Version Space Learning for Verification on Temporal Differentials

Mark Santolucito
Yale University
New Haven, Connecticut 06511
Email: mark.santolucito@yale.edu

Ruzica Piskac
Yale University
New Haven, Connecticut 06511
Email: ruzica.pisac@yale.edu

I. Introduction

Machine learning can be a powerful tool for for software analysis and verification [1], [2], [3], [4]. However, many popular machine learning algorithms, such as neural nets and n-gram models, are not designed to provide simple justifications for their classification results. While effective in practice, the lack of justification for the results limits the applicability. In the case of error detection, the system should not only report which files have errors, but also identify the errors. Version space learning is a learning strategy for logical constraint classification [5], which can be used to easily provide clear error messages.

Extensions to version space learning have been used in various software analysis techniques, such as programming-by-example [1], invariant synthesis [4], and error detection [2]. In previous work we used machine learning to verify configuration files by automatically learning a language model from a set of correct examples [2]. We give a formal definition of the algorithm in terms of version space learning to show how clear error messages can be produced with this technique.

Although the error messages are clear, the approach used previously produces a high false positive rate (marks correct files as incorrect). To decrease this, we propose an extention to the algorithm to handle sets of both incorrect and correct examples, which have a partial order. A partial order is common in learning sets for machine learning program analysis, as code does not exist in isolation, but changes over time with development. Any analysis that makes use of code from a version control system like Github will have this property. We predict this algorithm has the potential to significantly reduce the false positive rate.

To test this approach in practice, we plan to implement our algorithm to check for TravisCI configuration errors. TravisCI is a continuous integration tool connected to Github that allows programmers to automatically run their test suite on every code update (commit). A user adds a configuration file to the repository that enables TravisCI and specifies build conditions, such as which compiler to use, which dependencies are required, and a set of benchmarks to test. This ensures the tool can always be automatically built correctly on a fresh machine.

A recent usage study of TravisCI found that 15-20% of failed TravisCI builds are due to "errors" - which means the configuration file was malformed and the software could not even be built [6]. Using the data from [6], we can also learn that since the start of 2014, approximately 88,000 hours of server time was used on TravisCI projects that resulted in an error status. This number not only represents lost server time, but also lost developer time, as programmers must wait to verify that their work does not break the build. If these malformed projects could be quickly statically checked on the client side, both TravisCI and its users could benefit.

II. Version Space for Verification

ConfigC [2] learns a classification model from a set of correct configuration files, then reports misconfigurations. Specifically, ConfigC uses version space learing to identify errors in configuration files for MySQL. Version space learning builds a logical constraint model for binary classification, or membership in a set [5]. Traditional version space learning builds a model using a series of disjunctions from a set of predefined hypotheses. By instead restricting the model to a series of conjunctions, ConfigC can not only flag misconfigurations, but also give the points of failure for non-membership.

Our definition of a configuration file is a file that can be transformed to an ordered list of $(keyword, value)$ pairs, called $Line$s. To build the model, ConfigC take a single configuration file, C, and derives the set of all possible relations from each file, $M(C)$. The classes of relations must be provided by the user as templates. A template is a function taking some number of lines and determining a relation on the keywords of the lines ($Line^n \rightarrow Rel_{k1,..,kn}$). In the case of ordering rules, this function will take two lines and indicate whether the first keyword should come before, after, or have no relation to the second. ConfigC features the following default rules.

Template	Input	Relation
ordering	(Line,Line)	Before | After | None
integer relation	(Line,Line)	$< \mid == \mid >$
type	(Line,Line)	String | Int | Filepath | IP
missing entry	(Line,Line)	Required | Not

It is assumed if a file is correct, all relations in that file are in the set of necessary relations for any other file to be correct ($Correct(C) \implies \forall r \in M(C), r \in Nec$). In this way,

the initial model is built by creating the strongest conditions for a correct file, called the *specific boundary* in version space learning. This model is then iteratively relaxed as more examples are seen, a process called *candidate elimination*. Since we maintain the strongest condition for correctness, this approach will identify many correct files as incorrect, called a high false positive rate. This is a problem when a user is then asked to manual review many files for errors, when the errors are not truly a problem.

When the system is run on a new file, one of the relations from the templates will not be satisfied. To provide an error message,the system only needs to report which relation is not satisfied. For example, ConfigC might output `Ordering Error: Expected "extension mysql.so" before "extension recode.so"`.

III. Learning from Temporal properties

ConfigC produces clean error messages, but has a high false positive rate. However, ConfigC only analyzes correct configuration files, and does not consider any ordering between these correct files. We present an extension to ConfigC to decrease the false positive rate in ConfigC by learning on both correct and incorrect examples, as well as temporal structure of these examples. Our solution is to buid a logical formula representing the entire history of examples, and use a SMT solver to find a classification model.

Since a TravisCI configuration file is dependent on the code it is trying to build, we must consider a more general sense of configuration file. We will call this a program summary P_t, which is a representation of the repository which contains the information relevant to the learning process. The subscript on P_t is a timestamp tag based on the ordered git commit history. In the case of TravisCI, this include the `.travis.yml` file, as well as code features that may effect build status, such as programming language and a list of imported libraries. The summary must be *sufficently detailed*, that is it must contain every piece of information that might lead to a build error.

From this summary we can then build a model $M(P_t)$, as in ConfigC, which is the full set of possible relations derivable from the program summary. Again, the user must provide templates for the learning process.

Since we now consider both positive and negative examples, we will denote the build status of P_t with $S(P_t)$. In this application, we consider all non-erroring build status to be passing, denoted $Pass$ and otherwise Err. For brevity, we denote sequences of build statuses with the following notation:

$$S(P_t) = Pass \wedge S(P_{t+1}) = Err \implies S(P_{t,t+1}) = PE$$

In contrast with ConfigC, we now consider both positive and negative examples and so must introduce the *general boundary*. The general boundary is the dual of the specific boundary, and is the most relaxed requirement for a positive classification. We denoted specific boundary as the set of necessary relations Nec, and now denote the general boundary as the set of breaking relations Br. With this notation, we can formally express the requirement that the program summary is sufficently detailed.

$$\forall S(P_t) = Err, \exists r \in M(P_t), r \in Br \tag{1}$$

From the above we know that if a build is erroring, then there must exist at least one error. By pushing the negation into the formula, we can also know that if a build is passing, then there must not exist any errors. That is, the model of a passing commit must not contain any rules which are breaking. Note we are not, however, guaranteed that any rules from a passing commit are necessary.

$$S(P_t) = Err \implies \exists r \in M(P_t), r \in Br \tag{2}$$

$$S(P_t) = Pass \implies \forall r \in M(P_t), r \notin Br \tag{3}$$

While Eq. 2 and 3 might build a basic model, they will do not capture all of the available knowledge. The key insight is that when we commit a break (PE), we can localize the error to one of the lines that changed. Either we removed something that was necessary, or added something that was breaking. Note that this is an inclusive disjunction, since a erroring commit can break multiple things at once. Expressed formally, where $\setminus$ is the set difference, that is:

$$\begin{aligned} S(P_{t,t+1}) = PE \implies & \\ & \exists r \in (M(P_t) \setminus M(P_{t+1})), r \in Nec \\ \vee\ & \exists r \in (M(P_{t+1}) \setminus M(P_t)), r \in Br \end{aligned} \tag{4}$$

We then can combine all these formulas with conjunctions and ask a SMT solver for a model satifsying the formula. While existential set operations can be expensive on large sets for an SMT solver, in our application this is not the case. Thanks to the practice of making incremental commits when using source control, these sets will be small and the SMT will be fairly cheap. In fact, the above implication generalizes to $P_{t,t+n}$, but for efficiency we must require that $M(P_t) \setminus M(P_{t+n})$ is manageably small. The definition of small here remains to be experimentally determined.

IV. Limitations

Note we have made two assumptions in Eq. 1; first that the summary P_t sufficently detailed, i.e. contains every piece of information that might lead to a build error, and second that the model $M(P_t)$ will learn all relations that might lead to a build error, i.e. has a templete for all types of errors. While these are the strong assumptions - our algorithm is able to detect cases where these assumptions are not met. If we cannot find a solution for Eq. 4, it means either P_t or $M(P_t)$ has been underspecified. It is then the user's responsibility to expand the definitions accordingly.

References

[1] T. A. Lau, P. M. Domingos, and D. S. Weld, "Version space algebra and its application to programming by demonstration." in *ICML*, 2000, pp. 527–534.

[2] M. Santolucito, E. Zhai, and R. Piskac, "Probabilistic automated language learning for configuration files," in *CAV*, 2016, pp. 80–87.

[3] T. Gehr, S. Misailovic, and M. Vechev, "Psi: Exact symbolic inference for probabilistic programs," in *CAV*, 2016.

[4] P. Garg, C. Löding, P. Madhusudan, and D. Neider, "Ice: A robust framework for learning invariants," in *CAV*, 2014, pp. 69–87.

[5] T. M. Mitchell, “Generalization as search,” *Artificial Intelligence*, vol. 18, no. 2, pp. 203 – 226, 1982. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0004370282900406

[6] Z. A. Beller M, Gousios G, “Oops, my tests broke the build: An analysis of travis ci builds with github,” PREPRINT, 2016. [Online]. Available: https://doi.org/10.7287/peerj.preprints.1984v1