Unit test generation using machine learning - Work in Progress -

Laurence Saes
University of Amsterdam
l.saes@live.nl

Joop Snijder
Info Support B.V.
Joop.Snijder@infosupport.com

Ana Oprescu
University of Amsterdam
a.m.oprescu@uva.nl

Abstract

Test suite generators could help software engineers ensure software quality by detecting software faults. This even applies to projects that do not have a test suite and could be done by generating an initial test suite that is in the future maintained and optimized by the developers. However, state-of-the-art test generators are still only able to capture a small portion of potential software faults. The Search-Based Software Testing 2017 workshop compared four unit test generation tools. These generators were capable of achieving an average mutation coverage below 51%. These methods are unable to detect all mutants that are covered by unit tests written by software engineers.

We propose a test suite generator that uses neural networks to detect mutants that could only be detected by manually written unit tests. Multiple networks, trained on open-source projects, are evaluated on the effectiveness of the generated test suite. The data set contains the unit tests and the code that it tests. The unit test method names are used to link unit tests to methods under test.

Preliminary results are promising: our linking mechanism could link 27.41% (36,301 out of 132,449) tests.

Proceedings of the Seminar Series on Advanced Techniques and Tools for Software Evolution SATToSE 2018 (sattose.org). 04-06 July 2018, Athens, Greece.

1 Introduction

Test suites are used to ensure software quality when a program's code base evolves. The effectiveness of a test suite is often measured as the ability to uncover faults in a program [ZM15]. The test suite effectiveness correlates with its mutation score [ZM15; Jus+14]. The mutation score of a test suite represents the test suite's ability to detect syntactic variations of the original source code, i.e., mutants, and is computed using a mutation testing framework.

Although intensively researched [Alm+17; KC17; C+08; FZ12; Rob+11], state-of-the-art test suite generators lack coverage that could be achieved with manual testing. Almasi et al. [Alm+17] explained a category of faults that are not detectable by these test suite generators. These faults are usually surrounded by complicated conditions and statements for which complex objects have to be constructed and populated with specific values.

Additionally, the 2017 edition of the Search-Based Software Testing (SBST) workshop[1] held a competition for Java unit test generators. Test suite effectiveness of generated test suites and manually written test suites was evaluated. The manually written test suites scored on average 53.8% mutation coverage, while the highest score obtained by a generated test suite was 50.8% [Fra+17]. *However, the set of mutants detected by the generated tests is not a subset of the mutants detected by the manually written tests* [FZ12].

To leverage the ability to automatically test many different execution paths and the ability to learn how to test complex situations of generated and manually written test suites, we propose a test suite generator that uses machine learning techniques. Our solution uses neural networks and combines manual and automated test suites by learning patterns between tests and code in order to generate test suites with a higher effectiveness.

[1] http://sbst2017.lafhis.dc.uba.ar/

1.1 Research questions

Our **research goal** is to study machine learning approaches to generating test suites with high effectiveness: learn how code and tests are linked and apply this logic on projects' code base. Although neural networks are widely used for translation problems [SVL14], training them is likely time-consuming, thus we also research heuristics to alleviate this issue.

RQ1 What neural network solutions could be applied to generate test suites for software projects in order to achieve a higher test suite effectiveness?

RQ2 What is the impact on the training time of eliminating partial predictions which are not compliant with the language's grammar?

RQ3 What is the impact of input and output sequence compression on the training time and accuracy?

2 Background

Machine learning is often applied to source code. In our case, source code (method body) is translated to source code (unit test). Beltramelli et al. [Bel17] used machine learning to translate a mock-up into HTML. They target domain-specific languages (DSLs) with a limited vocabulary in order to reduce the complexity. Ling et al. [Lin+16] translated game cards into code. For this, they developed a machine learning algorithm that is specialized in predicting tokens while the tokens themselves are not very common in the training data. Parr et al. [PV16] developed a universal code formatter that learns the code style from a code base. They used a k-Nearest Neighbor machine learning model. Karaivanov et al. [KRV14] used phrase tables in order to translate one programming language into another. Devlin et al. [Dev+17] developed a solution that repairs syntactic errors and semantic bugs in a code base. Zheng et al. [Zhe+17] developed a solution that translates code to comments. Yin et al. [YN17] developed an algorithm that translates Natural Language into code. They use an abstract syntax tree (AST) in order to capture the strong underlying syntax of the programming language.

3 Related work

Multiple approaches address the software challenge of achieving a high test suite effectiveness. Tests could be generated based on the project's source code by analyzing all possible execution paths. An alternative is using test oracles, which can be trained to distinguish between incorrect and correct method output.

Common methods for code-based test generation are random testing [Alm+17], search-based testing [Fra+17; Alm+17], and symbolic testing [C+08]. Almasi et al. benchmarked random testing and search-based testing on the closed source project LifeCalc [Alm+17] and found that search-based testing had at most 56.40% effectiveness, while random testing achieved at most 38%. Symbolic testing was not evaluated because there was no symbolic testing tool available that supports the analyzed project's language.

Cadar et al. [C+08] applied symbolic testing on the HiStar kernel achieving 76.4% test suite effectiveness compared to 48.0% with random testing. Fraser et al. [FZ12] analyzed an oracle generator that generated assertions based upon mutation score. For Joda-time, the oracle generator covered 82.95% of the mutants compared to 74.26% for the manual test suite. For Commons-Math, the oracle generator covered 58.61% of the mutants compared to 41.25% for the manual test suite. The test oracle generator employs machine learning in order to create the test oracles. Each test oracle captures method behavior for a single method in the software program by training on the method with random input. During testing, the method is given random data and its output is validated by the earlier generated test oracle. Compared to this approach, our proposed method generates code while this method guesses the output of methods.

3.1 Machine learning

There are multiple neural networks solutions that could transform a sequence to another sequence. A possible solution is a sequence-to-sequence neural networks that can be based on recurrent neural networks (RNNs) or convolutional neural networks (CNNs). The version that uses RNNs could contain long short-term memory (LSTM) nodes [SVL14; SSN12] and can be configured so that it can make predictions on large sequences [BCB14] and on out of vocabulary (OOV) words [Gu+16; Dev+17]. CNNs could be used as an alternative. Recent research shows that CNNs can also be applied to make predictions based on source code [APS16]. They also require less training data and are faster to train [Zhe+17]. To answer **RQ**1, we evaluate both CNNs and RNNs as both tools look promising.

4 A Machine Learning-based Test Suite Generator

Our solution focuses on generating test suites for projects that have no test suite at all. Our solution requires all the project's method bodies, with the name of the method, and the class that contains the method. The test generator sends the method bodies in a textual representation to the neural network. The neural network generates test method bodies. The test generator places the generated methods in test classes.

These could be used in the future to test the project's source code on faults.

In an ideal situation, a model is already trained. When this is not the case, then additional actions are required. The neural network has to be trained with training examples. A selection of training projects should be made. All training projects should use the same unit test framework and can be built and tested. A unit test linking algorithm is used to extract training examples from these projects. The found methods and the unit test method are given as training examples to the neural network. This leads to the model that the neural network uses to generate the test method bodies.

4.1 Linking code to test

The linking application links unit tests with the method that is under test. In general, every test class is developed to test a single class. The interface of the unit test class is used to determine what class and methods are under test. We consider that every class that is in the unit test class could be under test. For every class, we determine what methods based on their name match the best with the interface of the unit test class. The class with the most matches is assumed to be the class under test. The methods are linked with the unit test methods that have the best match. A match is only made when the name of the test contains the method name. The best match is the match that is the longest. This also means that a unit test method cannot be linked when it does not have a match with the class under test.

However, this algorithm has limitations. For example, in Code 1 stack and messages are both considered the class under test. It is possible to detect that stack is under test when the linking algorithm only is applied on the statements that are required to perform the assertion. This because statements that are not used to compute if the assert succeeded are eliminated. Backward slicing could be used to generate this subset of statements because the algorithm can extract what statements have to be executed to perform the targeted statement [Bar+10]. This would work in this case because this subset will only contain calls to stack. However, this algorithm will not work when asserts are also used to check if the test state is valid to perform the operation that is tested.

Code 1: Unit tests that will be incorrectly linked without statement elimination

```
public void push() {
    ...
    stack = stack.push(133);
    messages.push("Asserting");
    assertEquals(133, stack.top());
}
```

5 Evaluation Setup

Our approach is evaluated using a metric that enables us to compare our result with alternative test generators and manually written test suites. We need a tool that can perform the metric and a set of projects that can be used as a benchmark. Multiple projects were selected that have a working test suite and for which the metric can be calculated. We also made sure that the projects are different in size and metric performance.

5.1 Metric

Test suites are compared based on their mutation score. The measurement is done with a fork of the PIT Mutation Testing[2] because they combined multiple mutation generations. A comparison is based on the test suite's ability to detect mutants.

5.2 Baseline

The effectiveness of a project's manually written test suite and that of automatically generated test suites are used as the baseline. Only test suit generators that implement search-based testing and random testing are considered because many open-source tools are available for these methods and they are often used in related work. We use Evosuite[3] for search-based testing and Randoop[4] for random testing, as these are the highest scoring open-source test suite generators in the 2017 SBST Java Unit Testing Tool Competition in their respective categories [PM17].

We evaluate our approach on six projects that were chosen because they are supported by the chosen state-of-the-art mutation testing tool (see sec. 5.1) and they have a varying mutation coverage. The varying coverage is needed to better measure the impact of our tool. Table 1 lists the size and coverage of each project.

[2]https://github.com/pacbeckh/pitest
[3]http://www.evosuite.org/
[4]https://randoop.github.io/randoop/

Table 1: Mutation coverage by project

Project	Java files	Mutation coverage
Apache Commons Math 3.6.1	1617	79%
GSON 2.8.1	193	77%
La4j 0.6.0	117	68%
Commons-imaging 1.0	448	53%
JFreeChart 1.5.0	990	34%
Bcel 6.0	484	29%.

5.3 Selection of training data

The proposed solution using neural networks requires training data. We analyzed 3,385 open-source projects and eliminated all projects that could not be built or tested. This resulted in 1,106 usable projects for which we in total could extract 560,413 unit tests. These unit tests could be used to create training examples. However, the total amount of training data could be less than the number of unit tests because the linking algorithm might be unable to link every unit test (as described in Section 4.1).

6 Preliminary Results

The first result where we can report on is the algorithm which can link unit tests with methods. More results will be added in the final version of the thesis.

6.1 Linking code to unit tests

As described in section 4.1, a unit test can be linked with the method under test by looking at naming in the test method. This means that unit tests with clear names will disambiguate the link between the unit tests and the methods under test. Therefore, self-documenting code not only aids humans, but it could also aid machine learning algorithms.

6.2 Effectiveness of linked tests

To our knowledge, no method that is able to pair a unit tests to the method under test so far. We developed two tools that use our algorithm to link code to tests. We developed an AST and Bytecode analysis tool. Bytecode analysis in comparison with AST analysis has the advantage that it can determine concrete classes in a given context but bytecode analysis is harder to perform because frameworks that interpret the bytecode often fail.

The projects in our dataset contain in total 560,413 unit tests for which 175,943 are left when removing duplicates. However, many of these projects were copies. In our analysis, we remove duplicate tests based on their package name, class name, and test name. Unfortunately, this will not eliminate duplicates when the name of the unit test, class or package is changed, and it will remove none duplicates when they have by coincidence the same package, class and method name. For example, tests could be removed when two different versions of the same project are analyzed.

Bytecode analysis was supported in 74.01% (132,675) of the cases, AST analysis in 98.02% (175,717), and they have an overlap of 73.89% (132,449). Using both an AST and bytecode analysis would give a coverage of 98.15% (175,943).

To compare both methods, we made a comparison on the 132,449 registrations that are supported by both tools. Both tools combined could link 27.41% (36,301) of the tests. The AST approach could link 21.89% (28,987) and with bytecode analysis 26.10% (34,572). Both tools could both create a link to 20.58% (27,258) of the registrations.

7 Conclusion and next steps

In section 6.2 we mentioned the number of links we could make on a set of projects. However, it is possible that for some tests, an incorrect class under test is considered. This will result in false positives matches. Many of them could be eliminated with backward slicing as described in section 4.1. We expect that we will eliminate false positives when we link based on the statements outputted by this method.

A risk could be introduced when our method is replacing humans for writing tests. Our generator could be unable to cover all mutants that would have been captured by a human. This will result in test suites with less effectiveness.

In our research, we only considered a selection of open-source projects. The result could be different when other open or closed source projects were used.

We used an alternative version of PIT Mutation Testing. During our project, this tool achieved the most reliable result to measure test suite effectiveness. However, new versions of the original tool were released and could outperform this version. The performance of the original tool should be checked in future research and used when it has better performance. Our research is not invalidated when the tool performs better. The result would only be more accurate when comparing to real faults.

Our proposed solution could also be used to aid software engineers in writing tests. It could be used to generate initial unit tests for newly created methods. This could reduce the time needed to write unit tests, leaving more time for optimizing them.

The next step for this research is to train the neural networks. The result of the best solution has to be compared with state-of-the-art test suite generators and the project's initial test suite in order to compare

our approach with the results from other researchers. Both **RQ**2 and **RQ**3 should be implemented in order to benchmark if there is a performance improvement.

The generated tests should be analyzed in order to help software engineers and to improve the test generator. We should investigate what test cases were missed by our solution, what tests are missed by others, what type of mutations are/are not covered by our solution compared to human-written tests, and the maintainability of the generated tests. Additional test guidelines could be made for software engineers to improve their testing quality, and our solution can be trained with training examples of mutants that it misses.

References

[Alm+17] M. M. Almasi et al. In: *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*. 2017.

[APS16] M. Allamanis, H. Peng, and C. Sutton. In: *International Conference on Machine Learning*. 2016.

[Bar+10] J. B. Barros et al. In: *Proceedings of the 2010 8th IEEE International Conference on Software Engineering and Formal Methods*. SEFM '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 93–102. ISBN: 978-0-7695-4153-2. DOI: 10.1109/SEFM.2010.18. URL: http://dx.doi.org/10.1109/SEFM.2010.18.

[BCB14] D. Bahdanau, K. Cho, and Y. Bengio. In: *CoRR* abs/1409.0473 (2014). arXiv: 1409.0473. URL: http://arxiv.org/abs/1409.0473.

[Bel17] T. Beltramelli. In: *CoRR* abs/1705.07962 (2017). arXiv: 1705.07962. URL: http://arxiv.org/abs/1705.07962.

[C+08] C. Cadar, D. Dunbar, D. R. Engler, et al. In: 2008.

[Dev+17] J. Devlin et al. In: *CoRR* abs/1710.11054 (2017). arXiv: 1710.11054.

[Fra+17] G. Fraser et al. In: *Proceedings of the 10th International Workshop on Search-Based Software Testing*. 2017.

[FZ12] G. Fraser and A. Zeller. In: *IEEE Transactions on Software Engineering* 38.2 (2012).

[Gu+16] J. Gu et al. In: 1 (2016).

[Jus+14] R. Just et al. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 2014.

[KC17] T. Kapus and C. Cadar. In: *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. 2017.

[KRV14] S. Karaivanov, V. Raychev, and M. Vechev. In: *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. 2014.

[Lin+16] W. Ling et al. In: *CoRR* abs/1603.06744 (2016). arXiv: 1603.06744. URL: http://arxiv.org/abs/1603.06744.

[PM17] A. Panichella and U. R. Molina. In: *IEEE/ACM 10th International Workshop on Search-Based Software Testing (SBST)*. 2017.

[PV16] T. Parr and J. J. Vinju. In: *CoRR* abs/1606.08866 (2016). arXiv: 1606.08866.

[Rob+11] B. Robinson et al. In: *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*. 2011.

[SSN12] M. Sundermeyer, R. Schlüter, and H. Ney. In: *Thirteenth Annual Conference of the International Speech Communication Association*. 2012.

[SVL14] I. Sutskever, O. Vinyals, and Q. V. Le. In: *Advances in neural information processing systems*. 2014.

[YN17] P. Yin and G. Neubig. In: *CoRR* abs/1704.01696 (2017). arXiv: 1704.01696. URL: http://arxiv.org/abs/1704.01696.

[Zhe+17] W. Zheng et al. In: *arXiv preprint arXiv:1709.07642* (2017).

[ZM15] Y. Zhang and A. Mesbah. In: *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*. 2015.