Data-Driven Learning of Strong Conjunctive Invariants

Arkesh Thakkar  
Department of Computer Science and Automation  
Indian Institute of Science  
Bengaluru, India  
arkesh.thakkar14@gmail.com, arkezhkumar@iisc.ac.in

Deepak D’Souza  
Department of Computer Science and Automation  
Indian Institute of Science  
Bengaluru, India  
deepakd@iisc.ac.in

Abstract—Coming up with adequate inductive invariants is the key step in the deductive verification of programs. Moreover, coming up with strong inductive invariants is important in several application scenarios like compositional verification. Houdini-style approaches are popular techniques that learn the strongest conjunctive subset of a given set of constraints that is adequate to prove a given pre-post specification for a program. However, both the adequacy and strength of the learned invariants depend on the quality of the initial set of constraints. In this work, we propose a data-driven way of learning an initial set of constraints from sample runs of the program. We use a novel combination of linear regression for equality constraints and linear programming for inequality constraints. The evaluation of our technique shows that it is more efficient and leads to stronger adequate invariants than some of the state-of-the-art data-driven techniques.

I. INTRODUCTION

Finding likely invariants at points in a program from sample execution traces of the program has many applications. These include program understanding and specification mining [1], program testing [12], [22], and program verification [18], [21]. Our interest is mainly in the application of learning likely numerical invariants in the context of the Floyd-Hoare style verification of pre-post specifications for programs. Many existing techniques, like Houdini [8] and ICE-learning [6], [10] attempt to learn adequate inductive invariants from a set of base predicates. Such techniques could also benefit from a good base set of likely invariants.

In many applications, coming up with “tight” invariants that fit the sample data well while still generalizing well to unseen data or leading to adequate inductive invariants is important. Tight invariants are useful in proving strong postconditions, or in inferring strong postconditions for compositional verification (say of programs with sequential loops).

Prior works in data-driven invariant generation have typically used techniques like enumerative checking of a set of bounded templates [5], algebraic equality solving [16], [21], convex hulls [16], and neural network based learning [20], [23]. In this paper, we propose a technique for data-driven invariant learning based on classical linear regression for equality constraints, and linear programming for inequality constraints. We used linear regression along with feature selection to find a good set of equality invariants; feature selection gives us more optimal answers and avoids overfitting the data. Our linear programming formulation for finding tight inequality invariants are both more efficient and produce stronger invariants in comparison to techniques based on convex hulls or neural networks.

We have evaluated our technique on a range of simple loop verification benchmarks from SVCOMP [2], by collecting sample data at the head of the loop and learning a set of candidate invariants. We then try to find a conjunctive subset of these invariants that is adequate to prove the postconditions using the Houdini algorithm. We did the same with three state-of-the-art invariant generation tools. We were able to find more adequate invariants than all the other tools (100 out of 104 programs), the strongest adequate invariant, in comparison with the other tools, in 87 programs, and the average time to generate the candidate set of invariants is 8.32 seconds.

The rest of the paper is organized as follows. We begin with an overview of our approach in Sec. II, followed by background material in Sec. III which gives the basics of Floyd-Hoare verification and the Houdini technique. Secs. IV, V and VI describe our technique to generate equality and inequality invariants, and the overall algorithm, respectively. Sec. VII describes the experimental evaluation of our approach. We discuss related work in Sec. VIII and conclude in Sec. IX.

II. OVERVIEW

Given a program $P$ having a set of variables $V$, precondition $pre$ and postcondition $post$ and a loop as shown in Fig. 1, we first collect the traces $T$ of variables (i.e., the value that each variable holds) at different loop iterations by giving a random set of inputs that satisfy the pre condition. We also generate polynomial terms (i.e. non-linear terms) from the existing variables of $P$ upto a user-given degree $r$. This helps us learn non-linear invariants as well, along with the linear invariants.

After collecting the trace data $T$ and generating a polynomial degree, let the total number of dimensions (including new variables representing polynomial terms) be $n$. Considering each variable as a target variable and other $n-1$ variables as independent variables, we apply feature selection techniques to get the features (i.e. variables among the $n-1$ independent variables) that are closely related to the target variable. We then apply linear regression to find the equality relationships
among the target and closely related independent variables. Applying feature selection is important because if \( n \) is large, linear regression without feature selection could lead to overfitting and we may not be able to infer a generalized equality relationship between the target and independent variables.

Consider one of the programs named “knuth”, shown below, from the SVCOMP nla-digbench benchmark [2]. The program implements an algorithm that searches for a divisor for factorization.

\[
\begin{align*}
\text{unsigned } n, a; \\
\text{unsigned } r, k, q, d, s, t; \\
d = a; \\
r = n \% d; \\
t = 0; \\
k = n \% (d - 2); \\
q = 4 \cdot (n / (d - 2) - n / d); \\
s = \sqrt{\text{sqrt}(n)}; \\
\text{while } (1) \\
\quad \text{if } (!((s >= d) \&\&(r != 0))) \\
\quad \quad \text{break;}
\quad \text{if } (2*\ r + q < k) \\
\quad \quad \quad \ t = r; \\
\quad \quad \quad \ r = 2*\ r - k + q + d + 2; \\
\quad \quad \quad \ k = t; \\
\quad \quad \quad \ q = q + 4; \\
\quad \quad \quad \ d = d + 2; \\
\quad \text{else if } ((2*\ r + q >= k) \\
\quad \quad \quad \ & \& \ (2*\ r + q < d + k + 2)) \\
\quad \quad \quad \ \ t = r; \\
\quad \quad \quad \ \ r = 2*\ r - k + q; \\
\quad \quad \quad \ \ k = t; \\
\quad \quad \quad \ \ d = d + 2; \\
\quad \text{else if } ((2*\ r + q >= k) \&\& \\
\quad \quad \quad \ \ (2*\ r + q >= d + k + 2)) \&\& \\
\quad \quad \quad \ \ (2*\ r + q < 2*d + k + 4)) \\
\quad \quad \quad \ \ t = r; \\
\quad \quad \quad \ \ r = 2*\ r - k + q - d - 2; \\
\quad \quad \quad \ \ k = t; \\
\quad \quad \quad \ \ q = q - 4; \\
\quad \quad \quad \ \ d = d + 2; \\
\quad \text{else} \\
\quad \quad \ t = r; \\
\quad \quad \ r = 2*\ r - k + q - 2*d - 4; \\
\quad \quad \ k = t; \\
\quad \quad \ q = q - 8; \\
\quad \quad \ d = d + 2; \\
\quad \end{align*}
\]

A snapshot of collected traces is shown in the following table:

<table>
<thead>
<tr>
<th>( n )</th>
<th>( a )</th>
<th>( r )</th>
<th>( k )</th>
<th>( q )</th>
<th>( d )</th>
<th>( s )</th>
<th>( t )</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>3</td>
<td>60</td>
<td>1</td>
<td>60</td>
<td>5</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>25</td>
<td>3</td>
<td>165</td>
<td>1</td>
<td>60</td>
<td>5</td>
<td>5</td>
<td>60</td>
</tr>
<tr>
<td>49</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>132</td>
<td>3</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>49</td>
<td>3</td>
<td>124</td>
<td>1</td>
<td>124</td>
<td>5</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>49</td>
<td>3</td>
<td>357</td>
<td>124</td>
<td>116</td>
<td>7</td>
<td>7</td>
<td>124</td>
</tr>
<tr>
<td>49</td>
<td>3</td>
<td>688</td>
<td>357</td>
<td>108</td>
<td>9</td>
<td>7</td>
<td>357</td>
</tr>
</tbody>
</table>

The documented invariant for the program is

\[
ddq - 2qd - 4rd + 4kd + 8r = 8n.
\]

As it requires some degree 3 terms, generating terms up to degree 3 from the traces goes up to 164 dimensions. Without applying feature selection, linear regression leads to overfitting and doesn’t give any equality invariant. Thus, feature selection is important for learning useful equality invariants. Furthermore, we use Linear Programming to help us learn tight inequality constraints in two and three dimensions.

As shown in Sec. VII, none of the three existing tools, namely Daikon, DIG, and GCLN, we experimented with were able to find an adequate invariant for this program, due to lack of effective feature selection techniques. In contrast, our tool LPGEN finds an adequate invariant for the above program, due to its use of feature selection techniques (details in Sec. IV).

### III. BACKGROUND

#### A. Preliminaries

We will use \( \mathbb{Z} \) and \( \mathbb{R} \) to denote the set of integers and reals respectively. For an \( m \times n \) matrix \( M \) over the reals, we will denote by \( M^\top \) the transpose of \( M \), which is the \( n \times m \) matrix obtained by changing the rows of \( M \) into columns. For \( m \times n \) matrices \( M \) and \( N \), we will denote by \( M + N \) (respectively \( M - N \)) the matrix obtained by the pointwise addition (respectively subtraction) of elements of \( M \) and \( N \), and write \( M \leq N \) to mean that each \( (i, j) \)-th element of \( M \) is less than or equal to the \( (i, j) \)-th element of \( N \). For vectors \( u = [u_1, \ldots, u_n] \) and \( v = [v_1, \ldots, v_n] \), the inner product of \( u \) and \( v \), denoted \( u \cdot v \), is \( \sum_{i=1}^{n} u_i v_i \). The \( L_2 \) norm of \( u \), denoted \( \|u\| \), is defined to be \( \sqrt{\sum_{i=1}^{n} u_i^2} \), while the \( L_1 \) norm of \( u \), denoted \( \|u\|_1 \), is defined to be \( \sum_{i=1}^{n} |u_i| \).

We will be dealing with a quantifier-free logical language of constraints. Given a set of variables \( V = \{x_1, \ldots, x_n\} \), we define the following language of constraints over \( V \). The terms \( t \) of this language will comprise variables from \( V \), constants \( l \in \mathbb{Z} \), and products and sums of terms. An (atomic) constraint over \( V \) is now of the form \( t \sim t' \), where \( \sim \in \{=, <, \leq, >, \geq\} \).

In some cases we will also allow atomic constraints of the form \( x \mod l = d \) and \( x \mod l = m \), where \( x \) is an integer variable, and \( l \) and \( m \) are integers, representing that \( x \mod l = d \) and the bitwise-and of \( x \) and \( l \) is \( m \); and \( \gcd(x, y) = \gcd(x', y') \) where \( x, y, x', y' \) are integer-valued variables in \( V \), and \( \gcd \) denotes the greatest common divisor function. A constraint

<table>
<thead>
<tr>
<th>Table I</th>
<th>Variable Values in Traces of Example Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n )</td>
<td>( a )</td>
</tr>
<tr>
<td>25</td>
<td>3</td>
</tr>
<tr>
<td>25</td>
<td>3</td>
</tr>
<tr>
<td>49</td>
<td>3</td>
</tr>
<tr>
<td>49</td>
<td>3</td>
</tr>
<tr>
<td>49</td>
<td>3</td>
</tr>
<tr>
<td>49</td>
<td>3</td>
</tr>
</tbody>
</table>
over $V$ is a boolean combination of atomic constraints over $V$.

A valuation $u$ of variables in $V$ is a map that assigns to each variable $x$ in $V$ a real value $u(x)$. Wherever convenient, we will represent such a valuation as a vector $[u(x_1), \ldots, u(x_n)]$. A valuation $u$ assigns a real value $t^u$ to each term $t$ over $V$, by inductively defining $t^u = l$, $x^u = u(x)$, and $(t + t')^u = t^u + (t')^u$, etc. We say $u$ satisfies an atomic constraint $t \sim t'$ if $t^u \sim (t')^u$. Finally, boolean combinations are handled in the expected manner. We write $u \models c$ to denote the fact that the valuation $u$ satisfies the constraint $c$.

B. Verification Conditions for Simple Programs

We will consider programs with a simple loop structure, as shown in Fig. 1. The programs have a single loop, with a set of initialization statements $S_1$, loop body statements $S_2$, and post-loop statements $S_3$. The statements in these blocks are assignments or if-then-else statements.

We will use standard Floyd-Hoare logic [9], [13] to specify and reason about the correctness of these programs. A state of a program is simply a valuation to its variables, and thus we can ask whether it satisfies a given constraint on the variables of the program. An “pre-post” specification for a program $P$ over a set of variables $V$ is pair of constraints $(pre, post)$ over $V$. We say $P$ satisfies the specification $(pre, post)$ if every terminating execution of $P$ that begins in a state satisfying $pre$, terminates in a state satisfying $post$.

A standard way to prove that a program $P$ satisfies a specification $(pre, post)$ is to come up with a constraint $inv$ over the variables of the program, satisfying the conditions (C1), (C2) and (C3) on the right in Fig. 1 (in the the sense that the implications are logically valid). In the conditions, we use the notation $[S]$ to denote the standard logical semantics of the statement $S$ over the set of variables $V$ and $V' = \{x' \mid x \in V\}$, representing the states before and after the execution of $S$, respectively. For example, for an assignment statement $x := x + 1$ in a program with variables $V = \{x, y\}$, we have $[x := x + 1]$ to be $x' = x + 1 \land y' = y$. We also use the notation $inv'$ to denote the constraint obtained by substituting $x'$ for $x$ in $inv$, for each variable $x \in V$. We call these conditions verification conditions, and they essentially require $inv$ to be an adequate inductive invariant for $P$, with conditions (C1) and (C3) enforcing adequacy of $inv$ w.r.t. the pre-post conditions, and condition (C2) enforcing inductiveness of $inv$.

C. Houdini Algorithm

We briefly outline the Houdini algorithm of [8] (see also [15]), which is useful in efficiently identifying an adequate inductive conjunctive subset of a given set of candidate invariants.

Let $P$ be a program of the form shown in Fig. 1 over a set of variables $V$, and $(pre, post)$ be a given pre-post specification for $P$. Let $C$ be a finite set of constraints over $V$. Then the Houdini algorithm finds the strongest conjunctive subset of $C$ that is an adequate inductive invariant for $P$ w.r.t. $(pre, post)$.

The algorithm proceeds as follows. As usual, we treat $\bigwedge \emptyset$ to be true.

1) Let $C_1$ be the subset of $C$ containing all constraints $c \in C$ such that $pre \land [S_1] \implies c'$.
2) Check if $\bigwedge C_1$ satisfies condition (C3) (i.e. $\bigwedge C_1 \land \neg b \land [S_3] \implies post'$). If not, return “No adequate invariant exists.”
3) Check if $\bigwedge C_1$ satisfies condition (C2) (i.e. $\bigwedge C_1 \land b \land [S_2] \implies \bigwedge C'_1$). If so, return $\bigwedge C_1$ as an adequate inductive invariant.

Else, let $(u, v)$ be a counter-example, where $u \models b \land \bigwedge C_1$ but $v' \not\models \bigwedge C'_1$, where $v$ is the state resulting from executing $S_2$ in $u$. Let $C_2$ be the subset of $C_1$ obtained by dropping constraints that are not satisfied by $v$. Thus $C_2 = C_1 - \{c \in C_1 \mid v' \not\models c\}$. Set $C_1$ to $C_2$, and go back to Step 2.

IV. Equality Invariants using Linear Regression

Suppose we are interested in tracking the values of $n$ different terms at the loop head in a program $P$. These terms could be variables of the program (like $x$ and $i$) or product terms like $(x^2 \times x)$. Let us say we have collected $m$ sample values. Then we can represent the values collected as a data set $D = \{d_1, \ldots, d_m\}$, where each $d_i = [d_{11}, \ldots, d_{1n}]$ represents the values of notional variables $x_1, \ldots, x_n$ standing for the terms we are interested in tracking. These notional variables are also called features. Let us fix such a data set $D$, with feature dimension $n \geq 2$ and number of samples $m \geq 1$, for the next couple of sections.

In this section, our goal is to learn a linear equality relationship between the notional variables, whenever one exists. To do this we consider each variable $x$ as the target variable, and consider the remaining $n - 1$ variables as independent variables. The goal is to see if the target variable can be expressed as a “linear” combination of the independent variables. For example, if $x_1$ is our target variable, we would like to know if $x_1 = w_1 x_2 + w_2 x_2 + \cdots + w_n x_n$, for some real-valued weights $w_1, \ldots, w_n$. The technique of linear regression helps us do this.

However, when the number of independent variables is large, linear regression may not give us good results as it could
overfit the data in a dataset \( D \) and may fail to generalize. Hence we first find a subset of closely-related independent variables (this step is called feature selection), and then apply linear regression on this restricted set of variables.

In our proposed technique, we use two feature selection techniques, namely L1 regularization and univariate feature selection. We then perform two linear regression steps: once on the subset obtained by regularization and once more on the subset obtained by univariate feature selection. Finally, after obtaining the equality constraints, we check whether they satisfy all the data points and discard those that don’t.

These steps are summarized in Fig. 2. In the rest of this section, we describe them in more detail.

**A. Linear Regression**

Given the data set \( D = \{d_1, \ldots, d_m\} \), we would like to know whether one of the variables (say \( x_1 \)) bears a linear relationship to the remaining variables, in that there exist real values \( w_1, \ldots, w_n \) such that for each \( i, \) \( d_i \) equals (or is close to) \( w_1 + w_2d_2 + \cdots + w_n d_n \). In the method of least-squares regression, for a given vector \( W = [w_1, \ldots, w_n] \), we define the error or loss it entails by

\[
L(D, W) = \sum_{i=1}^m (d_{i,1} - (w_1 + \sum_{k=2}^n w_k d_{i,k}))^2. \tag{1}
\]

We now ask for a weight vector \( W \) which minimizes \( L(D, W) \).

In matrix form one can express \( L(D, W) \) as \( \|Y - XW\|^2 \), where

\[
X = \begin{bmatrix} 1 & d_{12} & d_{13} & \cdots & d_{1n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & d_{m2} & x_{m3} & \cdots & x_{mn} \end{bmatrix}
\]

and \( Y = [d_{11}, \ldots, d_{m1}]^\top \). A standard result says that the value of \( W \) that minimizes \( L(D, W) \) can be computed by the closed-form solution:

\[
W = (X^\top X)^{-1}X^\top Y. \tag{2}
\]

**B. L1 Regularization**

L1 regularization is a way of selecting the features that are correlated to the target variable. It seeks to penalize features that are not correlated to the target variable by making their coefficient weights zero.

Mathematically, in L1 regularization, we modify the loss function from Eq (1), by adding a term as follows:

\[
L(D, W) = \|Y - XW\|^2 + 2\lambda \|W\|. \tag{3}
\]

Here \( \lambda \) is a hyperparameter that controls the amount of regularization, and taking the L1-norm of \( W \) helps penalize outlier features by optimizing them towards 0. The value used for \( \lambda \) is 0.1 in LPGEN.

As the L1-norm is not differentiable at zero, its closed form does not exist, so the optimal value, i.e., minimal loss, is obtained by optimization techniques like gradient descent.

The features that are correlated to \( x_1 \) can now be taken to be those \( x_i \)’s for which the coefficient \( w_i \) is non-zero.

**C. Univariate Feature Selection**

Univariate feature selection is another technique for finding correlated features. It is typically faster than L1 regularization for large dimensions. Let us say we want to find features that are correlated to \( x_1 \). We first run linear regression for the target variable \( x_1 \) by considering a single independent variable \( x_j \) at a time, with \( x_j \) ranging over \( x_2, \ldots, x_n \) in turn. Let us say that for a particular variable \( x_j \), linear regression learns the estimate \( \hat{w}_1 + w_j \). We can now define the \( F \)-score of \( x_j \), denoted \( F(x_j) \), to be \( MSR(x_j)/MSE(x_j) \) where \( MSR(x_j) \) is the “mean sum of squares regression” of \( x_j \), defined to be

\[
MSR(x_j) = \sum_{i=1}^m (\bar{d} - (w_1 + w_{dij}))^2
\]

where \( \bar{d} = (\sum_{i=1}^m d_{i1})/m \) is the mean of \( d_{11}, \ldots, d_{m1} \); and 

\[
MSE(x_j) = \sum_{i=1}^m (d_{i1} - (w_1 + w_{dij}))^2 / m - 2
\]

The \( F \)-score is an indicator of how significantly a feature is related to the target variable. The greater the \( F \)-score, the greater its significance.

We now compute the “top-k” features for \( x_1 \) as follows. Let \( \bar{f} \) be the mean \( F \)-score of the variables \( x_2, \ldots, x_n \). We now select those features \( x_j \) whose \( F \)-score is at least as much as \( \bar{f} \), and return these as the features most correlated to \( x_1 \).
subject to: \[ w_1 d_{11} + w_2 d_{12} + b \geq 0 \] \[ w_1 d_{m1} + w_2 d_{m2} + b \geq 0 \] \[ w_1, -w_2 > 0 \] where \( w_1, w_2, b \) are the integer values we want to infer. We place the restriction that these values should be in the range \([-10^3, 10^3]\). The coefficients in the minimization formulas were based on experimental heuristics. The idea behind the choice of coefficients is shown in Figure 3(b). We want to minimize the slope and intercept to learn the cyan-colored line.

Let’s say we try to get \( w_1 \) and \( w_2 \) with equal weights in the minimization, i.e. Minimize \( w_1 - w_2 + b \). Then it will give \( 2x - y + 0 \geq 0 \) (corresponding to the red line), which is not as tight as the line \( 3x - 2y + 0 \geq 0 \) (cyan line). Due to this reason, we gave a bit more weight to the negative coefficient than the positive one (i.e., 0.4 to \( w_1 \), which is positive, and 0.49 to \( w_2 \), which is negative). Apart from this, 1000000 is added to make the total addition positive, as the LP solver does not support negative minimization objectives.

Similarly, we formulate cases (b)–(d) by changing the objective function (4) as follows:

\[
\begin{align*}
\text{case (b)} & : 1000000 + 0.49w_1 + 0.4w_2 + 0.2b \\
\text{case (c)} & : 1000000 - w_1 - w_2 + 0.2b \\
\text{case (d)} & : 1000000 + w_1 + w_2 + 0.2b
\end{align*}
\]

The constraints each model is subject to are (5) and the corresponding version of (6).

We now ask an ILP solver to find optimal values of \( w_1, w_2, b \) for each of the four problem instances, giving us four inequality constraints.

This is repeated for each pair of variables.

### B. Three Dimensional Inequality Invariants

Inequalities in three dimensions are inferred similar to the case of two dimensions. The goal is to infer constraints of the form \( w_1 x + w_2 y + w_3 z + b \geq 0 \), for a given triplet of variables \( x, y, z \) which are satisfied by the data set \( D \). Also, we want to find (heuristically) “tight” constraints. Here apart from six min/max bounds for three variables, we find eight bounding lines corresponding to (a) \( w_1 \) being positive, \( w_2 \) being negative and \( w_3 \) being positive, (b) \( w_3 \) being positive, \( w_2 \) being positive and \( w_1 \) being negative, (c) \( w_1 \) being positive, \( w_2 \) being positive and \( w_3 \) being negative, (d) \( w_1 \) being negative, \( w_2 \) being positive and \( w_3 \) being positive, (e) \( w_1 \) being positive, \( w_2 \) being negative and \( w_3 \) being positive, (f) \( w_1 \) being negative, \( w_3 \) being positive, \( w_2 \) being negative and \( w_3 \) being negative, and (g) \( w_1 \) being negative, \( w_2 \) being negative and \( w_3 \) being negative.

For obtaining tighter bounds in three dimensions, we use the following ILP formulation for case (a) (with \( x, y, \) and \( z \) corresponding to \( x_1, x_2 \) and \( x_3 \) respectively):

\[
\begin{align*}
\text{minimize} & \quad 1000000 + 0.6w_1 + 0.2w_2 + 0.6w_3 + 0.2b \\
\text{subject to:} & \quad w_1 d_{11} + w_2 d_{12} + w_3 d_{13} + b \geq 0 \\
& \quad \ldots \\
& \quad w_1 d_{m1} + w_2 d_{m2} + w_3 d_{m3} + b \geq 0 \\
& \quad w_1, -w_2 > 0
\end{align*}
\]
tool finally outputs a set of terms the tool should consider in the invariants it learns. The max-degree value below in more detail.

Fig. 4. Steps to generate invariant

subject to:

\[
\begin{align*}
\sum_{i=1}^{3} w_i d_1 + w_2 d_2 + w_3 d_3 + b &\geq 0 \\
\sum_{i=1}^{3} w_i d_{2i} + w_2 d_{2i} + w_3 d_{2i} + b &\geq 0 \\
\sum_{i=1}^{3} w_i d_{3i} + w_2 d_{3i} + w_3 d_{3i} + b &\geq 0 \\
\end{align*}
\]

where \(w_1, w_2, w_3, b\) are the integer values we want to infer. We place the restriction that these values should be in the range \([-10^5, 10^5]\).

Similarly, we formulate the cases (b)–(h) by changing the objective function Eq 10 as follows:

- case (b) \(100000 + 0.6w_1 + 0.6w_2 + 0.2w_3 + 0.2b\) 
- case (c) \(100000 + 0.6w_1 - 0.2w_2 - 0.2w_3 + 0.2b\) 
- case (d) \(100000 + 0.2w_1 + 0.6w_2 + 0.6w_3 + 0.2b\) 
- case (e) \(100000 - 0.2w_1 - 0.2w_2 + 0.6w_3 + 0.2b\) 
- case (f) \(100000 - 0.2w_1 - 0.2w_2 - 0.2w_3 + 0.2b\) 
- case (g) \(100000 - w_1 - w_2 - w_3 + 0.2b\) 
- case (h) \(100000 + w_1 + w_2 + w_3 + 0.2b\)

The constraints each model is subject to are (11) and the corresponding version of (12).

We now ask an ILP solver to find optimal values of \(w_1, w_2, w_3, b\) for each of the eight problem instances, giving us eight inequality constraints.

This is repeated for each triple of variables.

VI. OVERALL LPGEN PROCEDURE

We now describe how we put these techniques together in our tool LPGEN. The tool takes as input a data set (typically obtained by collecting execution traces at the head of the loop in a program), over a set of \(k\) named variables. We also supply a max-degree value \(r\), which represents the max degree of terms the tool should consider in the invariants it learns. The tool finally outputs a set \(C\) of candidate invariants. The main steps taken by the tool are shown in Fig. 4, and are described below in more detail.

a) Data preprocessing: Initialize \(C\) to empty. Process the data set as follows:

1) If any variable \(x\) has a constant value \(l\), add \(x = l\) as an invariant to \(C\), and remove \(x\) from the data set. This reduces the computation time and chance of overfitting.
2) If any two variables \(x\) and \(y\) always have equal values, collect \(x = y\) as an invariant and remove one of the variables from the data set.
3) For each remaining variable \(x\), check if \(x \mod l\) (for some integer \(l\)) is constant, and add it as an invariant to \(C\). Also check if \(x\%l\) is constant, and if so add it as an invariant to \(C\).
4) For all pairs of variables \((x, y)\) and \((w, z)\) check if \(\gcd(x, y) = \gcd(w, z)\), and if so add it to \(C\).
5) Generate non-linear terms and data from the remaining variables upto \(r\) degree. For example, if the variables are \((x, y)\) and \(r = 2\), we get \(\{x, y, x^2, xy, y^2\}\). The non-linear terms will help to find non-linear equality invariants. The number of terms from \(k\) variables of degree at most \(r\) is \(n = \binom{k+r}{r} - 1\), which is now the dimension of our modified data set.
6) Finally, generate min and max bounds for each variable, and add them as invariants to \(C\).

b) Learning Equalities: We now apply the linear regression based technique described in Sec. IV on this data set. For each of the \(n\) variables, we first apply L1 regularization and univariate feature selection to find a set of correlated variables, followed by linear regression using these as the independent variables. This gives us \(2n\) equality invariants. Each of these constraints is first checked to see if it satisfies the given data, and if so it is added to \(C\).

c) Learning Inequalities: Next we apply the techniques of Sec. V to obtain inequality constraints on pairs and triples of variables respectively. We considered only linear inequality learning, i.e., applying inequality learning as mentioned in Sec. V only on the original set of variables and not on the higher degree variables. This can be extended to non-linear inequality learning but at the cost of increased computation time and verification complexity. After inferring inequality invariants, they are checked for consistency with the data set before adding them to \(C\).

We now return the set \(C\) as the set of candidate invariants.

VII. EXPERIMENTAL EVALUATION

We have implemented LPGEN in Python. We use the scikit-learn library to perform feature selection and linear regression. We use the PuLP Python linear programming toolkit for inferring inequality invariants.

The aims of our evaluation were threefold:

- (Adequacy) Whether the set of candidate invariants are sufficient to prove pre-post conditions for programs.
- (Strength) How strong are the adequate invariants?
- (Efficiency) How much time does it take to generate the invariants.

With these goals in mind, we evaluated the performance of LPGEN on a variety of pre-post verification benchmark suites.
from SVCOMP [2], and also did the same with three state-of-the-art data-driven invariant generation tools Diakon [5], DIG [16], and GCLN [23]. We first generate execution traces for

each program by running them on manually provided inputs that satisfy the given precondition, and collect data at the head of the loop across iterations and runs. We then run the tools to generate the set of candidate invariants. We then use our implementation of the Houdini algorithm (using the Z3 solver [4]) to find the strongest conjunctive subset that is adequate to prove the programs.

We use SVCOMP benchmarks named nla-digbench (these typically need non-linear arithmetic invariants), loop-invariants (linear invariants) and loop-zilu (mix of linear and non-linear invariants). Most of the programs are single loop programs. For the few programs containing multiple or nested loops, we make sub-problems out of these by considering the invariant of the outer loop as the pre for the inner loop, and the documented invariant of the inner loop as the post condition. The loop body is manually simplified in the case of nested loops for the inductiveness check. For the nla-digbench benchmark, the adequacy and inductiveness conditions are taken from the GCLN implementation [23]. For nested loops data is collected separately for each loop and it is treated as a separate problem for invariant inference. While running Houdini however, the loop body needs to be engineered carefully capturing the semantics of the loop for
<table>
<thead>
<tr>
<th>Program</th>
<th>Daikon</th>
<th>DIG</th>
<th>GCLN</th>
<th>LPGen</th>
</tr>
</thead>
<tbody>
<tr>
<td>benchmark01 Conjunctive</td>
<td>N 1.11</td>
<td>Y 0.86</td>
<td>5/5 0.29</td>
<td>Y 0.02</td>
</tr>
<tr>
<td>benchmark02 Linear</td>
<td>Y 1.20</td>
<td>Y 1.49</td>
<td>Y 18.70</td>
<td>Y 0.56</td>
</tr>
<tr>
<td></td>
<td>Y 1.20</td>
<td>Y 1.49</td>
<td>Y 18.70</td>
<td>Y 0.56</td>
</tr>
<tr>
<td>benchmark03 Linear</td>
<td>Y 1.53</td>
<td>Y 1.99</td>
<td>Y 21.41</td>
<td>Y 0.77</td>
</tr>
<tr>
<td></td>
<td>N 1.48</td>
<td>Y 1.91</td>
<td>Y 43.14</td>
<td>Y 0.92</td>
</tr>
<tr>
<td>benchmark05 Conjunctive</td>
<td>Y 1.41</td>
<td>Y 1.71</td>
<td>Y 17.33</td>
<td>Y 0.77</td>
</tr>
<tr>
<td></td>
<td>Y 1.39</td>
<td>Y 1.82</td>
<td>Y 85.33</td>
<td>Y 7.06</td>
</tr>
<tr>
<td>benchmark06 Conjunctive</td>
<td>Y 1.33</td>
<td>Y 1.54</td>
<td>N 3.46</td>
<td>Y 1.80</td>
</tr>
<tr>
<td></td>
<td>Y 1.44</td>
<td>Y 1.72</td>
<td>Y 10.98</td>
<td>Y 0.70</td>
</tr>
<tr>
<td>benchmark09 Conjunctive</td>
<td>Y 1.18</td>
<td>Y 1.46</td>
<td>Y 1.41</td>
<td>Y 0.50</td>
</tr>
<tr>
<td></td>
<td>N 1.36</td>
<td>Y 1.54</td>
<td>Y 1.78</td>
<td>Y 0.13</td>
</tr>
<tr>
<td>benchmark11 Linear</td>
<td>Y 1.33</td>
<td>Y 1.58</td>
<td>Y 4.58</td>
<td>Y 0.16</td>
</tr>
<tr>
<td></td>
<td>Y 1.33</td>
<td>Y 1.58</td>
<td>Y 4.58</td>
<td>Y 0.16</td>
</tr>
<tr>
<td>benchmark12 Linear</td>
<td>Y 1.23</td>
<td>Y 1.63</td>
<td>Y 7.53</td>
<td>Y 10.69</td>
</tr>
<tr>
<td></td>
<td>Y 1.40</td>
<td>Y 1.64</td>
<td>Y 10.88</td>
<td>Y 0.17</td>
</tr>
<tr>
<td>benchmark14 Linear</td>
<td>Y 1.25</td>
<td>Y 1.38</td>
<td>Y 0.42</td>
<td>Y 0.01</td>
</tr>
<tr>
<td></td>
<td>Y 1.34</td>
<td>Y 1.69</td>
<td>Y 22.30</td>
<td>Y 1.27</td>
</tr>
<tr>
<td>benchmark16 Conjunctive</td>
<td>N 1.19</td>
<td>Y 1.35</td>
<td>Y 3.28</td>
<td>Y 0.16</td>
</tr>
<tr>
<td></td>
<td>Y 1.46</td>
<td>Y 1.85</td>
<td>Y 14.44</td>
<td>Y 0.21</td>
</tr>
<tr>
<td>benchmark17 Conjunctive</td>
<td>Y 1.20</td>
<td>Y 1.48</td>
<td>Y 13.33</td>
<td>Y 0.19</td>
</tr>
<tr>
<td></td>
<td>Y 1.23</td>
<td>Y 1.53</td>
<td>Y 18.63</td>
<td>Y 0.24</td>
</tr>
<tr>
<td>benchmark20 Conjunctive</td>
<td>Y 1.27</td>
<td>Y 1.69</td>
<td>Y 25.93</td>
<td>Y 0.24</td>
</tr>
<tr>
<td></td>
<td>N 1.51</td>
<td>Y 1.66</td>
<td>N 5.14</td>
<td>Y 0.24</td>
</tr>
<tr>
<td>benchmark22 Conjunctive</td>
<td>N 1.30</td>
<td>Y 1.58</td>
<td>Y 1.83</td>
<td>Y 0.14</td>
</tr>
<tr>
<td></td>
<td>N 1.50</td>
<td>Y 1.81</td>
<td>Y 16.35</td>
<td>Y 0.14</td>
</tr>
<tr>
<td>benchmark25 Linear</td>
<td>N 1.10</td>
<td>Y 1.19</td>
<td>Y 0.29</td>
<td>Y 0.01</td>
</tr>
<tr>
<td></td>
<td>Y 1.38</td>
<td>Y 1.73</td>
<td>Y 4.71</td>
<td>Y 0.17</td>
</tr>
<tr>
<td>benchmark27 Linear</td>
<td>Y 1.42</td>
<td>Y 1.61</td>
<td>Y 26.51</td>
<td>Y 0.81</td>
</tr>
<tr>
<td></td>
<td>Y 1.16</td>
<td>Y 2.77</td>
<td>Y 3.40</td>
<td>Y 0.14</td>
</tr>
<tr>
<td>benchmark29 Linear</td>
<td>N 1.25</td>
<td>Y 1.42</td>
<td>N 4.27</td>
<td>Y 0.19</td>
</tr>
<tr>
<td></td>
<td>Y 1.08</td>
<td>Y 1.47</td>
<td>Y 2.24</td>
<td>Y 0.02</td>
</tr>
<tr>
<td>benchmark31 Disjunctive</td>
<td>N 1.25</td>
<td>Y 1.39</td>
<td>Y 6.72</td>
<td>N 0.14</td>
</tr>
<tr>
<td></td>
<td>Y 1.39</td>
<td>Y 1.61</td>
<td>Y 0.12</td>
<td>Y 0.01</td>
</tr>
<tr>
<td>benchmark33 Linear</td>
<td>N 1.35</td>
<td>Y 1.47</td>
<td>Y 0.25</td>
<td>Y 0.01</td>
</tr>
<tr>
<td></td>
<td>Y 1.60</td>
<td>Y 2.19</td>
<td>Y 33.46</td>
<td>Y 1.24</td>
</tr>
<tr>
<td>benchmark35 Linear</td>
<td>N 1.04</td>
<td>Y 1.17</td>
<td>Y 0.10</td>
<td>Y 0.01</td>
</tr>
<tr>
<td></td>
<td>Y 1.30</td>
<td>Y 1.42</td>
<td>Y 2.47</td>
<td>Y 0.01</td>
</tr>
<tr>
<td>benchmark37 Conjunctive</td>
<td>N 1.30</td>
<td>Y 1.46</td>
<td>Y 2.35</td>
<td>Y 0.17</td>
</tr>
<tr>
<td></td>
<td>Y 1.50</td>
<td>Y 1.75</td>
<td>Y 3.73</td>
<td>Y 0.18</td>
</tr>
<tr>
<td>benchmark39 Conjunctive</td>
<td>Y 1.15</td>
<td>Y 1.49</td>
<td>Y 1.82</td>
<td>Y 0.18</td>
</tr>
<tr>
<td></td>
<td>Y 1.56</td>
<td>Y 1.83</td>
<td>N 4.96</td>
<td>Y 0.18</td>
</tr>
<tr>
<td>benchmark40 Polynomial</td>
<td>Y 1.42</td>
<td>Y 1.72</td>
<td>Y 11.80</td>
<td>Y 0.21</td>
</tr>
<tr>
<td></td>
<td>Y 1.23</td>
<td>Y 1.54</td>
<td>Y 11.01</td>
<td>Y 0.21</td>
</tr>
<tr>
<td>benchmark42 Polynomial</td>
<td>N 1.24</td>
<td>Y 1.40</td>
<td>N 5.67</td>
<td>Y 0.18</td>
</tr>
<tr>
<td></td>
<td>N 1.18</td>
<td>Y 1.32</td>
<td>Y 4.21</td>
<td>Y 0.18</td>
</tr>
<tr>
<td>benchmark45 Disjunctive</td>
<td>N 1.42</td>
<td>Y 1.51</td>
<td>N 3.57</td>
<td>Y 0.18</td>
</tr>
<tr>
<td></td>
<td>Y 1.49</td>
<td>Y 1.65</td>
<td>N 22.59</td>
<td>Y 0.96</td>
</tr>
<tr>
<td>benchmark47 Linear</td>
<td>N 1.18</td>
<td>Y 1.32</td>
<td>Y 4.63</td>
<td>Y 0.14</td>
</tr>
<tr>
<td></td>
<td>Y 1.36</td>
<td>Y 1.58</td>
<td>Y 30.15</td>
<td>Y 10.80</td>
</tr>
<tr>
<td>benchmark49 Linear</td>
<td>Y 1.39</td>
<td>Y 1.49</td>
<td>Y 6.25</td>
<td>Y 0.18</td>
</tr>
<tr>
<td></td>
<td>N 1.13</td>
<td>Y 1.26</td>
<td>N 0.17</td>
<td>Y 0.01</td>
</tr>
<tr>
<td>benchmark52 Polynomial</td>
<td>N 1.19</td>
<td>Y 1.33</td>
<td>Y 0.22</td>
<td>Y 0.01</td>
</tr>
<tr>
<td></td>
<td>Y 1.29</td>
<td>Y 1.59</td>
<td>N 2.13</td>
<td>Y 0.19</td>
</tr>
</tbody>
</table>

**TABLE IV**
Performance of tools on the loop-zilu benchmark

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Total Programs</th>
<th>Without Feature Selection</th>
<th>With Feature Selection</th>
</tr>
</thead>
<tbody>
<tr>
<td>nia-digbench</td>
<td>38</td>
<td>25</td>
<td>38</td>
</tr>
<tr>
<td>loop-zilu</td>
<td>57</td>
<td>53</td>
<td>53</td>
</tr>
<tr>
<td>loop-invariants</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>
checking inductive conditions. Moreover, invariants involving gcd and bitwise-and are manually checked because Z3 does not support gcd and sometimes times out for bitwise operations. So we generated the gcd and bitwise-and during the procedure (i.e., pre-processing stage) but manually decided whether to include it in the conjunctive invariant returned by Houdini on the remaining candidate invariants.

Our experiments were performed on Google Colab, with a single CPU and 12 GB of RAM.

The results are summarized in Tables II, III and IV. The Program column specifies the name of the program along with the loop index. For example, “lcm1_2” represents the lcm1 program, and the loop index is 2 as it has 3 different loops within the program. The Adq column represents whether an adequate conjunctive subset was found or not. Column Inv gives the time in seconds taken to infer the set of candidate invariants, and Total gives the total time in seconds for inferring candidate invariants and running the Houdini algorithm.

In the Adq column “Y” represents the fact that an adequate invariant was found, and “N” represents the fact no adequate invariant was found. As GCLN infers invariants non-deterministically, its Adq column shows out of 5 runs (with different random seeds) how many times we were able get an adequate invariant. “TO” in the Inv column mean the tool didn’t compute any candidate invariants within 300 seconds. “ERROR” means that the tool crashed with some error.

Fig. 5 summarizes the total number of programs for which each tool inferred an adequate invariants. Out of 104 program over the three benchmarks we got adequate invariants for 100 of them. The remaining four required disjunctive invariants which are out of the scope of our tool. GCLN was the closest with adequate invariants for 70 programs.

Table V summarizes the total number of programs for which adequate invariants were inferred by LPGEN, with and without using feature selection. When the number of features (i.e. dimension of the data) is high, as in the case of nla-digbench benchmarks, feature selection is important as without using it, LPGEN was able to come up with only 25 adequate invariants out of 38 programs. Both loop-zilu and loop-invariants benchmarks had small dimension of the data and thus got the same number of adequate invariants even without feature selection. But when the dimension increases, feature selection plays an important role, as in the case of nla-digbench. Among the two feature selection techniques, L1 Regularization is more effective, and out of 38 programs of nla-digbench, LPGEN found 37 adequate invariants by using only L1 regularization as a feature selection technique. Only 12 were found to be adequate out of 38 using only the univariate feature selection technique. By combining both, we got adequate invariants for all 38 programs in nla-digbench.

In terms of efficiency in generating invariants, the average time taken by LPGEN to infer candidate invariants is 8.32 s which is around 5x slower than the average time of the fastest tool (Daikon) which is 1.56 s. However we note that the total number of adequate invariants inferred by our tool is almost twice that of Daikon.

The LPGEN tool can be accessed at: https://github.com/Arkesh-Thakkar/LPGen

VIII. RELATED WORK

We classify related work broadly according to whether they are white-box or black-box (depending on whether the use the source of the program or not).

There are several white-box invariant learning techniques that use the program text to come up with invariants are sufficient to prove the correctness of the program. These include [7] and [3] which use constants and expressions from the program text to infer candidate invariants.

Among black-box techniques we consider two categories: techniques that use multi-round learning and techniques that
use single-shot learning. In the first category the ICE-learning line of work features prominently [6], [10], [11]. These techniques are able to learn both conjunctive and disjunctive invariants using decision-tree based learning. However they do not use execution traces of the program, and instead rely on counterexamples provided by a Teacher who has access to the program.

In the second category, where our work falls, the earliest and most prominent work is that of Daikon [5]. Daikon essentially checks whether 75 different types of invariant templates satisfy the trace data, and report those that do as candidate invariants.

Nguyen et al [16], [17] and Sharma et al [21] infer polynomial equalities from trace data, using algebraic equation solving techniques. [16] also infer polynomial inequality invariants using a convex polyhedron approach, and its relaxed version, octagonal inequalities, by creating a convex hull in two dimensions.

Finally, [23] and [20] proposed a novel neural architecture called GCLN and CLN architectures, respectively, to learn loop invariants. They use fuzzy logic based gates called T-norms and T-conorms representing the continuous versions of conjunction and disjunction. The GCLN tool uses a Piecewise Biased Quadratic Unit (PBQU) activation function for inferring inequality invariants.

All these works differ from ours in various aspects. Daikon is limited to generating equality invariants in only three dimensions and based on predefined templates. The DIG approach to finding equality invariants with equation solvers which are costly for finding complex equality invariants, and finding inequality invariants by constructing convex hulls can be exponential in time complexity. The GCLN line of work is nondeterministic in nature due to random dropouts and lack of a proper feature selection technique. We overcome the above limitations as our tool allows equality invariants in n-dimensions and with proper feature selection techniques, namely L1 Regularization and univariate feature selection, we were able to find a good set of equality invariants. For inequalities, we use Linear Programming (LP) solvers for finding inequality invariants in two and three dimensions. LP solvers are much faster than constructing polyhedrons and with the optimization objective given in Sec.V, we were able to find tighter bounds.

IX. CONCLUSION

In this work we have presented a novel combination of techniques, including linear regression, feature selection, and linear optimization, to learn tight invariants from execution trace data. Our experimental evaluation shows that our tool is able to learn strong adequate conjunctive invariants for a variety of programs requiring linear and non-linear predicates, in a reasonably efficient manner.

Going ahead we would like to extend our work to learning disjunctive invariants, in both single shot and multi-round teacher-learner settings.

REFERENCES


73
