Establishing Benchmarks For Learning Program Representations

Anjan Karmakar
Faculty of Computer Science
Free University Bozen-Bolzano
akarmakar@unibz.it

Abstract

Recent advances in the field of machine learning have shown great promise in solving various software engineering tasks. However, unlike machine learning techniques used in fields such as NLP (Natural Language Processing) where text-based tokens are used as model inputs, in software engineering (SE) structured representations of source code have proven to be more effective for various SE tasks. Despite the findings, structured representations of source code are still underused. In this paper, we propose to define a benchmark that promotes the usage of structured representations of source code as model inputs, via tasks that are explicitly defined towards that goal.

1 Introduction

With the advent of big code, applying machine learning techniques on large corpora of code have yielded excellent results for a number of software engineering tasks. Particularly, neural networks have been very effective since they are able to learn the features from the input. However most of the tasks so far use program structure in a shallow manner (name prediction from snippets, source code summarization, finding mappings between APIs, source code search, etc). More recently a number of papers have utilized the structured nature of source code and accomplished state of the art results for certain software engineering tasks.

Like natural language, source code also is structured and repetitive [5], therefore representing instances of the input source code effectively, while leveraging their semantic and syntactic properties, could facilitate better learning of machine learning models. Studies such as [3] [1] used source code representations in the form of ASTs, and graphs, to essentially capture the semantic and syntactic properties of source code and then use them to train their model.

Although there are a number of ways to represent source code, such as simple token-based representations, ASTs (Abstract Syntax Trees), call graphs, bytecode, we are interested in the more structured representations of code. Furthermore, we hypothesize that the structured representation of source code as inputs to machine learning models would perform better for a variety of software engineering tasks, including tasks such as code completion and defect prediction.

Therefore, to evaluate our hypothesis, we aim to propose a benchmark made up of tasks designed in such a way that, to succeed, it is necessary to learn more and more about the program structure. The tasks shall be of increasing difficulty (i.e., learning more and more of the structure is necessary to yield desirable results). We expect that on the hardest ones, current neural approaches will fare little better than random chance. The tasks will be defined based on a set of static analyses of the source code.

2 Background and Motivation

Recently, there has been an increasing interest in applying machine learning techniques to solve SE (Software Engineering) tasks. However, most of the work has directly tried to reuse natural language processing (NLP) methods for SE tasks, mainly by treating source code as a sequence of tokens - which ultimately

In: Anne Etien (eds.): Proceedings of the 12th Seminar on Advanced Techniques Tools for Software Evolution, Bolzano, Italy, July 8-10 2019, published at http://ceur-ws.org

fail to capitalize on the unique opportunities offered by code's known structure and semantics [1].

Even though there are many similarities between natural language and source code, one interesting aspect that differentiates source code from natural language is that source code is highly structured, which can be leveraged to obtain a greater understanding of the context of the code. Owing to this structured nature of code, program elements are usually dependent on each other, and therefore when building models to predict program properties, simple sequence-based models, which treat these program elements to be independent aspects, fail to make use of the *inter-dependence* on other code elements [7]. Also, simple sequence-based models fail to determine which variables are in scope at any point in the program, which can be resolved simply by embracing more structured representations of code [6]. Some recent studies which have utilized this structured representation of source code have accomplished state of the art results on many SE tasks.

For example, Alon et al. [3] introduce the use of different path-based abstractions of the program's abstract syntax tree (AST) to represent source code. The goal is to extract a representation that captures relevant and interesting properties from the program's AST, while keeping the representation open for generalization. One such way to produce such a representation of source code is to decompose the AST into paths between nodes that repeat across programs but can also discriminate between different programs.

The authors show that representing source code as AST paths can be useful in a diverse set of programming tasks such as predicting variable names, predicting method names, and predicting types of variables. Furthermore, they claim that the use of AST paths can significantly improve the performance of the various learning algorithms without modifying them, while achieving state of the art results.

In yet another recent work by Zhang et al. [11], the authors address the challenge of source code representation, to effectively capture syntactic and semantic information, with an AST-based Neural Network (ASTNN) to learn vector representations of source code. The model decomposes large ASTs of code fragments into sequences of smaller statement trees, and then obtains statement vectors by recursively encoding multi-way statement trees. Based on the sequence of statement vectors, a bidirectional RNN model learns the vector representations of code fragments by leveraging the naturalness of statements, which is then evaluated on two tasks, namely source code classification and code clone detection, producing state-of-the-art results.

Allamanis et al. [1] propose a new method to represent code to capture the syntactic and semantic structure of code using graphs, and then use graph-based deep learning methods to learn to reason over program structures. The authors propose a new representation technique by encoding source code as graphs, in which edges represent syntactic relationships as well as semantic relationships. They observe that exposing source code semantics explicitly as structured input to a machine learning model reduces the requirements on the amounts of training data and model capacity - making way for solving tasks, such as variable naming (VarNaming), and detecting instances misused variables (VarMisuse), and achieving state of the art results.

From the studies above, it is clear that structured representations of code fare much better than simple text-based token representations, for the tasks investigated above. For our study, we are going to specifically focus the defect prediction tasks and attempt to use structured representations of code as input to a learning model to predict bugs. To evaluate the effectiveness of utilizing the structured representations of code, the task of bug prediction is particularly potent since there is a wide range of available bug types - some are easily detected while others require a thorough understanding of the syntactic formulation, semantics, and structure of the code.

3 Benchmarking

Our intention is to build a neural network model that is able to highlight buggy code in a given code corpus. In order to accomplish the said goal our methodology would require an amalgamation of different techniques that have already been proposed in the literature for diverse tasks. For the specific case of bug detection task, however, the techniques need to be applied together and evaluated in the right manner, since all our research questions are subject to experimental evaluation, preferably against established benchmarks.

An established benchmark, when embraced by a community, can have a strong positive effect on the scientific maturity of a community [9]. Benchmarking can result in a more rigorous examination of research contributions and pave the way for the rapid development and evaluation of new methods and tools.

While looking for established benchmarks in the field, we have discovered datasets like the publicly available PROMISE dataset [8], which also include contributions from the NASA Metrics Data Program [4], where a lot of defect prediction datasets are available. However, the tests conducted on these datasets are mostly metric-based, essentially using static measures to guide software quality predictions, and are essentially too "coarse" meaning they often highlight

bugs on the file level based on the complexity measures such as essential complexity, cyclomatic complexity, design complexity and lines of Code. On the other hand, we would like to have tests which are more "fine-grained", in the sense that, they could not only identify suspicious or buggy files but also highlight the exact bug locations in the lines of code.

Thus, as a preliminary goal, we need to first determine some benchmarks that define tasks that would require a thorough understanding of the programs structure and semantics. We then need to evaluate the performance of our trained model on these tasks and compare it against the tasks comprising our benchmark.

An additional desirable aspect of a benchmark is to control the difficulty of the tasks. An example in NLP is the work of Weston et al. [10], which defines 20 NLP tasks in increasing levels of complexity. The tasks are artificial, but establish minimum levels of complexity that a prediction model must fulfill. While any model that can solve these tasks is not necessarily close to full reasoning, however, if a model fails on any of these tasks then there are likely to fail in real-world tasks too.

In the case of bug prediction, we would like a similar property, but with more realism. These could range from basic tasks for which the information is directly accessible without needing to understand the program structure - where sequential processing is still viable, to more advanced tasks where understanding of non-local and indirect structure is necessary.

To sum up, unlike existing benchmarks we need tasks where the output is fine-grained, and where we can control the complexity of the tasks, so that we can provide incentives for models that leverage the program structure. One candidate that fulfills these conditions is to leverage static analysis tools. Static analysis tools leverage program structure and find real bugs. The static analyses can point out precise program locations, and the analyses vary in complexity, from simple and local pattern matching to full-program analyses.

Therefore, we could consider proceeding in defining certain tasks which would form the benchmark to evaluate machine learning models using the structured nature of code. The range of tasks based on difficulty could be categorized as:

1. **"Easy" tasks:** tasks for which the information is directly accessible without needing structure. E.g. a property of a method that can be deduced by information in the method body.

2. **"Medium" tasks:** tasks for which some degree of structure is necessary. E.g. a property of a method or variable that can be deduced, but you need to take the entire file context into account, not just the method body.

3. **"Hard" tasks:** tasks for which non-local structure is necessary. E.g. a property of a method that needs information from its direct callers.

4. **"Very hard"** tasks: tasks for which non-local and also indirect structure is necessary. E.g. you have to look into the callers of the callers.

5. **"Inaccessible" tasks:** tasks for which information very distant from the source is necessary. E.g., one statement in one method that is indirectly called by the method, but they are 5-10 steps away.

For example, the VarMisuse task defined by Allamanis et al. [1] serves as a good sample task for bug detection. The missing variables in the VarMisuse task must be predicted properly else they risk causing system failure, and thereby, when applied to our case our model could highlight or detect whether a variable has been misused and whether the module is buggy.

In essence, some variables are omitted from a given code snippet and fed as input to Allamanis et al's model. The learned model from Allamanis et al. then accurately predicts the expected variables with a high accuracy for all the variables but the last. Such a task could be categorized as a Medium task, since to accurately predict the omitted variable the model needs to take the code from the entire file into consideration, rather than just the local method.

To successfully tackle the task of bug detection, one needs to understand the role and function of the program elements and understand how they relate to each other. In the VarMisuse task discussed above, the model learns to predict the correct variable that should be used at a given program location. Since the model produces near accurate results, any variable in the code that does not match the expected variable predicted by the model could then be a point of inspection. This task could be complementary to our bug detection task. Given a certain code snippet, we must therefore ascertain whether it contains certain buggy fragments and then we can compare our results with the predictions made by the model proposed by Allamanis et al. [1].

These tasks from the established benchmark will also help us compare our results against those of static bug finders, which use approaches ranging from simple code pattern-matching techniques to static analyses that process semantic abstractions of code. From the list of tasks defined in our benchmarks, simple static bug finders would likely be able to detect bugs "easy" and "medium" tasks but fare poorly for "hard" tasks

where a greater understanding of the program structure is required.

Making a direct comparison with static bug finders against our model could reveal the effectiveness and/or weakness of our model. We could match the highlighted faulty lines of code or defect locations from the static bug finders and our model, and evaluate them for false positives and false negatives.

There are a number of Static Bug Finders (SBF) we could compare our results with to conclude on the effectiveness of our model. For example, Hybrid tools like FindBugs which incorporate both static data-flow analysis, and pattern matching, could be a good evaluation candidate. Also, tools like ESC-Java and CodeSonar could be considered, since their analyses are reportedly more effective and they could possibly highlight bugs in "hard" tasks.

Advanced tools like Infer could even understand the program structure and fare better than traditional bug detection tools, and it will be interesting to compare their results against the set of tasks from the established benchmarks, and whether they are able to detect bugs in "hard" and "very hard" tasks.

4 Discussion

The eventual goal of our research is to enable learned models to effectively detect bugs from a new corpus - which is not an easy task. Detecting code faults is a task that requires through understanding of the code and reasoning about the program semantics. Even for an experienced programmer this is a challenging task.

To allow a learning model to grasp the correlations between code fragments and understand how they work, we need a better way to map code fragments instead of simple sequential token-based mapping. Even though code fragments have something in common with plain texts, they should not be simply dealt with text-based or token-based methods due to their richer and more explicit structural information.

Recent work [2] [1] provides strong evidence that semantic and syntactic knowledge from source code as structured representations contributes a great deal in modeling source code and can obtain better results than traditional sequential token-based methods. Because of the highly structured nature of code, treating code snippets as structured representations for the machine learning models, can capture critical semantic information that reflects common code patterns, and even significantly lessens the requirements on the amount of training data compared to learning over sequential token-based representations [2], and it is still general enough so that it can be applied to a diverse range of tasks.

However, to measure the effectiveness of the models based on various representations of code, and to compare against the results from static analysis, we need a set of tasks as a benchmark. Therefore, in our preliminary work we attempt to define a benchmark measuring the accuracy and usefulness of a model based on certain type of representation. The tasks comprising the benchmark would essentially test the understanding of the model, starting from simple tests for which the information is directly accessible without needing to understand the program structure, to increasingly difficult tests where understanding of non-local and indirect structure is necessary. Furthermore, this benchmark would also serve new models with novel representation techniques for their evaluation purposes - to compare against current state of the art.

Once our benchmarks are established, as a part of our future work, we could begin our bug detection approach by first mining programs in our training set as Abstract Syntax Trees (ASTs). Then selecting the most effective paths in the ASTs, we could derive code embeddings from the program as an input for our learning model. And finally, train the learning model with pairs of buggy and correct code, based on the vectors representations of the code fragments. When the model is trained it should be able to highlight buggy fragments of code from the input - an unseen code corpus. Another approach could be to mine the programs as ASTs and represent them as graphs which could be used as a direct input to feed Graph Neural Networks. Establishing a solid benchmark for the defect prediction task, would boost the development and evaluation of new techniques and tools in defect prediction, and allow for the comparison of their effectiveness. Our aim with this paper is to introduce the notion of benchmarking for learner models based on structured representations of code, and propose a set of tasks categories that would evaluate the usage of program structure specifically for the task of defect prediction.

References

[1] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. *CoRR*, abs/1711.00740, 2017.

[2] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *CoRR*, abs/1803.09473, 2018.

[3] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. A general path-based representation for predicting program properties. *CoRR*, abs/1803.09544, 2018.

[4] Jackson W. Chapman SM., Callis P. Metrics data program. *NASA IV and V Facility*, 2004.

[5] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 837–847, Piscataway, NJ, USA, 2012. IEEE Press.

[6] Chris J. Maddison and Daniel Tarlow. Structured generative models of natural source code. *CoRR*, abs/1401.0514, 2014.

[7] Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting program properties from "big code". In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 111–124, New York, NY, USA, 2015. ACM.

[8] J Sayyad Shirabad and Tim J Menzies. The promise repository of software engineering databases. *School of Information Technology and Engineering, University of Ottawa, Canada*, 24, 2005.

[9] S. E. Sim, S. Easterbrook, and R. C. Holt. Using benchmarking to advance research: a challenge to software engineering. In *Proceedings of the 25th International Conference on Software Engineering, ICSE '03*, pages 74–83, May 2003.

[10] Jason Weston, Antoine Bordes, Sumit Chopra, Alexander M Rush, Bart van Merriënboer, Armand Joulin, and Tomas Mikolov. Towards ai-complete question answering: A set of prerequisite toy tasks. *arXiv preprint arXiv:1502.05698*, 2015.

[11] Zhang H Sun H Wang K Liu X Zhang J., Wang X. A novel neural source code representation based on abstract syntax tree. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, May 2019.