are predictive, the interpolants computed via classification are likely to be relevant predicates or invariants. Finally, the machine learning view also enables us to handle superficial non-linearities. Even if the underlying problem structure is linear, the symbolic constraints can give an impression that we are solving a non-linear problem. Since learning algorithms try to mine the underlying structure directly, we can discover the linear structure for such problems. We demonstrate the feasibility of our approach via experiments over benchmarks from various papers on program verification.

Keywords: Static analysis, interpolants, machine learning

1 Introduction

Problems in program verification can be formalized as learning problems. In particular, we show how interpolants [4,17,11] that are useful heuristics for computing "simple" proofs in program verification can be looked upon as classifiers in supervised machine learning. Informally, an interpolant is a predicate that separates good or positive program states from bad or negative program states and a set of appropriately chosen interpolants forms a program proof. Our main technical insight is to view interpolants as classifiers that distinguish positive examples from negative examples. This view allows us to make the following contributions:

- We are able to use state-of-the-art classification algorithms for the purpose of computing invariants. Specifically, we show how support vector machines (SVMs) [21] for binary classification can be used to compute interpolants.
- Since classification algorithms are predictive, the interpolants we compute are relevant predicates for program proofs. We show that we can discover inductive invariants for a number of benchmarks. Moreover, since SVMs are routinely used in large scale data processing, we believe that our approach can scale to verification of practical systems.

* This work was supported by NSF grant CCF-0915766 and the Army High Performance Computing Research Center.

– Classification based interpolation also has the ability to detect superficial non-linearities. As shown in Sect. 4, even if the underlying problem structure is linear, the symbolic constraints can give an impression that we are solving a non-linear problem. Since our algorithm mines the underlying structure directly, we can discover the linear structure for such problems.

The rest of the paper is organized as follows. We informally introduce our technique by way of an example in Sect. 1.1. In Sect. 2, we describe necessary background material including a primer on SVMs. Sect. 3 describes the main results of our work. We first introduce a simple algorithm Basic that uses an SVM as a black box to compute a candidate interpolant and we formally characterize its output. SVMs rely on the assumption that the input is linearly separable. Hence, we give an algorithm SVM-I (which makes multiple queries to an SVM) that does not rely on the linear separability assumption and prove correctness of SVM-I. We augment Basic with a call to SVM-I; the output of the resulting algorithm is still not guaranteed to be an interpolant. This algorithm fails to output an interpolant when we do not have a sufficient number of positive and negative examples. Finally, we describe an algorithm Interpolant that generates a sufficient number of positive and negative examples by calling Basic iteratively. The output of Interpolant is guaranteed to be an interpolant and we formally prove its soundness. In Sect. 4, we show how our technique can handle superficial non-linearities via an example that previous techniques are not capable of handling. Sect. 5 describes our implementation and experiments over a number of benchmarks. Sect. 6 places our work in the context of existing work on interpolants and machine learning. Finally, Sect. 7 concludes with some directions for future work.

1.1 An Overview of the Technique

We show an example of how our technique for interpolation discovers invariants for program verification. Consider the program in Fig. 1. This program executes the loop at line 2 a non-deterministic number of times. Upon exiting this loop, the program decrements `x` and `y` until `x` becomes zero. At line 6, if `y` is not 0 then we go to an error state. To prove that the `error()` statement is unreachable, we need invariants for the loops. We follow the standard verification by interpolants recipe and try to find invariants by finding interpolants for finite infeasible traces of the program. The hope is that the interpolants thus obtained will give us predicates that generalize well. In particular, we aim to obtain an inductive loop invariant. For example, $x = y$ is a sufficiently strong loop invariant for proving the correctness of Fig. 1.

Suppose we consider a trace that goes through all the loops once. Then we get the following infeasible trace: (1, 2, 3, 2, 4, 5, 4, 6, 7). We decompose this trace into two parts A and B and thereby find interpolants for this infeasible trace. A represents the values of `x` and `y` obtained after executing lines 1, 2, and 3. B represents the values of `x` and `y` such that if we were to execute lines 4, 5, 6, and 7 then the program reaches the `error()` statement. Now, we have (A, B)

```
foo( )
{
1:  x = y = 0;
2:  while (*)
3:   { x++; y++; }
4:  while ( x != 0 )
5:   { x--; y--; }
6:  if ( y != 0 )
7:    error() ;
}
```

Fig. 1. Motivating example for our technique.

where $A \wedge B \equiv \bot$:

$$A \equiv x_1 = 0 \wedge y_1 = 0 \wedge ite(b, x = x_1 \wedge y = y_1, x = x_1 + 1 \wedge y = y_1 + 1)$$
$$B \equiv ite(x = 0, x_2 = x \wedge y_2 = y, x_2 = x - 1 \wedge y_2 = y - 1) \wedge x_2 = 0 \wedge \neg(y_2 = 0)$$

Here *ite* stands for if-then-else. As is evident from this example, A is typically the set of reachable states and B is the set of states that reach `error()`. An interpolant is a proof that shows A and B are disjoint and is expressed using the common variables of A and B. In this example, x and y are the variables common to A and B. Our technique for finding an interpolant between A and B operates as follows: First, we compute samples of values for (x, y) that satisfy the predicates A and B. Fig. 2 plots satisfying assignments of A as +'s (points $(0, 0)$ and $(1, 1)$) and of B as $\circ$'s (points $(1, 0)$ and $(0, 1)$). Next, we use an SVM to find lines separating the $\circ$'s from the +'s.

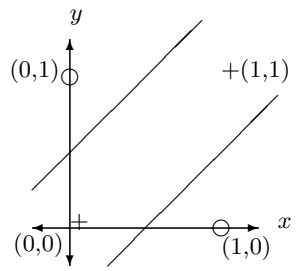

Fig. 2. Finding interpolants using an SVM.

We consider the $\circ$ points one by one and ask an SVM to find a line which separates the chosen $\circ$ point from the +'s. On considering $(0, 1)$, we get the line $2y = 2x + 1$ and from (1,0) we obtain $2y = 2x - 1$. Using these two lines, we obtain the interpolant, $2y \leq 2x + 1 \wedge 2y \geq 2x - 1$. It can be checked that this predicate is an invariant and is sufficient to prove the `error()` statement of Fig. 1 unreachable.

We will see in Sect. 2.1 that we can easily obtain the stronger predicate $x = y$. Intuitively, we just have to translate the separating lines as close to the +'s as possible while ensuring that they still separate the +'s from the ∘'s.

2 Preliminaries

Let A and B be two formulas in the theory of *linear arithmetic*:

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

$w = (w_1, \ldots, w_n)^T \in \mathbb{R}^n$ is a *point*: an n-dimensional vector of constants; $x = (x_1, \ldots, x_n)^T$ is an n-dimensional vector of variables. The *inner product* $\langle w, x \rangle$ of w and x is $w^T x = w_1 x_1 + \ldots + w_n x_n$. The equation $w^T x + d = 0$ is a *hyperplane* in n-1 dimensions. Each hyperplane *corresponds* to two *half-spaces*: $w^T x + d \geq 0$ and $w^T x + d \leq 0$. A half-spaces divides $\mathbb{R}^n$ into two parts: variable values that satisfy the half-space and those which do not. For example, $x - y = 0$ is a 1-dimensional hyperplane, $x - y + 2z = 0$ is a 2-dimensional hyperplane, and $x \geq y$ and $x \leq y$ are half-spaces *corresponding* to the hyperplane $x = y$.

Suppose $A \wedge B \equiv \bot$, i.e., there is no assignment to variables present in the formula $A \wedge B$ that makes the formula *true*. Informally, an *interpolant* is a simple explanation as to why A and B are disjoint. Formally, it is defined as follows:

Definition 1 (Interpolant [17]). *An interpolant for a pair of formulas (A, B) such that $A \wedge B \equiv \bot$ is a formula I satisfying $A \Rightarrow I$, $I \wedge B \equiv \bot$, and I refers only to variables common to both A and B.*

Let $\mathit{Vars}(A, B)$ denote the common variables of A and B. We refer to the values assigned to $\mathit{Vars}(A, B)$ by satisfying assignments of A as *positive examples*. Dually, *negative examples* are values assigned to $\mathit{Vars}(A, B)$ by satisfying assignments of B. *Sampling* is the process of obtaining positive and negative examples given A and B. For instance, sampling from $(A \equiv y < x)$ and $(B \equiv y > x)$ with common variables x and y, can give us a positive example $(1, 0)$ and a negative example $(0, 1)$.

A well studied problem in machine learning is *binary classification*. The input to the binary classification problem is a set of points with associated *labels*. By convention, these labels are $l \in \{+1, -1\}$. The goal of the binary classification problem given points with labels is to find a *classifier* $C : \mathit{point} \rightarrow \{\mathit{true}, \mathit{false}\}$ s.t. $C(a) = \mathit{true}$ for all points a with label $+1$, and $C(b) = \mathit{false}$ for all points b with label -1. This process is called *training* a classifier and the set of labeled points is called the *training data*. The goal is to find classifiers that are predictive, i.e., even if we are given a new labeled point w with label l not contained in the training data then it should be very likely that $C(w)$ is *true* iff $l = +1$.

Our goal in this paper is to apply standard binary classification algorithms to positive and negative examples to obtain interpolants. We will assign positive examples the label $+1$ and the negative examples the label -1 to obtain the training data. We are interested in classifiers, in the theory of linear arithmetic, that *classify correctly*.

Definition 2 (Correct Classification). *A classifier C classifies correctly on a given training data X if for all positive examples $a \in X$, $C(a) = true$, and for all negative examples $b \in X$, $C(b) = false$. If there exists a positive example a such that $C(a) = false$ (or a negative example b such that $C(b) = true$), then C is said to have misclassified a (or b).*

There are classification algorithms that need not classify correctly on training data [10]. These are useful because typically the data in machine learning is noisy. A classifier that misclassifies a training example is definitely not an interpolant. Hence we focus on classifiers that classify correctly on training data. In particular, we use optimal margin classifiers generated by *support vector machines* (SVMs).

2.1 SVM Primer

We provide some basic background on SVMs in the context of binary classification using half-spaces. Let us denote the training data by X, the set of positive examples by X^+, and the set of negative examples by X^-.

Let us assume that the training data X is *linearly separable*: there exists a hyperplane, called a *separating hyperplane*, $w^T x + d = 0$ such that $\forall a \in X^+$. $w^T a + d > 0$ and $\forall b \in X^-$. $w^T b + d < 0$. For linearly separable training data, an SVM is guaranteed to terminate with a separating hyperplane. To use a separating hyperplane to predict the label of a new point z we simply compute $sgn(w^T z + d)$. In other words, if $w^T z + d \geq 0$ then we predict the label to be +1 and -1 otherwise.

An interesting question to consider is the following: If there are multiple separating hyperplanes then which one is the best? If a point is away from the separating hyperplane, say $w^T x + d >> 0$, then our prediction that x is a positive example is reasonably confident. On the other hand, if x is very close to the separating hyperplane then our prediction is no longer confident as a minor perturbation can change the predicted label. We say such points have a very low *margin*. The *optimal margin classifier* is the separating hyperplane that maximizes the distance from the points nearest to it. The points closest to the optimal margin classifier are called *support vectors*. An SVM finds the optimal margin classifier and the support vectors given linearly separable training data efficiently [21] by solving a convex optimization problem.

An example of SVM in action is shown in Fig. 3. The positive examples are shown by +'s and negative examples by ∘'s. Line 4 is a separating hyperplane and we can observe that several points of training data lie very close to it and hence its predictions are not so confident. Line 2 is the optimal margin classifier. The points on the dotted lines are closest to the optimal margin classifier and hence are the support vectors.

We observe that using SVMs provides us with a choice of half-spaces for the classifier. We can return the half-space above line 2 as a classifier. All positive examples are contained in it and all negative examples are outside it. Or we can return the half-space above line 1 and that will be a stronger predicate. Or we

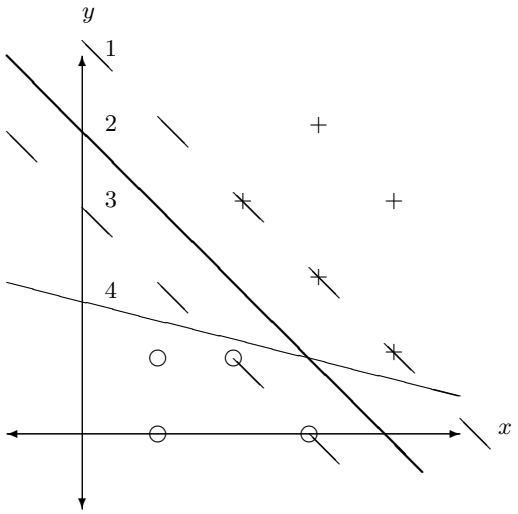

Fig. 3. Line 2 and line 4 are separating hyperplanes. The support vectors for optimal margin classifier (line 2) lie on dotted lines.

can return the negation of the half-space below line 3 and that will be a weaker predicate. Or any line parallel to line 2 and lying between line 1 and line 3 will work. The choice of predicate depends on the application (i.e., the program verification tool that consumes these predicates) and all these predicates can be easily generated by taking a linear combinations of the support vectors.

3 Classification Based Algorithms for Interpolation

We now discuss an algorithm for computing interpolants using an SVM as a black box. We start with a basic version as described in Fig. 4. BASIC takes as input two predicates A and B over the theory of linear arithmetic and generates as output a half-space h over the common variables of A and B. BASIC also has access to (possibly empty) sets of already known positive examples X^+ and negative examples X^-.

```
BASIC(A, B)
Vars := Common variables of A and B
Add Samples(A,X+) to X+
Add Samples(B,X−) to X−
SV := SVM(X+, X−)
h := Process(SV, X+, X−);
return h
```

Fig. 4. The basic algorithm for computing a separating hyperplane.

BASIC first computes the variables common to both A and B and stores them in the set *Vars*. It then computes the positive examples X^+ by repeatedly asking a theorem prover for satisfying assignments of A not already present in X^+(by calling *Samples*(A, X^+)). The values assigned to variables in *Vars* by these satisfying assignments are stored in X^+. The negative examples X^- are computed from B in a similar fashion (by calling *Samples*(B, X^-)). Let us assume that X^+ and X^- are linearly separable. Next, we compute the support vectors (SV of Fig. 4) for X^+ and X^- by calling an off-the-shelf SVM to generate the optimal margin classifier. The result is then processed via the call to procedure *Process* which takes a linear combination of support vectors in SV to obtain the classifier $h = w^T x + d \geq 0$ s.t. $w^T x + d = 0$ is the optimal margin classifier between X^+ and X^-, and $\forall a \in X^+.\ h(a) > 0$ and $\forall b \in X^-.\ h(b) < 0$. This half-space h is returned as output after correction for minor numerical artifacts (say rounding 4.9996 to 5). *Process* can be modified to produce stronger or weaker predicates (Sect. 2.1). The output of BASIC is characterized by the following lemma:

Lemma 1 (Correctness of SVM). *Given positive examples X^+ which are linearly separable from negative examples X^-, SVM and Process compute a half-space h s.t. $\forall a \in X^+.\ h(a) > 0$ and $\forall a \in X^-.\ h(x) < 0$.*

Proof. The lemma follows from the fact that SVM returns an optimal margin classifier under the assumption that X^+ and X^- are linearly separable, and that rounding performed by *Process* does not affect the predicted label of any example in X^+ or X^-.

However, the algorithm BASIC has two major problems:

1. SVM will produce a sound output only when X^+ and X^- are linearly separable.
2. BASIC computes a separator for X^+ and X^- which might or might not separate all possible models of A from all possible models of B.

We will now provide partial solutions for both of these concerns.

3.1 Algorithm for Intersection of Half-spaces

Suppose BASIC samples X^+ and X^- which are not linearly separable. If we denote $x_1, \ldots, x_n$ as the variables contained in *Vars* then there is an obvious (albeit not very useful) separator between X^+ and X^- given by the following predicate:

$$P = \bigvee_{(a_1,\ldots,a_n)\in X^+} x_1 = a_1 \wedge \ldots \wedge x_n = a_n$$

Observe that $\forall a \in X^+.\ P(a) = \mathit{true}$ and $\forall b \in X^-.\ P(b) = \mathit{false}$. The predicate P is a union (or disjunction) of intersection (or conjunction) of half-spaces. To avoid the discovery of such specific predicates, we restrict ourselves to the case where the classifier is either a union or an intersection of half-spaces. This means that we will not be able to find classifiers in all cases even if they exist in the theory of linear arithmetic. We will now give an algorithm which is only guaranteed

to succeed if there exists a classifier which is an intersection of half-spaces. We only discuss the case of intersection here as finding union of half-spaces can be reduced to finding intersection of half-spaces by solving the dual problem.

Definition 3 (Problem Statement). *Given X^+ and X^- such that there exist a set of half-spaces $H = \{h_1, \ldots, h_n\}$ classifying X^+ and X^- correctly (i.e., $\forall a \in X^+.\ \bigwedge_{i=1}^{n} h_i(a)$ and $\forall b \in X^-.\ \neg \bigwedge_{i=1}^{n} h_i(a)$) find H.*

SVM-I(X^+,X^-)
H := *true*
Misclassified := X^-
while |*Misclassified* |$\neq 0$
    Arbitrarily choose b from *Misclassified*
    h := *Process*(*SVM*(X^+, $\{b\}$), X^+, X^-)
    $\forall b' \in$*Misclassified* s.t. $h(b') < 0$: remove b' from *Misclassified*
    H := $H \wedge h$
end while
return H

Fig. 5. Algorithm for classifying by intersection of half-spaces

We find such a classifier using the algorithm of Fig. 5. We initialize the classifier H to *true* or $0 \leq 0$. Next we compute the set of examples misclassified by H. $\forall a \in X^+.H(a) = true$ and hence all positive examples have been classified correctly. $\forall b \in X^-.H(b) = true$ and hence all negative examples have been misclassified. Therefore we initialize the set of misclassified points, *Misclassified*, by X^-. We consider a misclassified element b and find the support vectors between b and X^+. Using the assumption that a classifier using intersection of half-spaces exists for X^+ and X^-, we can show that b is linearly separable from X^+. Using Lemma 1, we will obtain a half-space $h = w^T x + d \geq 0$ for which $h(b) < 0$. We will add h to our classifier and remove the points which h classifies correctly from the set of misclassified points. In particular, b is no longer misclassified and we repeat until all examples have been classified correctly. A formal proof of the following theorem can be developed along the lines of the argument above:

Theorem 1 (Correctness of SVM-I). *If there exists an intersection of half-spaces, H, that can correctly classify X^+ and X^- then* SVM-I *is a sound and complete procedure for finding H.*

We make the following observations about SVM-I:

- The classifier found depends on the order by which the misclassified element b is chosen and different choices can lead to different classifiers.
- In the worst case, it is possible that SVM-I will find as many half-spaces as the number of negative examples. But since optimal margin classifiers generalize well, the worst case behavior does not usually happen in practice.

– SVM-I is related to the problem of "learning intersection of half-spaces". In the latter problem, given positive and negative examples, the goal of the learner is to output an intersection of half-spaces which classifies any new example correctly with high probability. There are several negative results about learning intersection of half-spaces. If no assumptions are made regarding the distribution from which examples come from, we cannot learn intersection of even 2 half-spaces in polynomial time unless RP=NP [2,18].

SVM-I can be incorporated into Basic by replacing the calls to *SVM* and *Process* with SVM-I in Fig. 4. Now Basic with SVM-I can find classifiers when X^+ and X^- are not linearly separable but can be separated by an intersection of half-spaces.

3.2 A Sound Algorithm

We observe that Basic, with or without SVM-I, only finds classifiers between X^+ and X^-. The way Basic is defined, these candidate interpolants are over the common variables of A and B. But if we do not have enough positive and negative examples then a classifier between X^+ and X^- is not necessarily an interpolant. When this happens, we need to add more positive and negative examples refuting the candidate interpolant.

```
Interpolant(A, B)
X+, X- := ∅
while true
      H := Basic(A, B) // Basic with SVM-I
      if SAT(A ∧ ¬H)
         Add satisfying assignment to X+ and continue
      if SAT(B ∧ H)
         Add satisfying assignment to X- and continue
      break
return H
```

Fig. 6. A sound algorithm for interpolation

The algorithm Interpolant computes a classifier H which classifies X^+ and X^- correctly i.e., $\forall a \in X^+.\ H(a) = \mathit{true}$ and $\forall b \in X^-.\ H(b) = \mathit{false}$ by calling Basic with SVM-I. If H is implied by A and is unsatisfiable in conjunction with B then we have found an interpolant and we exit the loop. Otherwise we update X^+ and X^- and try again. We have the following theorem:

Theorem 2 (Soundness of Interpolant). Interpolant*(A,B) terminates if and only if the output H is an interpolant between A and B.*

Proof. The output H is defined over the common variables of A and B (follows from the output of Basic).

only if : Let INTERPOLANT(A, B) terminate. This means that both conditions $B \wedge H \equiv \bot$ and $A \wedge \neg H \equiv \bot$ must be satisfied (these are conditions for reaching `break` statement), which in turn implies that $A \Rightarrow H$ holds and therefore H is an interpolant of A and B.

if : Let H be an interpolant of A and B. This means that $A \Rightarrow H$ and hence $A \wedge \neg H \equiv \bot$. $B \wedge H \equiv \bot$ holds because H is an interpolant and therefore, the `break` statement is reachable and INTERPOLANT(A, B) terminates.

4 Handling Superficial Non-linearities

Most program verification engines do not reason about non-linear arithmetic directly. They try to over-approximate non-linear functions, say by using uninterpreted function symbols. In this section, we discuss how to use our technique to over-approximate non-linear arithmetic by linear functions.

Suppose $A \wedge B \equiv \bot$ and A is a non-linear predicate. If we can find a linear interpolant I between A and B then $A \Rightarrow I$. Hence I is a linear over-approximation of the non-linear predicate A. We discuss, using an example, how such a predicate I can be useful for program verification.

Suppose we want to prove that line 5 is unreachable in Fig. 7. There are some lines which are commented. These will be considered later. This program assigns `z` non-deterministically and does some non-linear computations. If we can show that an over-approximation of reachable states after line 3 is disjoint from $x = 2 \wedge y \neq 2$ then have a proof that `error()` is unreachable.

```
foo()
{
// do{
1:    z = nondet();
2:    x = 4 * sin(z) * sin(z);
3:    y = 4 * cos(z) * cos(z);
// } while (*);
4:    if ( x == 2 && y != 2 )
5:      error() ;
}
```

Fig. 7. A contrived example with superficial non-linearities

We use our technique for computing interpolants over the non-linear predicates to construct an easy to analyze over-approximation of this program. We want to find an interpolant of the following predicates (corresponding to the infeasible trace (1, 2, 3, 4, 5)):

$$A \equiv x = 4sin^2(z) \wedge y = 4cos^2(z)$$
$$B \equiv x = 2 \wedge y \neq 2$$

Observe that SVMs consume examples and are agnostic to how the examples are obtained. Since A is non-linear, we can obtain positive examples by randomly

substituting values for z in A and recording the values of x and y. Since B is linear, we can ask an SMT solver [19] for satisfying assignments of B to obtain the negative examples. We have plotted one possible situation in Fig. 8 – the positive and negative examples are represented by +'s and ∘'s respectively. Running SVM-I and choosing the stronger predicate from the available choices (Sect. 2.1) generates the predicate $P \equiv (x + y = 4)$. We remark that to obtain this predicate, we only need one negative example above, one negative example below, one positive example to the left, and one positive example to the right of $(2, 2)$. Adding more examples will leave P unaffected, due to the way optimal margin classifier is defined (Sect. 2.1). This shows the robustness of the classifier. That is, once a sufficient number of samples have been obtained then the classifier is not easily perturbed by changes in the training data.

Now we need to verify that P is actually an interpolant. We use an SMT solver to show that $P \wedge B \equiv \bot$. To show $A \Rightarrow P$ can be hard. For this example, any theorem prover with access to the axiom $sin^2(x) + cos^2(x) = 1$ will succeed. But we would like to warn the reader that the verification step, where we check $A \Rightarrow I$ and $I \wedge B \equiv \bot$, can become intractable for arbitrary non-linear formulas.

Using the interpolant P, we can replace Fig. 7 by its over-approximation given in Fig. 9 for verification. A predicate abstraction engine using predicates $\{x + y = 4, x = 2, y = 2\}$ can easily show the correctness of the program of Fig. 9. Moreover, suppose we uncomment the lines which have been commented out in Fig. 7. To verify the resulting program we need a sufficiently strong loop invariant. To find it we consider a trace executing the loop once and try to find the interpolant. We do the exact same analysis we did above and obtain the interpolant $(x + y = 4)$. This predicate is an invariant and is sufficient to prove the unreachability of `error()`.

Other techniques for interpolation fail on this example because either they replace *sin* and *cos* by uninterpreted functions [13,24] or because of the restricted expressivity of the range of interpolants computed (e.g. combination of boxes [15]). We succeed on this example because of two reasons:

1. We are working with examples and hence we are not over-approximating the original constraints.
2. SVM succeeds in computing a predicate which generalizes well.

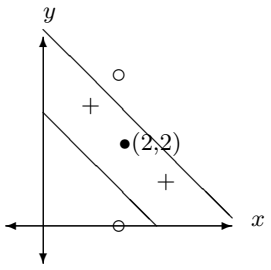

Fig. 8. Positive and negative examples for Fig. 7. The lines show classifiers.

```
foo()
{
  assume ( x + y  ==  4 );
  if ( x == 2 && y != 2)
    error() ;
}
```

Fig. 9. An over-approximation of Fig. 7.

5 Experiments

We have implemented a prototype version of the algorithm described in this paper in 1000 lines of C++ using LIBSVM [3] for SVM queries and the Z3 theorem prover [19]. Specifically, we use the C-SVC algorithm with a linear kernel for finding the optimal margin classifier. C-SVC is parametrized by a cost parameter c. A low value of c allows the generated classifier to make errors on the training data. Since we are interested in classifiers that classify correctly, we assign a very high value to c (1000 in our experiments). The input to our implementation is two SMT-LIB formulas and the output is also obtained as an SMT-LIB formula. We try to sample for at most ten distinct positive and negative examples each before BASIC makes a call to LIBSVM. In these experiments, the classifier is described by the hyperplane parallel to the optimal margin classifier and passing through the positive support vectors. We consider the half-space, corresponding to this hyperplane, such that the negative examples lie outside the half-space. Hence we are considering the strongest predicate from the options provided to us by SVM (Sect. 2.1).

We have tried our technique on small programs and our results are quite encouraging (see Table 1). The goal of our experiments was to verify the implementability of our approach. We consider traces that go through the loops once and manually generate A and B in SMT-LIB format for input to our tool. These programs contain assertions that can be discharged using loop invariants that are a conjunction of linear inequalities.

First, let us consider the left half of the table. The programs `f1a`, `ex1`, and `f2` are adapted from the benchmarks used in [6]. The programs `nec1` to `nec5` are adapted from NECLA static analysis benchmarks [12]. The program `fse06` is from [7] and is an example on which YOGI [7] does not terminate because it cannot find the invariant $x \geq 0 \wedge y \geq 0$.

The program `pldi08`, adapted from [9], requires a disjunction of half-spaces as an invariant. We obtain that by solving the dual problem: we interchange the labels of positive and negative examples and output the negation of the interpolant obtained. For these examples, we were generating at most ten positive and negative examples before invoking SVM. Hence we expect the column "Total Ex" to have entries less than or equal to 20. Most entries are strictly less than twenty because several predicates have strictly less than ten satisfying assignments. This is expected for A as it represents reachable states and we are considering only one iteration of the loops. So very few states are reachable and

Table 1. File is the name of the benchmark, LOC is lines of code, Interpolant is the computed interpolant, Total Ex is the sum of the number of positive and negative examples generated for the first iteration of INTERPOLANT. For the second part, Iterations represents the number of iterations of INTERPOLANT.

File	LOC	Interpolant	Total Ex	Time (s)	Interpolant	Iterations	Time (s)
`f1a`	20	`x = y`	12	0.017	`x = y & y >= 0`	4	0.017
`ex1`	22	`xa + 2*ya >= 0`	13	0.019	`xa + 2*ya >= 0`	4	0.02
`f2`	18	`3*x >= y`	13	0.021	`3*x >= y`	12	0.022
`nec1`	17	`x <= 8`	19	0.015	`x <= 8`	9	0.02
`nec2`	22	`x < y`	12	0.014	`x < y`	2	0.019
`nec3`	15	`y <= 9`	11	0.014	`y <= 9`	1	0.012
`nec4`	22	`x = y`	20	0.019	`x = y`	4	0.017
`nec5`	9	`s >= 0`	11	0.013	`s >= 0`	1	0.016
`pldi08`	10	`x < 0 | y > 0`	17	0.02	`6*x < y`	1	0.013
`fse06`	8	`y >= 0 & x >= 0`	11	0.014	`y >= 0 & x >= 0`	2	0.015

hence A has very few satisfying assignments. Nevertheless, 11 to 20 examples are sufficient to terminate INTERPOLANT in a single iteration for all the benchmarks.

To get more intuition about INTERPOLANT, we generate the second part of the table. Here we start with one positive and one negative example. If the classifier is not an interpolant then we add one new point that the classifier misclassifies. The general trend is that we are able to find the same classifier with a smaller number of samples and few iterations. In `f1a` we generate a predicate with more inequalities. This demonstrates that the generated classifier from SVM-I might be sensitive to the order in which misclassified examples are traversed (Fig. 5). For `pldi08`, when we found the classifier between the first positive and negative example generated by Z3 then we found that it was an interpolant. Since the classifier has been generated using only two examples, the training data is insufficient to reflect the full structure of the problem, and unsurprisingly we obtain a predicate that does not generalize well. These experiments suggest that the convergence of INTERPOLANT is faster and the results are better if we start with a reasonable number of samples.

Finally, we compare with the interpolation procedure implemented within OPENSMT [16] in Table 2. OPENSMT fails to find the predicate representing the loop invariant for `f1a`, `pldi08`, and `fse06`. This is in line with our claim that machine learning algorithms can provide relevant predicates. OPENSMT fails on `nec1` because this benchmark contains non-linear multiplications. It turns out that the program has a linear interpolant, found by our technique, which is sufficient to discharge the assertions in the program. Finally, the timing measurements show that we are competitive with OPENSMT.

6 Related Work

In this section, we place our work in the context of existing work on interpolation and machine learning. Our philosophy of computing interpolants from samples

Table 2. File is the name of the benchmark and Interpolant is the interpolant computed by the interpolation procedure implemented within OpenSMT. SAME refers to the benchmarks for which interpolants computed by OpenSMT were identical to those computed by our technique.

File	Time(s)	Interpolant
f1a	0.022	((y = 1 | x <= 0) & x = 1) | (y = 0 & (y = 1 | x <= 0))
ex1	0.021	xa + 2*ya >= 0 | xa + 2*ya >= 5 | xa + 2*ya >= 5
f2	0.020	y <= 3*x | y <= 3*x + 1 | y <= 3*x + 1
nec1	NA	FAIL
nec2	0.018	x < y (SAME)
nec3	0.016	y <= 9 (SAME)
nec4	0.021	(x = y | y = 0) | (y = x) | (y = x)
nec5	0.018	s >= 0 (SAME)
pldi08	0.017	y > x
fse06	0.017	y + x >= 0 & y >= 0 & y >= 0 & y >= 0

is similar to Daikon [5]; Daikon computes likely invariants from program tests. Whereas, we compute sound interpolants statically.

We have considered interpolation only over the quantifier free theory of linear arithmetic. Extension to richer theories, such as the theory of arrays, is left for future work. The interpolants found by our technique are limited to conjunctions of linear inequalities. To handle programs requiring interpolants which are a combination of disjunctions and conjunctions of linear inequalities, we propose to use the existing techniques for control flow refinement [1,8,26]. These techniques perform source to source semantics preserving transformations so that the loops in the resulting program require only disjunction-free invariants.

Extending the work of [14,22], McMillan [17] computed interpolants of (A, B), where A and B are in the quantifier free theory of linear arithmetic, in a linear scan of the proof of unsatisfiablity of $A \wedge B$. This method requires an explicit construction of the proof of unsatisfiability. In a recent work, Kupferschmid et al. [15] gave a proof based method for finding Craig interpolants for non-linear predicates. The proof based methods like these are generally not scalable: Rybalchenko et al. [24] remark that "Explicit construction of such proofs is a difficult task, which hinders the practical applicability of interpolants for verification." Like our approach, their method for interpolation is also not proof based. They apply linear programming to find separating hyperplanes between A and B. In contrast to their approach, we are working with samples and not symbolic constraints. This allows us to use mature machine learning techniques like SVMs as well as gives us the ability to handle superficial non-linearities.

We selected SVM for classification as they are one of the simplest and most widely used machine learning algorithms. There are some classification techniques which are even simpler than SVM [10]. We discuss them here and give the reasons behind not using them for classification. In linear regression, we construct a quadratic penalty term for misclassification and find the hyperplane which minimizes the penalty. Unfortunately the classifiers obtained might err

on the training data even if it is linearly separable. Another widespread technique, logistic regression, is guaranteed to find a separating hyperplane if one exists. But the output of logistic regression depends on all examples and hence the output keeps changing even if we add redundant examples. The output of SVMs, on the other hand, is entirely governed by the support vectors and is not affected by other points at all. This results in a robust classifier which is not easily perturbed and leads to better predictability in results.

There has been research on finding non-linear invariants [25,20,23]. These techniques aim at finding invariants which are restricted to polynomials of variables. In contrast, we are not generating non-linear predicates. We are finding linear over-approximations of non-linear constraints and hence our technique only generates linear predicates. On the other hand, unlike [25,20,23] we are not restricted to non-linearities resulting only from polynomials and have demonstrated our technique on an example with transcendental functions.

7 Conclusion

We have shown that classification based machine learning algorithms can be profitably used to compute interpolants and therefore are useful in the context of program verification. In particular, we have given a step-by-step account of how off-the-shelf SVM algorithms can be used to compute interpolants in a sound way. We have also demonstrated the feasibility of applying our approach via experiments over small programs from the literature. Moreover, we are also able to compute interpolants for programs that are not analyzable by existing approaches – specifically, our technique can handle superficial non-linearities.

As future work, we would like to extend our algorithms to compute interpolants for non-linear formulas. We believe that SVMs are a natural tool for this generalization as they have been extensively used to find non-linear classifiers. We would also like to integrate our SVM-based interpolation algorithm with a verification tool and perform a more extensive evaluation of our approach.

Acknowledgements We thank the anonymous reviewers for their constructive comments. We thank David Dill, Bharath Hariharan, Prateek Jain, and Hristo Paskov for helpful discussions.

References

1. Balakrishnan, G., Sankaranarayanan, S., Ivancic, F., Gupta, A.: Refining the control structure of loops using static analysis. In: EMSOFT. pp. 49–58 (2009)
2. Blum, A., Rivest, R.L.: Training a 3-node neural network is NP-complete. In: Machine Learning: From Theory to Applications. pp. 9–28 (1993)
3. Chang, C.C., Lin, C.J.: LIBSVM: A library for support vector machines. ACM Transactions on Intelligent Systems and Technology 2, 27:1–27:27 (2011), software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`

4. Craig, W.: Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. J. Symb. Log. 22(3), 269–285 (1957)
5. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The Daikon system for dynamic detection of likely invariants. Sci. Comput. Program. 69(1-3), 35–45 (2007)
6. Gulavani, B.S., Chakraborty, S., Nori, A.V., Rajamani, S.K.: Automatically refining abstract interpretations. In: TACAS. pp. 443–458 (2008)
7. Gulavani, B.S., Henzinger, T.A., Kannan, Y., Nori, A.V., Rajamani, S.K.: Synergy: a new algorithm for property checking. In: SIGSOFT FSE. pp. 117–127 (2006)
8. Gulwani, S., Jain, S., Koskinen, E.: Control-flow refinement and progress invariants for bound analysis. In: PLDI. pp. 375–385 (2009)
9. Gulwani, S., Srivastava, S., Venkatesan, R.: Program analysis as constraint solving. In: PLDI. pp. 281–292 (2008)
10. Hastie, T., Tibshirani, R., Friedman, J.: The Elements of Statistical Learning. Springer Series in Statistics, Springer New York Inc. (2001)
11. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: POPL. pp. 232–244 (2004)
12. Ivancic, F., Sankaranarayanan, S.: NECLA Static Analysis Benchmarks, available at `http://www.nec-labs.com/research/system/systems_SAV-website/small_static_bench-v1.1.tar.gz`
13. Jhala, R., McMillan, K.L.: A practical and complete approach to predicate refinement. In: TACAS. pp. 459–473 (2006)
14. Krajícek, J.: Interpolation theorems, lower bounds for proof systems, and independence results for bounded arithmetic. J. Symb. Log. 62(2), 457–486 (1997)
15. Kupferschmid, S., Becker, B.: Craig interpolation in the presence of non-linear constraints. In: FORMATS. pp. 240–255 (2011)
16. Leroux, J., Rümmer, P.: Craig Interpolation for Presburger Arithmetic in OpenSMT, `http://www.philipp.ruemmer.org/interpolating-opensmt.shtml`
17. McMillan, K.L.: An interpolating theorem prover. Theor. Comput. Sci. 345(1), 101–121 (2005)
18. Megiddo, N.: On the complexity of polyhedral separability. Discrete & Computational Geometry 3, 325–337 (1988)
19. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS. pp. 337–340 (2008)
20. Müller-Olm, M., Seidl, H.: Computing polynomial program invariants. Inf. Process. Lett. 91(5), 233–244 (2004)
21. Platt, J.C.: Fast training of support vector machines using sequential minimal optimization. In: Advances in Kernel Methods: Support Vector Learning. pp. 185–208. MIT Press (1998)
22. Pudlák, P.: Lower bounds for resolution and cutting plane proofs and monotone computations. J. Symb. Log. 62(3), 981–998 (1997)
23. Rodríguez-Carbonell, E., Kapur, D.: Generating all polynomial invariants in simple loops. J. Symb. Comput. 42(4), 443–476 (2007)
24. Rybalchenko, A., Sofronie-Stokkermans, V.: Constraint solving for interpolation. In: VMCAI. pp. 346–362 (2007)
25. Sankaranarayanan, S., Sipma, H., Manna, Z.: Non-linear loop invariant generation using Gröbner bases. In: POPL. pp. 318–329 (2004)
26. Sharma, R., Dillig, I., Dillig, T., Aiken, A.: Simplifying loop invariant generation using splitter predicates. In: CAV. pp. 703–719 (2011)