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

***Abstract*—Rule based classification is an effective machine learning technique that yields low misclassification rates. However, building a rule based system requires manual creation of large databases of logical constraints. We present a method to generate rule based systems from temporally structured data. As an demonstration of this algorithm, we plan to implement a learner that automatically generates constraints for the TravisCI testing framework. The algorithm will utilize Github commit histories to generate logical constraints that allow us to detect potential build errors without actually building, saving valuable programmer and server time.**

I. Introduction

Machine learning is a powerful tool for software analysis and verification [1], [2], [3]. Among the popular algorithms are neural nets and n-gram models, which produce to probabilistic models of correctness. While often effective in practice, these do not provide the guarantees of a traditional verification approach. In addition, these tools are not designed to provide simple justifications for their classification outputs.

In contrast, version space learning is a machine learning strategy for logical constraint classification [4], appropriate when the learning set does not contain noise. Extensions to version space learning have been used various software analysis techniques, such as programming-by-example [5], invariant synthesis [3], and error detection [1]. We first give a formal definition of the algorithm in [1] in terms of version space learning. We then extend that work to leverage temporal structure in a learning set, specifically by looking at the differences between sequential examples.

As a concrete use case, we plan to implement our algorithm to check for TravisCI configuration errors. TravisCI is a continuous integration tool that allows programmers to automatically run their test suite on every code update. 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

Version space learning builds a logical constraint model for binary classification, or membership in a set. ConfigC uses this approach to identify errors in configuration files for MySQL [1] . ConfigC learns a model over a set of configuration files that have been labeled as correct, then reports misconfigurations.

Traditional version space learning will use a series disjunctions of 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.

To build the model, ConfigC take a single file and derives all possible relations from each file, such as A comes before B, B before C, and A before C. The user must provide templates for possible relations. It is assumed if a file is correct, all relations in that file are necessary for another 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*. This model is then iteratively relaxed as more examples are seen, a process called *candidate elimination*. To relax the model, two sets of relations from two files are merged into a single consistent set.

```
n = {}
for (c in files):
  n1 = M(c)
  n = merge(n, n1)
```

III. Learning from Temporal properties

In ConfigC, the configuration files were analyzed as standalone documents. 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. In the case of TravisCI, this include the `.travis.yml` file, as well as extract key code features that may effect build status, such as programming language and a list of imported libraries. The summary must contain every piece of information that might lead to a build error.

The subscript on P_t is a time stamp tag based on the ordered commit history. However, a git history is not a limited to a

single linear timeline. Git features the ability to *branch*, which allows to simultaneous commit chains. To handle the start of a branch, add a superscript to indicate the branch, and restart the counter on a branch. To handle the merge of two branches P_t^x and $P_{t'}^y$, step to P_{t+1}^x, where x is the mainline branch. We then say that $P_{t'}^y$ has no successor commit $P_{t'+1}^y$.

We will denote the build status of P_t with $S(P_t)$. In this application, we consider only the passing and erroring build status, denoted $Pass$, and Err respectively. All status that are not `error`, as defined by the Travis API, will be included as passing. For brevity, we denote sequences of build statuses with the following notation:

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

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. 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 complete.

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

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 (P E), 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. We use 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 send it to an SMT solver. 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

A. Unsatisfiability

Note we have made two assumptions in Eq. 1; first that the summary P_t 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. While these are the strong assumptions - our algorithm is able to detect cases where the summary is incomplete. If we cannot find a solution for Eq. 3, it means either P_t or $M(P_t)$ has been underspecified. It is then the user's responsibility to expand the definitions accordingly.

B. Trusted Base

In addition to the completeness of P_t and $M(P_t)$, we must pick a trusted base. Since it is possible for TravisCI to have bugs, it is possible that between versions of TravisCI, two identical program summaries may have different build statuses. Since version space learning is (generally) intolerant of noise, we require that $P_t = P'_{t'} \implies S(P_t) = S(P'_{t'})$.

Acknowledgment

This work was funded in part by NSF grant 000000000

References

[1] M. Santolucito, E. Zhai, and R. Piskac, "Probabilistic automated language learning for configuration files," in *Computer Aided Verification: 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, 2016, pp. 80–87.

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

[3] P. Garg, C. Löding, P. Madhusudan, and D. Neider, "Ice: A robust framework for learning invariants," in *International Conference on Computer Aided Verification*. Springer, 2014, pp. 69–87.

[4] 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

[5] 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.

[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