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 has been used in various software analysis techniques, such as programming-by-example [1], invariant synthesis [2], and error detection [3]. 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 to software analysis, where justification is often critical. For example, in the case of error detection, the system should not only report which files have errors, but also locate the errors. Version space learning is a technique for logical constraint classification [4]. As a logical constraint system, version space learning can provide the justification that is difficult to obtain from other methods, but its main weakness is that it cannot handle noisy data.

In our previous work [3] we built a prototype to verify configuration files from a set of correct examples. The tool, ConfigC, takes as a training set a set of correct configuration files and builds a set of rules describing a language model. This model is then used to check if a new configuration file adhears to those rules. If a file is incorrect, the tool identifies the location of error, and reports what is incorrect. For example, ConfigC might output `Ordering Error: Expected "extension mysql.so" before "extension recode.so"`. While this prototype's results are promising, the main weakness are that the learning can only use correct configuration files. Additionally, there is a relatively high false positive rate (marking correct files as incorrect).

In order to extend the prototype, in this paper we provide a new description of ConfigC in terms of version space learning. This description will allow us to understand how the justifications are produced in ConfigC. We can then extend ConfigC with a new algorithm to decrease the false positive rate, while maintaining the ability to provide justification.

We propose an extension to ConfigC that allows it to handle both correct and incorrect files in the training set. Additionally, we extend ConfigC with the ability to use structure within the training set. While the previous training set from ConfigC was a random sampling of configuration files, many other training sets have a structure based on version history. This structure is simply a partial order, and can be used by our proposed algorithm for more effective learning.

To test this approach in practice, we plan to implement our algorithm to check for TravisCI[1] 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 [5] of TravisCI found that 15-20% of failed TravisCI builds are due to ”errors” - which is the TravisCI name used to mean the configuration file was malformed and the software could not even be built. Using the data from [5], 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.

Our main contributions are then as follow:

1) We give a description of our prototype tool, ConfigC, in the context of version space learning.
2) We propose a new algorithm for ConfigC to handle both incorrect and correct training data, as well as internal structure of a training set.
3) We propose a real-world application of this approach, and outline the steps needed for an implementation.

II. Version Space for Verification

Version space learning builds a logical constraint model for binary classification, which we use to test a configuration file for membership in the set of all correct files [4]. Traditional version space learning builds a model that tests membership using a series of disjunctions from a set of predefined hypotheses. In ConfigC, we instead restrict the model to use a series of conjunctions. In other words, rather than describing a correct file by allowable traits, we describe the required traits.

[1]http://www.travis-ci.com

This way, ConfigC can not only flag misconfigurations, but also give the points of failure for non-membership. We now show in detail how this process works.

Our definition of a configuration file, C, is a file that can be transformed to an ordered list of $(keyword, value)$ pairs, called $Line$s. ConfigC builds a model for a single file, $M(C)$, that is the set of all possible relations between the lines in the file. A relation is described by user provided templates. A template is a function taking some number of lines and determining a relation on the keywords of the lines ($Line^n \rightarrow Rel_{k1,..,kn}$). As an example, ConfigC might derive the relation `extension mysql.so` comes before `extension recode.so` by using the ordering template. ConfigC features the following default templates.

Template	Input	Relation
ordering	(Line,Line)	Before | After | None
integer relation	(Line,Line)	$<$ | $==$ | $>$
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, Nec, 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*. 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)
```

Since we maintain the strongest condition for correctness, this approach will identify many correct files as incorrect, which we call a high false positive rate. This is a problem when a user is then asked to manually review many files for errors, when the files are in fact correct.

III. Learning from Temporal properties

ConfigC can provide justifications, 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 a new algorithm to decrease the false positive rate in ConfigC by learning on both correct and incorrect examples, as well as temporal structure of these examples. This extension will build 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 *sufficiently detailed*, that is it must contain every piece of information that might lead to a build error.

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^x_t and $P^y_{t'}$, step to P^x_{t+1}, where x is the mainline branch. We then say that $P^y_{t'}$ has no successor commit $P^y_{t'+1}$. Note the ability to handle branching implies this method is suitable to any learning set which has a partial order.

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.

We will denote the build status returned from TravisCI when run on P_t with $S(P_t)$. Any files with a $Pass$ build status are correct, and any files with an Err build status are incorrect. 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 incorrect and correct 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 sufficiently 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. 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.

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

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

Additionally, when we commit a break (PE), we can localize the error to one of the relations 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 Eq 2, 3, and 4 with conjunctions and ask a SMT solver for a model satifying the formula. The resulting model will be the sets Nec, and Br, which can be used to check new configuration files. Since we used a similar model to ConfigC, we will still be able to provide justifications for the classification results.

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.

A. 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 $P_t = P_{t'} \implies S(P_t) = S(P_{t'})$.

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

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

[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] 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