Mutant Selection Using Machine Learning Techniques

SJ Guillaume
Department of Computer Science
Allegheny College
guillaumes@allegheny.edu

***Abstract*—Mutation testing is an effective, but high cost approach to ensuring test suite adequacy. The issue of efficiency emerges. Thousands of mutants can be inserted in a program, and many tests may be run before the mutant is killed, if it is killed at all. This paper proposes that there is a way to reduce the cost of mutation testing employing data mining and machine learning algorithms to reduce the number of mutant operators run. Previous research in mutation testing proves that mutation test prioritization and reduction is possible without resulting in a significantly different mutation score. To our knowledge, no prior research techniques use machine learning models to perform mutant selection, specifically mutation operator selection.**

I. Introduction

MUTATION testing is a method of measuring test suite adequacy. Having a high quality test suite is inherently valuable, if the test suite is reliable in catching mistakes it ensures that real issues can be detected and fixed in a program. This process is important in software development, specifically mobile software development, because with the rate at which program development and updates are demanded and required by the market, we wonder if we are asking too much of programmers. Speed and accuracy do not correlate, and catching errors before releasing a product into the market is essential. Mutation testing can make this process easier, as the purpose of mutation testing is to generate errors in code that can indicate where a program or test suite is weak or incorrect.

The drawback of creating a quality product is the time requirement. While any type of software testing is a time consuming process, mutation testing demands an unreasonably high cost in producing test cases, running time, and development effort. Faster testing is essential. Researchers have investigated ways to reduce the high cost of mutation testing for years. This paper proposes that an efficient, accurate mutation testing framework can be discovered through the application of data mining techniques and applying machine learning algorithms to determine which mutants are unnecessary and can be eliminated. Fewer generated mutants results in reduced time involved in inserting, compiling, and running the test suite.

The purpose of mutation testing is to check that a test suite detects small syntactic errors, similar to mistakes all programmers are susceptible to. The faults produced may not be syntactically similar to those produced by human error according to. [6] However, Just et al.'s work argues that the MAJOR framework does produce faults comparable to real faults. [7] Regardless, mutation testing continues as a highly regarded way to measure test suite adequacy.[16]. Mutation testing is an effective tool for testing software with known faults, and therefore is a useful tool on software for which faults are unknown [12]. An effective mutant is one which catches that there has been a change in the program which alters the result. This works because the test suite is designed to check that the program performs as expected. The test suite may have multiple tests, called test cases, for the same part of the program. If one test case compares the expected result, what the original program should produce, to what the section of the program that has been mutated, that result should differ from the expected result. When this discrepancy occurs, the mutant is killed.

For example, in the MAJOR mutation framework, the mutation operator AOR(Arithmetic operator replacement)will go into the system under test(SUT), and generate mutants. For every $+$, AOR will alter the SUT and insert a $-$ in place of the $+$. Test suite, T, will then run its test cases on SUT. For each segment of a code, there may be multiple test cases in T checking SUT. When a test case runs on SUT, and a difference is detected between what the original program produced, and what the mutated program produced, we consider the mutant k̈illed.̈ When none of the test cases in T detect the difference in SUT, the mutant is considered l̈ive.̈ There are several explanations for a l̈ivem̈utant, including equivalent, redundant, and quasi-mutants which will be discussed later. When T executes all test cases, the mutation score can be calculated. The unmutated version of the system under test(SUT) will be referred to as the original program in this paper. Equivalent mutants are those who do not produce a different result than the original program does. Thus, when T runs all of its test cases on SUT all of the test cases pass, and the mutant is not detected. Similarly, redundant mutants are those which are syntactically the same as parts of the program which are not mutated. Quasi-mutants are different in that they are syntactically incorrect, often referred to as s̈till bornb̈ecause they are not fully formed mutations, it is impossible to check these statements.

There is supporting research on the benefits of mutation selection techniques. Wright et al.'s work shows that the use of many operators is a disadvantage in mutation testing. Too many mutants may defeat the overall purpose because it may create equivalent or redundant mutants that make the

testing less effective than desired.[14] The reduction technique developed in this paper suggests that by running mutation testing suites on a variety of programs, we may be able to utilize machine learning algorithms to produce a reduced set of mutation operators that are able to produce a mutation score comparable to the mutation score of the original set of mutation operators. The goal is for this method to determine set of operators that can be applied to many programs outside the training set, and have an accurate, comparable mutation score.

II. Technique

We will be using the R-language for statistical computing, specifically the caret package for classification and regression testing. We aim to use these tools to perform Data Mining. Data mining involves going through a massive amount of data to identify patterns. [11] Once patterns are identified, we can perform machine learning to select a subset of mutation operators that will produce a mutation score comparable to the original mutation score.

The mutation frameworks we will use in this paper are MAJOR[3] and PIT[5], operators for each are found in Table I and Table II respectively. We will perform mutation testing on programs with automatically created test suites and one manually created test suite. The automatically created test suites will be generated using Codepro[1], EvoSuite[2], and Randoop[4].

Table I contains the Mutation Operators from MAJOR.

Operator	Description
AOR	Arithmetic Operator Replacement
LOR	Logical Operator Replacement
COR	Conditional Operator Replacement
ROR	Relational Operator Replacement
SOR	Shift Operator Replacement
ORU	Operator Replacement Unary
STD	Statement Deletion Operator
LVR	Literal Value Replacement

TABLE I
TABLE OF MUTATION OPERATORS IN MAJOR: ADAPTED FROM $mutation - testing.org/doc/major.pdf$

Table II contains the Mutation Operators from PIT.

Operator	Description
CBM	Conditionals Boundary Mutator
NCM	Negate Conditionals Mutator
RCM	Remove Conditionals Mutator
MM	Math Mutator
IM	Increments Mutator
INM	Invert Negatives Mutator
IC	Inline Constant Mutator
RVM	Return Values Mutator
VCM	Void Method Call Mutator
NVM	Non Void Method Call Mutator
CCM	Constructor Call Mutator
EMM	Experimental Member Variable Mutator
ESM	Experimental Switch Mutator

TABLE II
TABLE OF MUTATION OPERATORS IN PIT: LIST FROM $pitest.org/quickstart/mutators/$

III. Algorithms

A. Basic Algorithms

There are many algorithms that have been developed to reduce the number of mutants generated. For our study, we use three of these techniques for baseline data to compare our machine learning algorithms against.

The first algorithm technique we use is the percentage reduction technique. This involves randomly selecting a set percentage of possible mutants to compile and execute[13].

The second algorithm technique we use is the percentage reduction by mutation operator technique. For each operator, we would reduce the number of mutants compiled and executed by a certain percentage[13]. This method ensures the set of mutants used for measuring the quality of the test suite is representative of the total set of possible mutants.

The third algorithm technique is an operator reduction technique. The idea of reducing an entire operator from testing is controversial, as that reduces all of a type of mutant that the test suite may or may not be capable of handling[16][10].

.

B. Machine Learning Algorithms

This paper aims to perform machine learning on mutation selection using a k-fold cross validation approach and training models with each fold being a single program, thus the model would train on all but one program, and then it would test to see how accurate it is on the remaining program. Once the model has trained, we then combine the trained models to make one model for the set.

Machine learning we perform on this includes boosted class trees, greedy, lars, and clustering.

IV. Evaluation

With the data collected, this paper aims to see if there is a subset of mutation operators that can be applied in any mutation testing, or if factors such as whether the test suite was created automatically or manually impacts the subset of mutation operators selected. Another factor we want to observe is if the subset of operators chosen from MAJOR or PIT are similar in what their functions are, also we can observe if the size of the program impacts the the subset of operators selected. All of these comparisons are important in using machine learning algorithms to select a subset of operators from the training set of programs that can create generalized rules for choosing a subset of mutation operators.

A. Metrics

Mutation score is calculated by dividing the number of killed mutants over the number of total non-equivalent mutants. We do not consider equivalent, redundant, or quasi-mutants in the calculation because they are essentially immune to the test cases due to their similarities to the original program. The mutation score is a number between zero and one, numbers closer to one indicate a higher quality test suite.

B. Benchmarks

To evaluate, ten programs were selected from the SF110 based on their size and associated test suites. These can be found in Table III. This set of programs provides a range of size. The largest program is Netweaver with 17,953 lines of code (LOC), and the smallest program is the Jni-inchi with 783 LOC.

Table III Benchmark Programs and Properties

Program	LOC	Cyclomatic Complexity
Netweaver	17953	2.82
Inspirento	1769	1.76
Jsecurity	9470	2.05
Saxpath	1441	2.10
Jni-inchi	783	2.05
Xisemele	1399	1.29
Diebierse	1539	1.74
Lagoon	6060	3.52
Lavalamp	1039	1.50
Jnfe	1294	1.38

TABLE III
TABLE OF BENCHMARK PROGRAMS

V. Related Work

Zhang et al. prove operator-based mutant selection is not superior to random mutant selection, where operator-based mutant selection only creates mutants on a subset of sufficient operators, and random mutant selection creates a subset of mutants from any mutation operator. The study in [16] proves that operator-based mutant selection and random mutant selection are competitive techniques, and random mutant selection is arguably superior because it chooses fewer mutants than any of the operator-based mutant selection techniques.

Research by Offutt et al. found that there are five operators, out of the 22 available in the MOTHRA mutation testing suite, that are sufficient for mutation testing, and considered better, or equal, to the full set of operators in mutation analysis [10].

Wong et al. show that random selection of mutants can effectively evaluate the quality of a test suite [13]. In this work, they learn that the use of only 10 percent of mutation operators yields a mutation score comparable to a full mutation set. The research shows that reduced mutant testing is a cost effective alternative to mutation testing.

Zhang et al. invent the FaMT technique to prioritize test cases such that ones most likely to kill the mutant are run earlier, and reduce test cases by running only a subset of all possible tests on a mutant in their paper [17]. The FaMT method reduced all executions for all mutants by about 50% but for some programs, it reduced the executions for all mutants by more than 63%. This resulted in a greatly reduced run time overhead.

Just et al. use a prioritization and reduction technique also. Their technique differs by evaluating redundancies and runtime differences of test cases to prioritize and reduce the cost of mutation analysis up to 65% [8].

In [14] removing equivalent, redundant, and quasi mutants leads to a more efficient, effective mutation analysis suite. Wright et al. propose that performing mutation analysis with so many mutants may defeat the overall purpose because it may create so many ineffective mutants. Ineffective mutants reduce accuracy and increase the cost of mutation analysis. Removing the mutants that do not make a valuable contribution to analysis proved to be an effective method for reducing mutation testing cost; there was a 56% decrease in time cost for the majority of schemas, time varied depending on the DBMS used. The mutation score increased for 75% of schemas after removing the ineffective mutants, and in 44% of cases it changed to 1, a perfect mutation score as all the mutants were killed. This is noted a statistically significant result.

Namin et al. use a mutation operator selection method on the Proteum system which generated less than 8% of mutants generated by the full set. They declare this subset is sufficient for determining test suite adequacy[12].

Size and structural coverage are important factors to consider when measuring test effectiveness. Coverage is sometimes correlated with effectiveness, but using both size and coverage gives a better prediction of effectiveness than size alone[9]. Coverage has been the most utilized way to measure test suite quality, it supports the belief that if you execute more of a program, you will be more likely to catch problematic elements. The study by Namin et al. determines that size and coverage independently influence test suite effectiveness, however, the relationship is not linear.

Offutt et. al completed an insightful study into sufficient mutant operators. The results indicate that certain operators, such as mutant operators that replace all operands with all syntactically legal operands and mutant operators that modify entire statements, do not contribute greatly to the effectiveness of mutation testing. Ultimately, this study determines that of the 22 mutant operators in Mothra, only five of these are necessary to effectively implement mutation testing[10]. These five mutation operators are ABS, AOR, LCR, ROR, and UOI.

Regression mutation testing is a technique created by Zhang et al. where they used previous tests on software to determine which mutants should be executed on subsequent tests on updated versions of the software based on kill ability and coverage of statement[18].

Research by Zhang et al. takes a different stance on mutant selection. Instead of choosing between the two types, combining mutation operator selection and random selection allows for even greater testing efficiency to be achieved at a low cost. This method applies random mutant selection on top of operator selection, to further reduce the number of mutants necessary to assert test suite adequacy. Only 5% of mutants are necessary to preserve the mutation score[15].

Our work is distinct from previous advances in mutation selection, reduction, and prioritization because the aim of this paper is to use sophisticated data mining methods to perform mutation operator selection. To our knowledge, no previous work has been done with taking a machine learning approach to mutant selection.

VI. Discussion

There were some difficulties hindering the success of this research. One is the difficulty we had with some programs and

MAJOR in producing the information needed to perform full analysis. Randoop generated test suites often did not produce accurate results, and thus is not suitable for use in comparison with other test suites.

VII. Conclusion and Future Work

This research accomplishes the goal of determining if machine learning techniques produce better mutation subsets than algorithms that have been widely used and studied before this paper was written: random selection and operator selection. While the work done so far in this paper provides a base on which to continue research, implementing more machine learning approaches is a clear goal for future research. With the greedy algorithm as a good indicator of what could be accomplished with machine learning application to mutant selection approaches, we anticipate more advanced machine learning methods will produce further reduced sets of mutants.

VIII. Acknowledgement

This work is supported by an NSF REU Grant.

References

[1] Codepro analytix. https://developers.google.com/java-dev-tools/codepro/doc/?hl=en.

[2] Evosuite: Automatic test suite generation for java. http://www.evosuite.org/.

[3] The major mutation framework. http://www.mutation-testing.org/.

[4] Randoop: Automatic unit test generation for java. http://mernst.github.io/randoop/.

[5] Real world mutation testing. http://www.pitest.org/.

[6] R. Gopinath, C. Jensen, and A. Groce. Mutations: How close are they to real faults? In *Software Reliability Engineering (ISSRE), 2014 IEEE 25th International Symposium on*, pages 189–200, Nov 2014.

[7] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 654–665, New York, NY, USA, 2014. ACM.

[8] R. Just, G. Kapfhammer, and F. Schweiggert. Using non-redundant mutation operators and test suite prioritization to achieve efficient and scalable mutation analysis. In *Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on*, pages 11–20, Nov 2012.

[9] A. S. Namin and J. H. Andrews. The influence of size and coverage on test suite effectiveness. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 57–68, New York, NY, USA, 2009. ACM.

[10] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *ACM Trans. Softw. Eng. Methodol.*, 5(2):99–118, Apr. 1996.

[11] B. Rajagopalan and R. Krovi. Benchmarking data mining algorithms. *Journal of Database Management*, 13(1):25–35, Jan 2002.

[12] A. Siami Namin, J. H. Andrews, and D. J. Murdoch. Sufficient mutation operators for measuring test effectiveness. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 351–360, New York, NY, USA, 2008. ACM.

[13] W. E. Wong and A. P. Mathur. Reducing the cost of mutation testing: An empirical study. *Journal of Systems and Software*, 31(3):185–196, 1995.

[14] C. J. Wright, G. M. Kapfhammer, and P. McMinn. The impact of equivalent, redundant and quasi mutants on database schema mutation analysis. In *Quality Software (QSIC), 2014 14th International Conference on*, pages 57–66. IEEE, 2014.

[15] L. Zhang, M. Gligoric, D. Marinov, and S. Khurshid. Operator-based and random mutant selection: Better together. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 92–102, Nov 2013.

[16] L. Zhang, S.-S. Hou, J.-J. Hu, T. Xie, and H. Mei. Is operator-based mutant selection superior to random mutant selection? In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 435–444, New York, NY, USA, 2010. ACM.

[17] L. Zhang, D. Marinov, and S. Khurshid. Faster mutation testing inspired by test prioritization and reduction. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 235–245, New York, NY, USA, 2013. ACM.

[18] L. Zhang, D. Marinov, L. Zhang, and S. Khurshid. Regression mutation testing. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 331–341, New York, NY, USA, 2012. ACM.