COMP 400 Report version 2

Adam Weiss

April 2021

Prioritizing SMT Bug reports by Learning from Historical Patches

McGill University

Thank you to my supervisor Dr. Xujie Si without whom this would not have been possible.

Abstract

Satisfiability Modulo Theory (SMT) solvers are widely used in academic computer science and industrial settings, it is therefore imperative that they be as bug free as possible. Project Yin-Yang[2] is an automated system for discovering bugs in such systems, namely Z3 and CVC4. This project was highly successful, identifying more that 1000 bugs in total, however this approach produced too many bug reports, including some about bugs which are not necessarily relevant. This has caused great annoyance for the developers. Therefore in my COMP 400 I worked on a machine learning based system for categorizing bug reports as high or low priority.

Introduction

Yin-Yang is one of many automated bug discovery systems, known as "fuzzers", and while identifying bugs is a useful result, not all bug reports are relevant/worth pursuing, since some may be a lower priority and there is a limited amount of time to devote to solving bugs. Lead developer Nikolaj Bjorner echoed this sentiment [1]. Some bugs, like ones with SMT formulas a user would be very unlikely to come up with, are lower priority, while bugs in features people use on a regular basis are higher priority. In this project I was able to use data from the bug reports on Github to create a Machine Learning based classification system.

Data

Bug report source

As Z3 and CVC4 are open source, their corresponding bug reports are available on Github, additionally, the Yin-Yang project maintains a list of bugs they've identified, and this list is a list of links to Github issues.

I was able to use Github's REST API accessed through a Python script running shell commands to download these issues, and read then process the bug reports as necessary, for example removing duplicate reports. As the end goal is a supervised Machine Learning system, I needed to classify all

the reports as high or low priority. The number of reports was such that doing this by hand would have been prohibitively time consuming, so instead I developed a heuristic based on factors such as:

- How long before the bug was addressed ?
- Was the bug ever addressed?
- Was a bug marked as it won't be dealt with? (This information was from Project Yin-Yang's page)

Categorization

I decided that if a bug had existed for more than 6 months and was not resolved, it was likely not a priority. I then was able to classify all the bugs as high priority or low priority, and had data I could input into my Machine Learning model. This shall be referred to as round 1 of data collection.

Logging

After seeing the results of round 1, it was then decided that supplying the model with a snapshot of the code itself may be a useful way to improve performance in an accuracy sense. The size of the codebases for both Z3 and CVC4 were too large for a time-appropriate snapshot to be directly input into a neural network, as such we decided providing an execution trace would be the approach we take.

In order to provide said execution trace, we decided to build Z3 and CVC4 using Clang, with it's tracing instrumentation, known as fxray, turned on. The build system for CVC4 was so complex, that an approach to instrumenting it could not be determined with that feature. Subsequently only Z3 data was collected. The build system for Z3 was far simpler, and a flag could be added in a configuration file (config.mk) to enable it. Ultimately, this was unsuccessful as well because using the flag and changing the appropriate shell variables, and confirming that the instrumentation code was present in the executable, the Z3 executable never produced a log. This behavior warrants future investigation as we were successful in instrumenting smaller programs.

Clang pass

Subsequently, a new approach was taken, to write our own instrumentation pass for Clang. I wrote the instrumentation pass and designed it to log every function call, ultimately this was not possible as the strings representing some function names were null pointers and would cause a segmentation fault in the compilation of Z3.

The pass worked as follows, it was an opt pass which would iterate through each line of IR and if it encountered a function call, it would insert a call to a logging function before the line that logged the name of the function called, and the parameters it was called with.

I therefore decided to simply skip logging those calls as they were quite rare. Having to run an additional pass was not compatible with the Z3 build system provided so I had to write my own build script in Python that was based on the script provided by Microsoft to generate a Makefile (mk_util.py). However there were some notable differences beyond the inclusion of an instrumentation pass, namely that I did not build support for language bindings or the provided test suite.

Generating data from the logging

Following the development of this program, an instrumented Z3 executable was produced. Despite the bugs being from several different versions of Z3, only one executable was produced as building each version automatically would have taken a prohibitively long time in addition to posing a large technical challenge, namely that the build system that worked for one version of Z3, did not necessarily work for another.

While a log file was produced, the files were impractically huge, e.g. running the program on no input produced a file that was several kilobytes, and verifying simple arithmetic statements such as $2+2=4$ ran for an undetermined amount of time, I ended execution after 15 minutes, and a 1.7 GB log file was produced. This approach to logging would be impractical for any even

remotely complex execution trace.

We theorized that the "interesting" parts of execution were likely higher level functions relating to how the program made decisions as a whole. Therefore a script was used to examine the Z3 executable and produce a directed graph of possible function calls. The script to produce the intial graph was taken from here :https://reverseengineering.stackexchange.com/questions/9113/how-can-i-generate-a-call-graph-from-an-unstripped-x86-linux-elf, the rest are mine. The graph was then traversed with Breadth First Search, and only function calls at or above a certain depth, 5, were recorded, and the instrumentation pass was updated to only log those calls. The logs produced were several orders of magnitude smaller but still had some room for improvement.

One potential flaw in this approach was that the names produced by examination of the executable and the names of the functions as seen during the Clang pass were not usually identical so a "fuzzy" match had to be made.

Firstly with a GNU tool called c++filt, the names could be changed to a friendlier format which allowed for easier processing. It was deemed unlikely that the source of the bugs was ever the C++ standard library, so all calls to that were stripped from the log, as were debugging information calls inserted by Clang.

It should be noted that the SMT2-lang code provided to observe the executable's behavior on was extracted from the bug reports posted.

Subsequently, a script was created to generate a log for each bug report. Unfortunately on 30 samples could be generated as after that, so much memory $>$ 20 GB was allocated that the host machine became unusual. Thus 30 logs, some of which were high priority and some of which were low priority were generated.

The log files have each call on a separate line, and are in chronological order. Additionally they are perceived by the neural network as as a series of tokens.

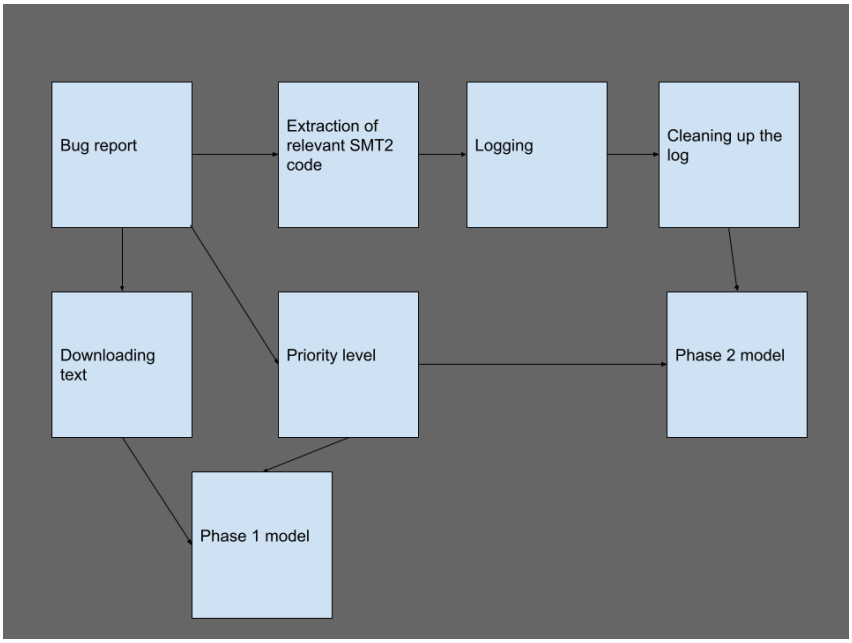

Figure 1: The data pipeline

Model

Device

All models were evaluated on Graham at Compute Canada on one Nvidia V100 GPU and 32GB of RAM, the CPU may have been either an Intel Xeon Gold 5120 or Intel Xeon Gold 6248 depending on which cluster a given job was assigned to. this yielded excellent performance as in phase 1, models generally trained in less than 3 minutes, and in phase 2 it usually took less than 7.

Goal

As the primary goal of this project was to classify textual data, all models presented center around a recurrent neural network (RNN) and use a binary cross-entropy loss function. Additionally to avoid overfitting, all models had dropout and recurrent dropout where possible and 5 fold cross validation was employed. The materials providing information on how to setup a RNN in

Tensorflow recommended Adagrad as the optimizer, so that was used. Furthermore to control for a random variable, the same seed was used for numpy across trials. Additionally, the models were highly performant, generally finishing training even with a large number of epochs in under 5 minutes, an important part of this was using a mixed precision policy in order to take advantage of Tensor Cores in NVIDIA GPUs with CUDA 7.0 or greater, as well ass choosing the sizes of layers to be multiples of 8 when possible.

Initial model

This series of models, for CVC4 and Z3 issue text to priority classification was highly successful, generally achieving accuracies in the $70-75\%$ range.

Layer (type)	Output Shape	Param #
embedding (Embedding)	(None, 500, 512)	512000
lstm (LSTM)	(None, 500, 256)	787456
lstm_1 (LSTM)	(None, 128)	197120
dense (Dense)	(None, 1)	129

Figure 2: The architecture of a network used on the first round of data

After one epoch, this model converged to a validation accuracy of .7548, and a training accuracy of .7216, a higher validation accuracy than training accuracy indicates that a model is likely training well and not overfitting.

Model for log data

A variety of models were employed in analyzing the log data, however their performance was much worse, generally in the $42-57\%$ range.

An identical model the one seen in the previous section achieved a validation

accuracy of .4286 and a training accuracy of .1667, indicating that although overfitting is not occuring, the model is likely insufficiently powerful.

Another approach to architecture was taken, namely the size of the embedding layer was increased to accommodate the longer input data, leading to the following network architecture:

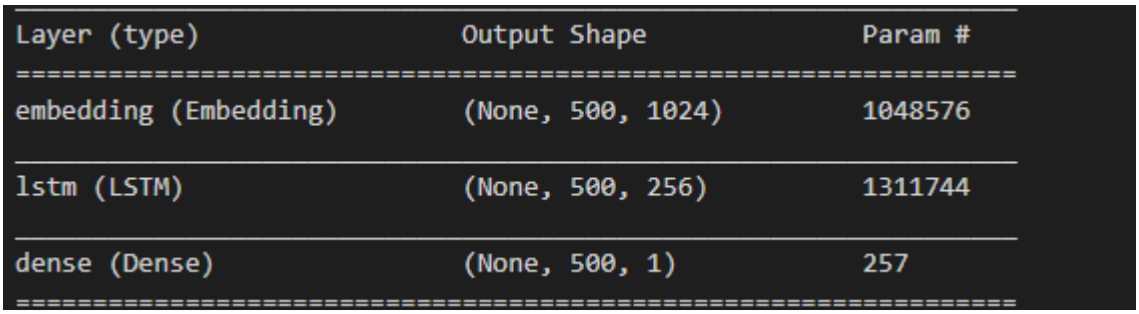

Figure 3: A network with a larger embedding layer

This model was overfit, converging to a training and validation accuracy of .5431 and .4534 after 1 epoch respectively.

This is not an exhaustive exploration of the architectures tested, such an exploration can be found in the project's files on Github, however it does illustrate the best models, and the shortfalls of the worst ones.

Regrettably, creating a useful plot was not possible as Tensorflow can only log loss in epoch time steps, and the models usually converged within 1 epoch.

Discussion of results

The data collection clearly shows that designing a robust logging system for large programs that outputs data in a format usable by a neural network is possible, as is the creation of a system to automatically generate the appropriate log given a bug report.

Additionally, the phase 1 results indicate that examination of bug report text by neural network can be be a viable approach to classification of bug

reports into low and high priority.

Finally, although the accuracies achieved in phase 2 were poor, the use of log data in bug classification cannot be ruled out as a viable approach. Firstly the sample size in phase 2 was extremely small, making it difficult for the model to learn rules accurately. Secondly, the samples were extremely complex due to their length, which may have made learning difficult.

Future work

There are a variety of areas in which future work can proceed, ranging from improved data collection techniques to model refinements and entirely new models.

- Integrating bug report heuristics
- Exploring different function call depths
- Tighter function name matching
- Random logging of functions
- Fixing the memory error for a greater sample size
- Building CVC4 with logging
- Improving the classification heuristic
- Expanding to bugs not found by Project Yin-Yang
- Integration of phase 1 and phase 2 style models
- Comparison of phase 1 and phase 2 models
- Richer categories

The bug reports available on Github contain information that was not integrated into phase 1 models, such as replies to the initial posts. These posts may contain language that is relevant to the classification of bug reports as

high or low priority as well as their volume being a potential indicator of priority.

The decision to log all function calls at depth 5 or lower was rather arbitrary, and exploring different call logging depths may yield better results in phase 2 style models.

The necessity of fuzzy matching meant that some functions that should have been logged were not, and they may have contained important information that was lost, thus adversely affecting model performance, therefore it would be a goal in the future to improve the name matching in order to generate better quality logs.

Assuming the relevant behavior in terms of function calls happened at a low depth is unproven, and random sampling of function calls may allow for logging to capture a more useful snapshot of behavior without introducing impractically large sample sizes.

Fixing the memory issue would render a larger number of samples available for learning from in log based models, and presumably improve their performance.

In a similar vein, being able to log execution traces through CVC4 would create a greater and more diverse set of samples for log based models to learn from.

While some details in the classification heuristic have strong evidence for being correct, e.g. a bug that is marked as something that will not be fixed is likely a low priority, some other criteria do not have such strong evidence, e.g. how long a bug's lifetime is, although experimentally 6 months appeared to produce good result, some other number may work better.

The bug reports examined in this project were solely those pertaining to bugs found by Project Yin Yang, this however is a small subset of bugs found in both Z3 and CVC4, and for this methodology to be extensible to other pieces of software, a more diverse set of bugs would need to be examined.

In order to boost the predictive power, a third type of model could be created which would take as input the execution logs as well as the bug report text. Its architecture may be the following:

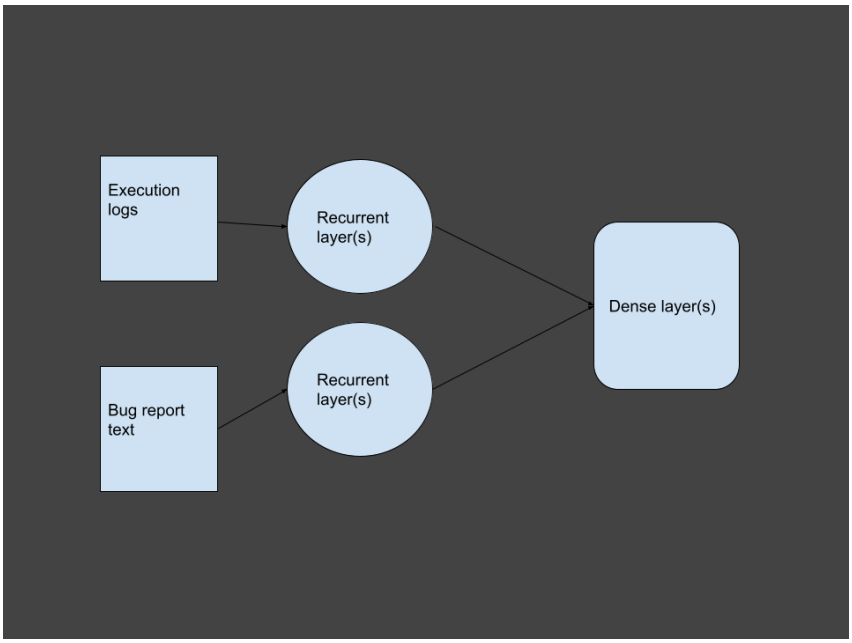

Figure 4: How the architecture of the network may look

This architecture was chosen as passing both texts through some sort of recurrent layer is still critical as is running that result through one or more dense layers for classification, however this may not be the ideal architecture. This would be a fairly straightforward modification as Tensorflow allows for the easy creation of this sort of network.

A comparison of phase 1 and phase 2 models would entail looking the sets of bugs classified by each model and answering questions such as: Are the sets similar? Is there evidence of a subset relation?

Designing a classification system with a richer set of classes may be useful as it better reflects how real programmers think of issues, not as just

high priority or low priority, but a wide variety of priority levels. Doing this would require a new (and more complicated heuristic for classification). Additionally, the neural networks would need a new loss function such as multi category cross entropy.

Notes for the reader

All code is available on the following Github repository : https://github.com/AdamW1002/COMP400

References

[1] Nikolaj's complaint.

[2] Project yin yang.