Utilizing machine learning to solve the equivalent mutant problem

Salman Nasir
Mentored by Josh Dehlinger and Lin Deng

Introduction

The importance of software validation only increases as society's reliance on software increases. For example, the software in self-driving cars must be rigorously tested before it can be widely adopted to avoid injury or even death. The strongest method of software testing is mutation testing.

To ensure a test suite is properly verifying the behavior of a computer program, mutation testing introduces bugs to determine if a test suite can detect the presence of errors and faults. In other words, a new "mutant" code is produced when the original code undergoes a "mutation." If the mutation was detected, the mutant is labeled dead, otherwise alive. If the mutant cannot be killed, it is labeled an "equivalent mutant." These equivalent mutants provide no useful information to the tester and add to the cost of mutation testing.

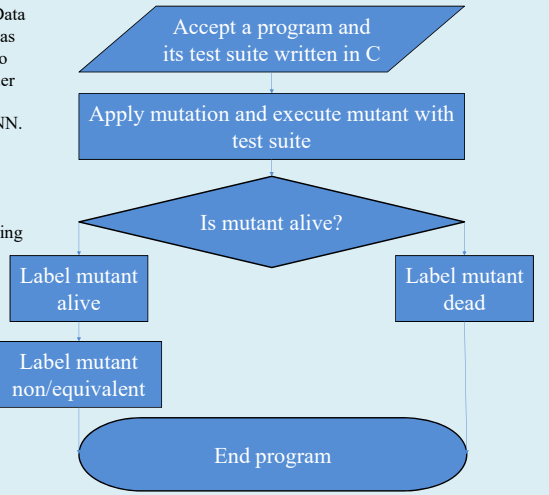
The purpose of this study was to detect equivalent mutants in the mutation testing of C programs using machine learning. The algorithm utilized abstract syntax trees (ASTs) to represent code while capturing its structure and semantics. Previous studies have achieved 75% to 80% accuracy when using machine learning (Strug & Strug, 2012). A different study designed a code similarity algorithm designed around ASTs (Zhang et al., 2019).

Materials and Methods

Data collection was performed according to Figure 1. In order to generate mutants of a subject, a program called Mull was downloaded from the software development platform GitHub (Desinov & Pankevich, 2018). A data set of 18 subjects written in the programming language C was acquired (Yao et al., 2014). Another 29 subjects were also acquired from a commonly used dataset. If the output was the same as the original subject, then the mutant was marked alive. Otherwise, the mutant was marked dead. Equivalence evaluation was performed by multiple people to reduce the chance of a nonequivalent mutant being marked as equivalent. Once all the mutants were marked as equivalent or nonequivalent, it was counted that there were 912 equivalent mutants in total and greater than 912 nonequivalent mutants. A random selection of 912 nonequivalent mutants were selected to keep the number of equivalent and nonequivalent mutants the same. These mutants were then used as training, test, and validation data for an Abstract Syntax Tree Neural Network (ASTNN). The ASTNN was designed as shown in Figure 2. The model was trained, tested, and validated using a ratio of 8:1:1. The ASTNN was trained to minimize loss using a binary cross-entropy loss function, and each mutant was evaluated for equivalence.

Materials and Methods (cont.)

Figure 1 (right): Data collection. This was the method used to collect data in order to train, test, and validate the ASTNN. A program was accepted as input along with its test suite, and then mutated. Then, using the output of the mutant and the original, the mutant was labeled as alive or dead. Alive mutants were labeled as equivalent or nonequivalent by hand.



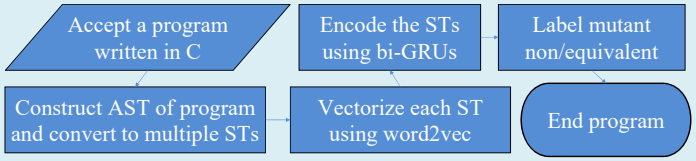


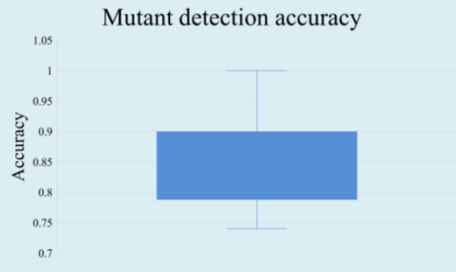
Figure 2 (above): The design of the ASTNN. It accepts a C program as input, and then converts it to an AST. The AST is broken at the statement level into STs to avoid the gradient vanishing problem. Each ST is vectorized and then encoded. Then, the model determines if the program is equivalent or not using a threshold value.

Results

The accuracy for each subject was evaluated by adding the number of true positives (correctly identified equivalent mutants) and the number of true negatives (correctly identified nonequivalent mutants) divided by the total number of mutants of a subject. Then, a one-sample *t*-test was conducted with an alpha level of 0.05 ($M = 0.851$, $SD = 0.070$, $N = 30$) with the null hypothesis that the average accuracy of the model is 90% and the alternate hypothesis that the average accuracy of the model exceeds 90%. The results were not statistically significant ($t(29) = -3.85$, $p = 1.000$) indicating that it cannot be concluded that the accuracy of the model was above 90%. Graph 1 illustrates the data.

Results (cont.)

Graph 1 (right): Graph of the accuracy of the ASTNN. Note how the data was fairly spread out with a small interquartile range. The median was also well below the benchmark of 90%. This indicates that most of the data did not meet the benchmark accuracy.



Conclusion

The purpose of this study was to detect equivalent mutants in the mutation testing of C code. This was done with an average of 85.1% accuracy: slightly higher than research performed by Strug & Strug (2012) but lower than the benchmark accuracy of 90%. The accuracy of this model should be improved before it is used in a professional setting. A larger data set could allow the model to better learn the traits of an equivalent mutant. This study was limited by time, and the data collection phase could have lasted longer given more time to collect a larger data set. The model could only be validated through small sample sizes of simple mutants due to technical limitations of the ASTNN. Further work should consider overcoming these technical limitations. Once these constraints are overcome, the research could be applied to C++ as C++ allows for Object-Oriented Programming (OOP) mutators. This would allow for more mutation types to be analyzed by the model, as there were mutations at the object level not included in this study. This study could also be applied to higher level languages like Java or Python, but this would be more challenging than applying this research to C++ due to language differences.

References

Desinov, A., & Pankevich, S. (2018). Mull it over: mutation testing based on LLVM. *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 25-31.

Strug, J., & Strug, B. (2012). Machine learning approach in mutation testing. *Lecture Notes in Computer Science, 7641*, 200-214.

Yao, X., Harman, M., & Jia, Y. (2014). A study of equivalent and stubborn mutation operators using human analysis of equivalence. *Proceedings of the 36th International Conference on Software Engineering*, 919-930.

Zhang, J., Wang, X., Zhang, H., Sun, H., Wang, K., & Liu, X. (2019). A novel neural source code representation based on abstract syntax tree. *ICSE '19 Proceedings of the 41st International Conference on Software Engineering*, 783-794.