Automatic Parameter Tuning for Boogie

Ankit Agrawal

Abstract

Using machine learning we can approximate the effectiveness of command line arguments for Boogie. When using these measures of effectiveness,we can automatically determine which flags to set when performing satisfiability analysis i.e. determining if a certain flag of SMACK tool must be set while using Boogie to check satisfiability of given property by a C program. The flag under consideration directly affects the overall runtime of SMACK needed to perform the check.

1 Introduction

A standard technique in verification is to transform a given program into set of verification conditions whose validity implies the satisfiability of correctness property under consideration. To achieve this, practical approach is:

1. Transform program and proof obligations into intermediate representation
2. Transform intermediate representation to logical formulas

Boogie is intermediate language which can encode verification conditions for Object Oriented Programming. Many languages like *Spec#*, C, Dafny, Java bytecode and Eiffel can be converted to Boogie code so instead of building a parser for each language, we can operate directly on Boogie code and perform verification on code in all these languages.
Using machine learning has it's own benefits:

- Almost everyone is finding ways to incorporate it in their work.
- Machine Learning techniques can learn from failed predictions and is self-learning. It also scales well as the data size increases.
- This needs good amount of training data to generate a model (current SV-COMP repository has 15,000 benchmarks)

Machine learning techniques come with their own difficulties:

- No sure way to know if 15,000 benchmarks are enough for a good model.
- Figuring out "good" features for boogie programs and extract them is a difficult task.

- Predictions are mostly heuristics based so we still need to run the verification tools to confirm the output.
- Figure out a good ML algorithm to train is a difficult choice.

The goal of this project is to figure out the behavior of `/trackAllVars` (a FLAG which can be used while SMACK tool tries to verify a given property on a C program).

---

2 Overview

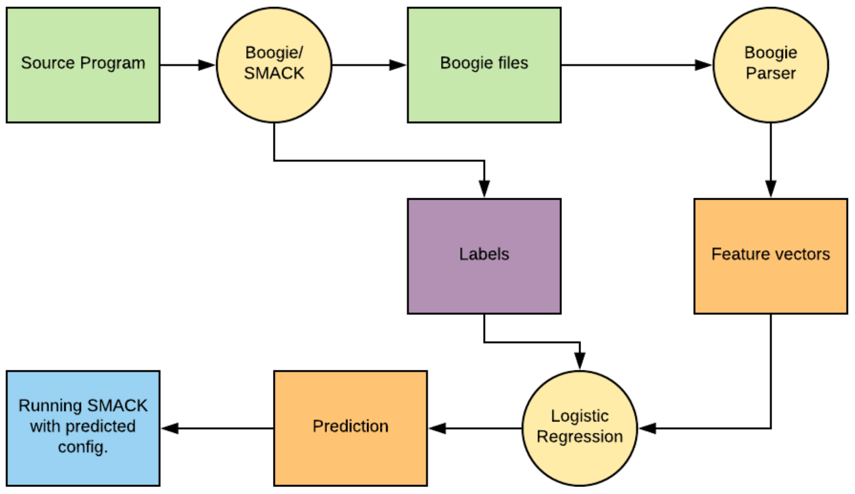


Figure 1: Architecture

The project is divided into the following 3 tasks:

1. Generate Features (section 2.1)
2. Generate Labels (section 2.2)
3. Train a Machine Learning model (section 2.3)

Features are generated from Boogie files (.bpl) whereas labels are generated by running SMACK tool on SV-COMP C benhmarks (.c files). Even more important than selecting a good Machine Learning algorithm is designing quality features. This is also known as feature engineering. There is a tradeoff between designing quality features vs selecting too many features. If we have too many features, we may run into a problem of Overfitting where the algorithm tries

hard to fit a model accurately but is not the most ideal representation of the problem.
For this problem, .bpl files can be generated in following 2 ways:

1. Run SV-COMP benchmarks on Boogie.
2. Run SV-COMP benchmarks on SMACK.

Since we are using SMACK to generate the labels and SMACK generates Boogie files, we used SMACK to generate the boogie files. Once all the .bpl files are generated, we use the feature extraction tool described in section 2.1 to extract the features. In generating .bpl files, we do not need the full verification run by SMACK so we set the SMACK timeout to 3 seconds, ust enough to generate the necessary .bpl files.

2.1 Generating features

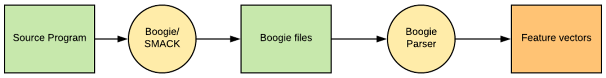
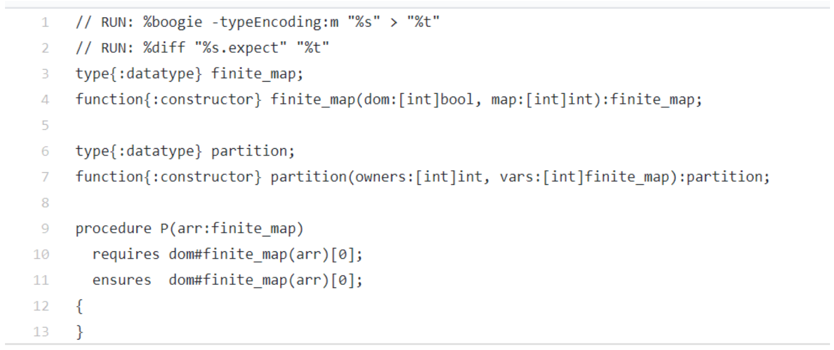


Figure 2: Feature Generation Data Flow

Once all the .bpl files are generated, we use the Boogie Feature Extractor to extract the features. Following is a sample of Boogie file.

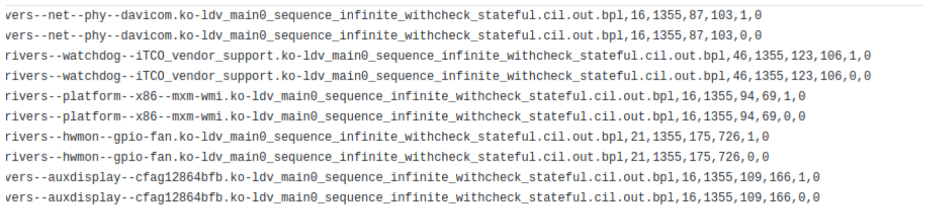


```
// RUN: %boogie -typeEncoding:m "%s" > "%t"
// RUN: %diff "%s.expect" "%t"
type{:datatype} finite_map;
function{:constructor} finite_map(dom:[int]bool, map:[int]int):finite_map;

type{:datatype} partition;
function{:constructor} partition(owners:[int]int, vars:[int]finite_map):partition;

procedure P(arr:finite_map)
  requires dom#finite_map(arr)[0];
  ensures  dom#finite_map(arr)[0];
{
}
```

Figure 3: Sample Boogie code

For this project, we used basic counting features like the number of functions, variables, constant type declarations, number of functions with implementations A singular matrix is a matrix with no inverse i.e. the matrix is not invertible (mostly caused when 2 rows are similar). Lets say that $r1, r2$ are 2 rows of a

matrix A and $r1 = c * r2$ where c is some constant then A has no inverse.
The features need to be selected carefully to avoid creating a singular matrix. In Boogie, number of functions and number of implementations of functions can be different because they are declared and implemented separately and some functions don't have an implementation (body). The pattern above is



```
vers--net--phy--davicom.ko-ldv_main0_sequence_infinite_withcheck_stateful.cil.out.bpl,16,1355,87,103,1,0
vers--net--phy--davicom.ko-ldv_main0_sequence_infinite_withcheck_stateful.cil.out.bpl,16,1355,87,103,0,0
rivers--watchdog--iTCO_vendor_support.ko-ldv_main0_sequence_infinite_withcheck_stateful.cil.out.bpl,46,1355,123,106,1,0
rivers--watchdog--iTCO_vendor_support.ko-ldv_main0_sequence_infinite_withcheck_stateful.cil.out.bpl,46,1355,123,106,0,0
rivers--platform--x86--mxm-wmi.ko-ldv_main0_sequence_infinite_withcheck_stateful.cil.out.bpl,16,1355,94,69,1,0
rivers--platform--x86--mxm-wmi.ko-ldv_main0_sequence_infinite_withcheck_stateful.cil.out.bpl,16,1355,94,69,0,0
rivers--hwmon--gpio-fan.ko-ldv_main0_sequence_infinite_withcheck_stateful.cil.out.bpl,21,1355,175,726,1,0
rivers--hwmon--gpio-fan.ko-ldv_main0_sequence_infinite_withcheck_stateful.cil.out.bpl,21,1355,175,726,0,0
vers--auxdisplay--cfag12864bfb.ko-ldv_main0_sequence_infinite_withcheck_stateful.cil.out.bpl,16,1355,109,166,1,0
vers--auxdisplay--cfag12864bfb.ko-ldv_main0_sequence_infinite_withcheck_stateful.cil.out.bpl,16,1355,109,166,0,0
```

Figure 4: Sample for feature vectors

{Filename} {no. of implementations} {no. of variables} {no. of constants} {no. of type declarations} {no. of functions} {/Trackallvars status}.
`/TrackAllvars status` is a special feature not extracted from the .bpl files. It is a SMACK Flag which affects the verification runtime to verify property for a given C program. We tried both, Keeping the `/trackAllVars Flag=ON` (0) and OFF (1). This was added as an extra feature because it is difficult to know if it is beneficial to turn the flag ON or keep it OFF for unknown C programs.

2.2 Generating Labels

In the previous section, we saw that `/TrackAllVars` was a special feature that we added. In this section, we explain in detail how the value of this feature was set and how the final labels were generated. In figure 5, we see how the

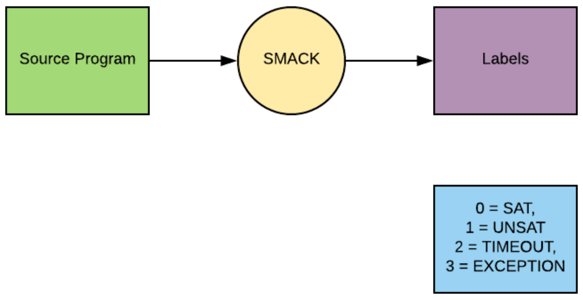
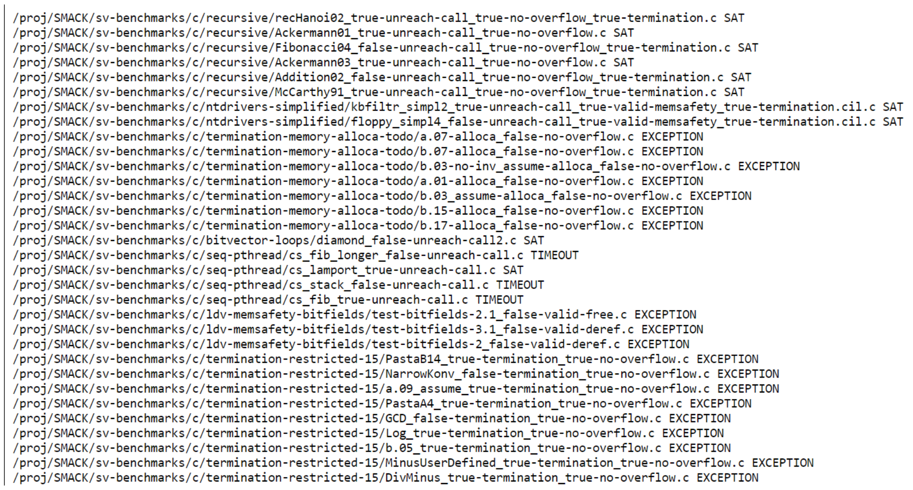


Figure 5: Label Generation Data Flow

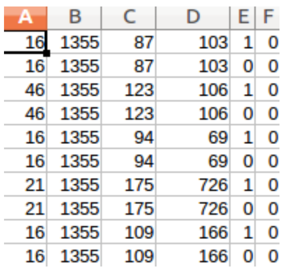
C programs from SV-COMP repository are run through SMACK. We ran 2 instances of SMACK for each input file, one with `/trackAllVars=ON` and other with `/trackAllVars=OFF`. The TIMEOUT=900 seconds i.e. SMACK will either verify the property to be SAT/ UNSAT in 900 seconds or return a label TIMEOUT. Return EXEPTION when SMACK terminates with error. In this



```
/proj/SMACK/sv-benchmarks/c/recursive/recHanoi02_true-unreach-call_true-no-overflow_true-termination.c SAT
/proj/SMACK/sv-benchmarks/c/recursive/Ackermann01_true-unreach-call_true-no-overflow.c SAT
/proj/SMACK/sv-benchmarks/c/recursive/Fibonacci04_false-unreach-call_true-no-overflow_true-termination.c SAT
/proj/SMACK/sv-benchmarks/c/recursive/Ackermann03_true-unreach-call_true-no-overflow.c SAT
/proj/SMACK/sv-benchmarks/c/recursive/Addition02_false-unreach-call_true-no-overflow_true-termination.c SAT
/proj/SMACK/sv-benchmarks/c/recursive/McCarthy91_true-unreach-call_true-no-overflow_true-termination.c SAT
/proj/SMACK/sv-benchmarks/c/ntdrivers-simplified/kbfiltr_simpl2_true-unreach-call_true-valid-memsafety_true-termination.cil.c SAT
/proj/SMACK/sv-benchmarks/c/ntdrivers-simplified/floppy_simpl4_false-unreach-call_true-valid-memsafety_true-termination.cil.c SAT
/proj/SMACK/sv-benchmarks/c/termination-memory-alloca-todo/a.07-alloca_false-no-overflow.c EXCEPTION
/proj/SMACK/sv-benchmarks/c/termination-memory-alloca-todo/b.07-alloca_false-no-overflow.c EXCEPTION
/proj/SMACK/sv-benchmarks/c/termination-memory-alloca-todo/b.03-no-inv_assume-alloca_false-no-overflow.c EXCEPTION
/proj/SMACK/sv-benchmarks/c/termination-memory-alloca-todo/a.01-alloca_false-no-overflow.c EXCEPTION
/proj/SMACK/sv-benchmarks/c/termination-memory-alloca-todo/b.03_assume-alloca_false-no-overflow.c EXCEPTION
/proj/SMACK/sv-benchmarks/c/termination-memory-alloca-todo/b.15-alloca_false-no-overflow.c EXCEPTION
/proj/SMACK/sv-benchmarks/c/termination-memory-alloca-todo/b.17-alloca_false-no-overflow.c EXCEPTION
/proj/SMACK/sv-benchmarks/c/bitvector-loops/diamond_false-unreach-call2.c SAT
/proj/SMACK/sv-benchmarks/c/seq-pthread/cs_fib_longer_false-unreach-call.c TIMEOUT
/proj/SMACK/sv-benchmarks/c/seq-pthread/cs_lamport_true-unreach-call.c SAT
/proj/SMACK/sv-benchmarks/c/seq-pthread/cs_stack_false-unreach-call.c TIMEOUT
/proj/SMACK/sv-benchmarks/c/seq-pthread/cs_fib_true-unreach-call.c TIMEOUT
/proj/SMACK/sv-benchmarks/c/ldv-memsafety-bitfields/test-bitfields-2.1_false-valid-free.c EXCEPTION
/proj/SMACK/sv-benchmarks/c/ldv-memsafety-bitfields/test-bitfields-3.1_false-valid-deref.c EXCEPTION
/proj/SMACK/sv-benchmarks/c/ldv-memsafety-bitfields/test-bitfields-2_false-valid-deref.c EXCEPTION
/proj/SMACK/sv-benchmarks/c/termination-restricted-15/PastaB14_true-termination_true-no-overflow.c EXCEPTION
/proj/SMACK/sv-benchmarks/c/termination-restricted-15/NarrowKonv_false-termination_true-no-overflow.c EXCEPTION
/proj/SMACK/sv-benchmarks/c/termination-restricted-15/a.09_assume_true-termination_true-no-overflow.c EXCEPTION
/proj/SMACK/sv-benchmarks/c/termination-restricted-15/PastaA4_true-termination_true-no-overflow.c EXCEPTION
/proj/SMACK/sv-benchmarks/c/termination-restricted-15/GCD_false-termination_true-no-overflow.c EXCEPTION
/proj/SMACK/sv-benchmarks/c/termination-restricted-15/Log_true-termination_true-no-overflow.c EXCEPTION
/proj/SMACK/sv-benchmarks/c/termination-restricted-15/b.05_true-termination_true-no-overflow.c EXCEPTION
/proj/SMACK/sv-benchmarks/c/termination-restricted-15/MinusUserDefined_true-termination_true-no-overflow.c EXCEPTION
/proj/SMACK/sv-benchmarks/c/termination-restricted-15/DivMinus_true-termination_true-no-overflow.c EXCEPTION
```

Figure 6: Example of SMACK results

step, we also add an extra feature `/trackAllVars` depending on which file it was merged from. So, for each input file, we had 2 feature vectors only differing in the column for `/trackAllVars`. This gave us the features and labels we need to generate a Machine Learning model. For some files, setting the `/trackAllVars flag=ON/OFF` didn't matter as they resulted in same label.



A	B	C	D	E	F
16	1355	87	103	1	0
16	1355	87	103	0	0
46	1355	123	106	1	0
46	1355	123	106	0	0
16	1355	94	69	1	0
16	1355	94	69	0	0
21	1355	175	726	1	0
21	1355	175	726	0	0
16	1355	109	166	1	0
16	1355	109	166	0	0

Figure 7: First 10 instances of $\hat{X}, \hat{Y}$

2.3 Train a Machine Learning model

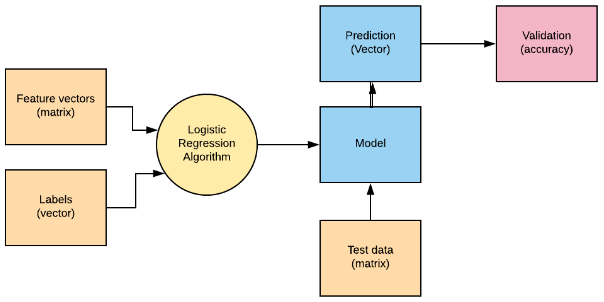


Figure 8: ML Data Flow

Since the possible labels are 0,1,2,3, we use a well known classification algorithm in machine learning known as **logistic regression**. We treat this problem as multi-class classification problem and use the algorithm known as **one vs all strategy**. **Notations:** Feature Matrix = X, Labels vector = Y Since we have 5 features, we generate a vector of length 6.

$$\theta = [\theta_0\theta_1...\theta_4\theta_5]$$

We define our prediction $h_\theta(x)$ as,

$$h_\theta(x) = g(\theta^T x)$$

$$h_\theta(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2 + ... + \theta_5 x_5)$$

Where g(z) is the sigmoid function defined as:

$$g(z) = \frac{1}{1 + e^{-z}}$$

We define our label for i^{th} test data, $x^{(i)}, y^{(i)}$ as, if

$$h_\theta(x^{(i)}) \geq 0.5, y^{(i)} = 1$$

if

$$h_\theta(x^{(i)}) < 0.5, y^{(i)} = 0$$

We experiment with 700 .c files from SV-benchmarks so X is a 700x6 matrix and Y is 700x1 vector. From this we create our training data of size 500 i.e. $\hat{X}$ is 500x6 matrix and $\hat{Y}$ is 500x1 matrix and test data of size 200 i.e. $\bar{X}$ is 200x6matrix and $\bar{Y}$ is 200x1 matrix. An exampe of $\hat{X}, \hat{Y}$ can be found in figure 7. The cost function for logistic regression is:

$$J(\theta) = \frac{-1}{m} \sum_{i=1}^{m} -ylog(h_\theta(x)) - (1-y)log(1 - h_\theta(x))$$

We want to compute the vector θ such that we $\min_{\theta} J(\theta)$ We can use gradient descent performing simultaneous updates to vector θ in the following way:

$$\theta_j := \theta_j - \alpha \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

The problem with performing gradient descent is that it depends on the parameter α which controls how slow or quick we converge to the local minimum. Instead of gradient descent, we use a numerical computing technique known as BFGS optimization to compute the θ which computes minimizes $J(\theta)$

We obtain 95% accuracy on our test data when we compare the results of our predictions to that of the original labels.
Example: For an unknown test file: We run the above model for /trackAllVars=0 and /trackAllvars=1, and let's say we obtain foollowing combinations: In table 1, if row 3 occurs, then we need can run SMACK only once and thus, we

`/trackAllVars=ON`	`/trackAllVars=OFF`	**Choice**
SAT/UNSAT	TIMEOUT/ EXCEPTION	ON
TIMEOUT/ EXCEPTION 2	SAT/UNSAT	OFF
TIMEOUT/ EXCEPTION 3	TIMEOUT/ EXCEPTION	Don't Care
SAT	UNSAT	$1, 0$
UNSAT	SAT	$1, 0$

Table 1: Status of /trackAllVars

can save overall runtime of SMACK. Whereas if one of the 2 results into SAT/ UNSAT then we activate the FLAG accordingly and still save runtime by not exploring all combinations. The worst case for this technique would occur in last 2 rows of table 1. In such case, we have no choice but to try out both settings.

---

3 Conclusion

The θ we received was $[-0.000001 - 0.000023 - 0.001635 - 0.0001450.000068 - 0.000001]$ and we achieved a 95% success in predicting the labels correctly. With the results from this report, we can see some hope in incorporating Machine Learning techniques to Software Verification tools to not necessarily perform the verification itself, but to optimize the parameters for verification tools to reduce the runtime significantly by not exploring over all possible parameter combinations for each input file. The reduction in number of TIMEOUT is a sign of a good verification tool.
The source code can be found at
`https://github.com/ankit--agrawal/SV_course/tree/master/Project/source_code` The results are reproducible and only the .bpl files are not available on the Github repository because they were large files.

---

4 References

References

[1] Demyanova, Yulia and Veith, Helmut and Zuleger, Florian *On the concept of variable roles and its use in software analysis.* Formal Methods in Computer-Aided Design (FMCAD), 2013. IEEE

[2] Andrew Ng *Machine Learning.* Coursera

[3] Demyanova, Yulia and Pani, Thomas and Veith, Helmut and Zuleger, Florian *Empirical software metrics for benchmarking of verification tools.* Formal Methods in System Design, 50.2-3 (2017)

[4] Leino, K Rustan M *This is Boogie 2.* Manuscript KRML, 178.131 (2008)