Version Spaces learning for verification from temporal differentials

Plan of Attack

Mark Santolucito
Yale University
mark.santolucito@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.

1. Introduction

Machine learning is a new direction in verification[3**?**]. However, algorithms such as neural nets and n-gram models lead to probabilistic models of correctness. While often effective in practice, these do not provide the true guarantees of a traditional verification approach. In addition, these tools often cannot provide a justification for the output. Neural nets behave surprisingly similarly to human brains, and providing proofs to conclusions is equally challenging.

Version space learning [2].

2. Availability of Data

TravisCI has recently released a metadata api to study their tool[1].

The first use of this API was analyzing the metadata of builds. It was found that ˜15-20% of failed TravisCI builds are due to "errors". My understanding of the API is that this corresponds to a misconfiguration. This is a large enough number that not only will reducing this be more convenient for users, but it could also significantly reduce server costs for Travis. By using the `tr_duration` field from the API, we can also figure out how much server time we can save[1].

CONF 'yy, Month d–d, 20yy, City, ST, Country.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

3. Feasability Anaysis

The language of TravisCI configuration files can likely be handled by ConfigC. ConfigC works on languages with shallow parse trees, and most of the .travis.yml files is shallow. However, because .travis.yml allows for bash scripts, some error may be out of scope. We want to identify the frequency of the types of errors - if too many are related to poorly formed bash scripts, a nave application of ConfigC will not suffice.

To confirm that the types of most misconfigurations are in the scope of ConfigC, we should compile a collection of common misconfigurations. Usually such a task requires a domain expert. However, in this case we may be able to learn the common errors by using the API and the temporal data from the commits.

We might code an error as the diff between a sequential pair of an erroring commit and a passing commit. As a new direction, we may also explore how useful this data is for learning/verification directly. My guess is that is data will be too noisy, but either way, it should give us a sense of the problem.

To build such a database, scrape the .travis.yml files from each commit for a set of repos. This can be achieved with the `tr_status, git_commit and gh_project_name` fields from the API.

3.1 Learning from Temporal properties

An interesting extension to ConfigC on a theoretical side is to look at the temporal relations between commits. The key observation is that with each commit that changes the build status, we can learn highly localized information about our model.

The first step is to build an intermediate representation of the data we will learn. This data must be structured as a shallow tree for generalizable learning. This restriction is why this approach is not appropriate for language learning on large grammars (such as a programming language).

While the travis.yml file has a simple structure and can be taken verbatim, we must use a simplification of the program code. Since we are only interested in the parts of the program effecting build status, we should extract key feautres such as programming language and a list of imported libraries. We will call this program summary P_t, a simplified representation of the repository which contains only the information relevant to the learning process. The summary must contains every piece of information that might lead to a build error.

The subscript on P_t is a 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 simultanious 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 $P^y_{t'}$ has no successor commit $P^y_{t'+1}$.

From this summary we can then build a model $M(P_t)$, which is the full set of possible rules derivable from the model. These possible rules must be given by the user. In [3], these were called Attributes, and express constraints such as, ordering of lines, or substring relations on values.

We will denote the build status of P_t with $B(P_t)$. 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:

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

With this notation, we can formally express the requirement that the program summary is complete. This is an assumption on the quality of the program summary.

$$\forall S(P_t) = Err, \exists r \in M(P_t), r \in Br$$

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, garunteed that any rules from a passing commit are necessary.

$$S(P_t) = Err \implies \exists r \in M(P_t), r \in Br \quad (1)$$
$$S(P_t) = Pass \implies \forall r \in M(P_t), r \notin Br \quad (2)$$

While Eq. 1 and 2 might build a basic model, they will do not caputre all of the avaible 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. 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:

$$S(P_{t,t+1}) = PE \implies$$
$$\exists r \in (M(P_t) \setminus M(P_{t+1})), r \in Nec \vee$$
$$\exists r \in (M(P_{t+1}) \setminus M(P_t)), r \in Br \quad (3)$$

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.

3.2 Unsatisfiability

Note we have made a strong assumption that the summary P_t contains every piece of information that might lead to a build error. This is the strongest assumption that we make - however our algorithm is able to detect cases where the summary is incomplete. If we cannot find a solution for Eq. 3 it means that the set of rules we are looking for is incomplete.

4. Implementation

To build the learning set, we will use the API to build tuples of .travis.yml and repo info. In addition to the .travis.yml file (collected as above), we will need at a minimum `gh_lang and tr_status`. With this `(File,gh_lang,tr_status,...)` tuple, we can then (almost) directly apply ConfigC to detect errors in the shallow part of the tree.

The first (and simplest) requirement is a .travis.yml parser, which is an extension of normal yml parsing to handle travis' ability to include bash scripts. Second, depending on the results from Sec 3, we may also require some extra domain knowledge of the types of errors to be encoded as possible rules. This is a slightly more involved task that requires writing some non-trivial Haskell code - but it really isn't that bad.

4.1 Limitations

In addition to the completness of the program summary P_t, and the completness of the relations that can be learned $M(P_t)$, there are challenges in a practical implementation. The most difficult is that the configuration language also have multiple versions. We will need to fix a version of TravisCI itself, and ensure we only learn on that version. Additionally, this algorithm makes the assumption that TravisCI is bug free.

References

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

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

[3] M. Santolucito, E. Zhai, and R. Piskac. *Probabilistic Automated Language Learning for Configuration Files*, pages 80–87. Springer International Publishing, Cham, 2016. ISBN 978-3-319-41540-6. .