Predicting Mutation Score Using Source Code and Test Suite Metrics

Kevin Jalbert, Jeremy S. Bradbury
Software Quality Research Group
University of Ontario Institute of Technology
Oshawa, Ontario, Canada
{*kevin.jalbert, jeremy.bradbury*}*@uoit.ca*

***Abstract*—Mutation testing has traditionally been used to evaluate the effectiveness of test suites and provide confidence in the testing process. Mutation testing involves the creation of many versions of a program each with a single syntactic fault. A test suite is evaluated against these program versions (mutants) in order to determine the percentage of mutants a test suite is able to identify (mutation score). A major drawback of mutation testing is that even a small program may yield thousands of mutants and can potentially make the process cost prohibitive. To improve the performance and reduce the cost of mutation testing, we propose a machine learning approach to predict mutation score based on a combination of source code and test suite metrics.**

I. Introduction

Mutation testing has traditionally been used as a coverage technique to evaluate the effectiveness of test suites and provide confidence in the testing process [1], [2]. For over 30 years, mutation testing has been applied to software written in programming languages including C [3], [4], Fortran [5] and Java [6], [7]. Furthermore, mutation testing has also been applied to non-programming artifacts such as formal specification languages [8] and spreadsheets [9].

Mutation testing uses a set of *mutation operators* to generate faulty versions of a program called *mutants*. Mutation operators are created based on an existing fault taxonomy and each operator usually corresponds to a specific type of fault. A test suite is evaluated against a set of mutants to determine the *mutation score*. The mutation score is defined as the percentage of non-equivalent mutants that are detected (*killed*) by a test suite. The better a test suite, the more mutants will be killed and thus the higher the mutation score.

A major drawback of mutation testing is that even a small program may yield hundreds or thousands of mutants – potentially making the process cost prohibitive in comparison to other coverage metrics. Three approaches have been proposed to improve mutation testing performance and scalability [10]:

1) **"Do fewer" approach:** this category of optimizations aim to decrease the computational cost of mutation testing by reducing the number of mutants that a test suite is evaluated against. The most popular example from this category is selective mutation – the use of a subset of mutation operators that have been empirically shown to be as effective as using an entire set of operators [11].
2) **"Do smarter" approach:** this category of optimizations aim to decrease the cost of mutation testing by improving the actual mutation testing technique. For example, weak mutation *"...is an approximation technique that compares the internal states of mutant and original program immediately after execution of the mutated portion of the code (instead of comparing the program output)"* [10].
3) **"Do faster" approach:** this category of optimizations aim to reduce the cost of mutation testing by focusing on performance. For example, one "do faster" approach improves compilation time using schema-based mutation – *"...encodes all mutations into one source level program..."* [10].

As an alternative to the above approaches, we propose a "do fewer and smarter" technique for mutation testing at the unit level. When mutation testing is used for the creation or improvement of a test suite, the test suite will often have to be applied to the mutants in an iterative fashion as tests are added, removed and modified. Furthermore, the effects on the mutation score after each iteration have to be observed. We propose to replace at least some of the mutation testing of intermediate tests with mutation score prediction and thus decrease the number of mutants that have to be evaluated using a test suite. Our proposed approach uses machine learning to predict the mutation score based on a combination of source code and test suite metrics of the code unit under test.

Next, in Section II we describe the machine learning technique, the mutation testing tool and the metrics gathering tools used in our research. In Section III we describe our overall approach to mutant score prediction. In Section IV we apply our approach to an open source project and discuss the effectiveness of our prediction based on this preliminary evaluation. Finally, in Sections V & VI we discuss related work, conclusions, and future work.

II. Background

In this section we describe the background techniques and tools used in our research. All tools were selected based on the ability to execute command-line using a script, the ability to export output reports, and the ability to fulfill the needs of our research.

A. LIBSVM – A Library for Support Vector Machines

A support vector machine (SVM) is an example of a linear discrimination machine learning technique and assumes that "*...instances of a class are linearly separable from instances of other classes*" [12]. A SVM is capable of learning how a set of features inform the classification of a data item. It can also be trained on known data items (in which the features and category are present), and then used to predict the category of unknown data items (in which only the features are present).

Traditionally, SVMs have been used for two-group classification problems [13] but have also been generalized to *n*-group classification problems. In our research we use LIBSVM (version 3.11), a SVM library capable of solving *n*-group classification problems [14].

B. Javalanche – A Mutation Testing Tool for Java

Mutation testing has previously been described in Section I. In our research we use Javalanche (version 0.4), a mutation testing tool for Java [15] that applies a subset of the method-level mutation operators. Method-level operators are the most common set of mutation operators applied to procedural programming languages and each method-level mutation defines a syntactic change to an operator, operand or statement in a program. Javalanche uses a subset of the method-level mutation operators (replace constant, negate jump, arithmetic replace, remove call, replace variable, absolute value, unary operator). These selected operators provide a close approximation of the effectiveness of using the entire set of method-level operators at a reduced cost [11].

We chose Javalanche for our research because it is customizable and extensible, therefore allowing us to modify Javalanche to calculate unit mutation scores and output a richer set of results. We plan to further take advantage of Javalanche's extensibility in the future by adding object-oriented mutation operators. Other benefits of Javalanche include: full integration with JUnit and the use of mutation schemas to improve performance.

C. Eclipse Metrics Plugin – A Source Code Metrics Tool

Source code metrics give insight into various aspects of the source code including it's complexity, size, coupling, cohesion as well as object-oriented attributes [16]–[21]. In our research, we use the Eclipse Metrics Plugin (version 1.3.8.20100730-001) to acquire source code metrics of the method- and class-level code unit under test [22]. We selected this tool as it provides a comprehensive set of metrics for Java programs (see feature sets ① & ③ from Table I).

D. EMMA – A Test Suite Coverage Tool

Test suite metrics can be gathered using similar technique to those used in the gathering of source code metrics. In fact, since we focus on JUnit test cases (which are Java classes) we can actually use the Eclipse Metrics Plugin to gather some of the test suite metrics (see feature set ④ from Table I). To gather other test suite metrics we use EMMA (version 2.0.5312) which is capable of determining the statement and basic block coverage of a test suite [23]. We use EMMA to acquire metrics for feature set ② from Table I.

III. Approach

We now describe our SVM approach to determining the mutation score of a unit under test based on source code and test suite metrics. In Section III-A we outline the process for collecting our initial data and training the SVM (see Figure 1) and in Section III-B we describe how to use the SVM for prediction.

Table I
THE SET OF METRICS USED IN OUR RESEARCH. ORGANIZED BY FEATURE SET (①: SOURCE CODE METRICS, ②: COVERAGE METRICS, ③: ACCUMULATED SOURCE CODE METRICS, ④: ACCUMULATED TEST CASE METRICS)

Metrics	Description	Scope	Set
MLOC	Method lines of code	Method	①
NBD	Nested block depth	Method	①
VG	McCabe cyclomatic complexity	Method	①
PAR	Number of parameters	Method	①
NORM	Number of overridden methods	Class	①
NOF	Number of attributes	Class	①
NSC	Number of children	Class	①
DIT	Depth of inheritance tree	Class	①
LCOM	Lack of cohesion of methods	Class	①
NSM	Number of static methods	Class	①
NOM	Number of methods	Class	①
SIX	Specialization index	Class	①
WMC	Weighted method per class	Class	①
NSF	Number of static attributes	Class	①
BCOV	Basic blocks covered in code unit	Class	②
BTOT	Total basic blocks for code unit	Class	②
SMLOC	Sum MLOC of methods	Class	③
SNBD	Sum NBD of methods	Class	③
SVG	Sum VG of methods	Class	③
SPAR	Sum PAR of methods	Class	③
AMLOC	Average MLOC of methods	Class	③
ANBD	Average NBD of methods	Class	③
AVG	Average VG of methods	Class	③
APAR	Average PAR of methods	Class	③
STMLOC	Sum MLOC of test methods	Both	④
STNBD	Sum NBD of test methods	Both	④
STVG	Sum VG of test methods	Both	④
STPAR	Sum PAR of test methods	Both	④
ATMLOC	Average MLOC of test methods	Both	④
ATNBD	Average NBD of test methods	Both	④
ATVG	Average VG of test methods	Both	④
ATPAR	Average PAR of test methods	Both	④

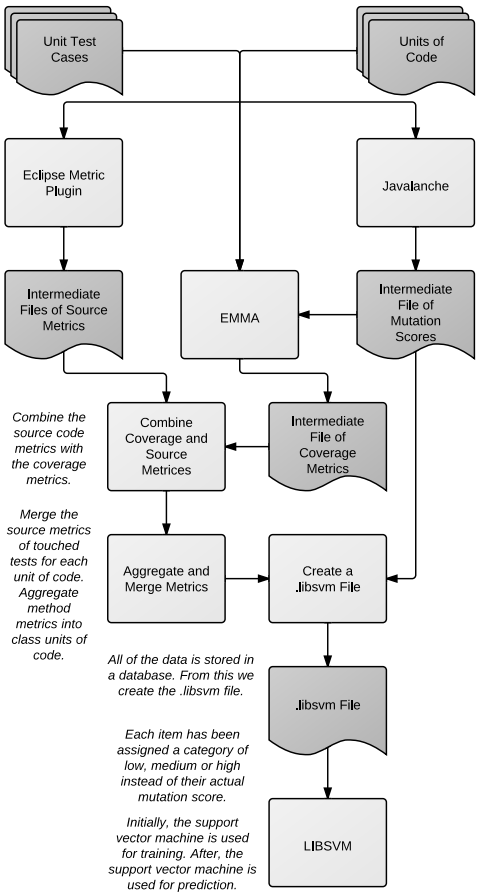

Figure 1. Our SVM training process.

A. Training

Our training process requires as input a set of units of code and for each code unit a corresponding set of unit test cases. Both the code units under test and test cases (i.e., JUnit tests) are Java source files.

The first step of our process is to collect the two types of data required to train the SVM:

- **Category Data:** Javalanche is used to generate the mutants of all code units in the project under test and to perform the mutation testing by executing the required tests for each mutant. It should be noted that currently our Javalanche configuration does not exclude equivalent mutants from the analysis. For our research, we added a new analyzer component to Javalanche that outputs an intermediate text file of mutation scores of the covered units (methods and classes).
- **Feature Data:** The feature data is comprised of the source code and test suite metrics (see Table I for feature sets referenced in the following text). By using the Eclipse Metric Plugin we collect source code metrics (sets ① & ③) for both method- and class-level code units. We also collect the test suite metrics (set ④) of each unit's test cases using the Eclipse Metric Plugin. Test suite coverage metrics (set ②) are obtained using the EMMA tool.

Next, we create a .libsvm file containing the category and feature data from the database. Instead of predicting a specific mutation score percentage, we categorize all mutation scores as *low, medium, high* which reduces the mutation score prediction to a three-group classification problem. The ranges of values in each category are determined based on the distribution of the mutation scores in our training data (further explained in Section IV-A). Finally, the .libsvm file is passed into LIBSVM to complete the training process.

B. Prediction

Once we have trained the SVM, we can then use the SVM for prediction. We can predict the mutation score category of an unknown unit of code by first determining the source code and test suite metrics. The metrics (i.e., features) are passed into the SVM which will then assign a category of *low, medium, high* for the mutation score. Currently, our approach only predicts mutation scores within a project. That is, the training and prediction data both come from the same project repository.

IV. Case Study: JGAP

A preliminary evaluation of our mutation score predictor was preformed on JGAP (version 3.6.1), an open source genetic algorithm and genetic programming framework for Java, which was selected due to its comprehensive and mature JUnit test suite [24]. Basic source code metrics for JGAP are provided in Table II. JGAP contains 1387[1] usable JUnit test cases. For JGAP, Javalanche generated 32031 mutants using the method-level mutation operators (see Section II-B) of which only 18378 were actually covered by JGAP's test suite. We only considered the set of covered mutations in our approach as the mutations not covered contained no associated test cases. Considering only the covered mutants, JGAP's test suite killed 13698 mutants in 688 minutes and 43 seconds[2] resulting in an overall mutation score of 74.53%.

[1]JGAP has 1412 JUnit test cases in total, however 25 of the tests caused errors in the Javalanche tool and were removed.

[2]Results from Javalanche on Intel Core i7-870 processor @ 2.93 GHz (with no parallel task execution).

Table II
The amount of classes, methods, lines of code (LOC), and JUnit test cases in the JGAP repository

JGAP Source Artifacts	# in Source	# in Test
Classes	415	180
Methods	3017	1626
LOC	28975	19556
JUnit Test Cases	–	1412[1]

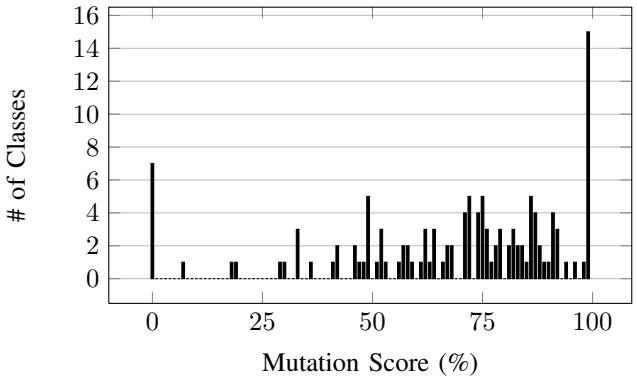

Figure 2. The mutation score distribution of JGAP's classes that can be used for training.

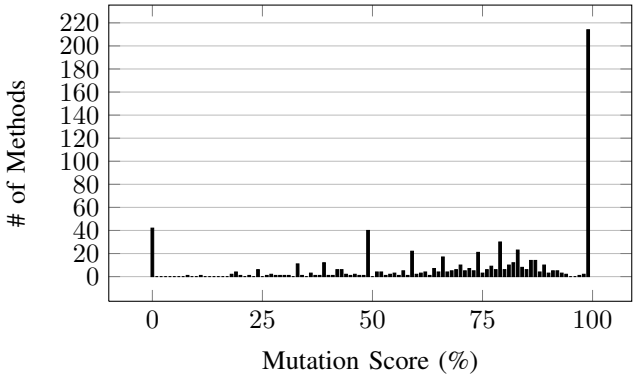

Figure 3. The mutation score distribution of JGAP's methods that can be used for training.

A. Results

We gathered 695 method-level data points and 127 class-level data points from JGAP. We first decided to view the distribution of JGAP's mutation scores (see Figure 2 & 3). It is clearly obvious that the distribution for both classes and methods are more heavily dense for high mutation scores. As discussed in Section III-A we categorize the collected data into three groups of *low, medium, high* mutation scores. Ideally we would have liked to have sufficient data coverage, such that we can use simple categories such as 0%–33% (*low*), 33%-66% (*medium)* and 66%-100% (*high*). Based on the distribution of mutation scores in JGAP we instead decided to categorize our data such that one third of our data fall into each category (see Table III).

Table III
THE MUTATION SCORE RANGES THAT EQUALLY PARTITION THE DATASETS INTO THREE MUTATION SCORE CATEGORIES.

Category	Class Mutation Score	Method Mutation Score
low	0.00% – 62.75%	0.00% – 66.66%
medium	62.75% – 83.25%	66.66% – 90.90%
high	83.25% – 100.00%	90.90% – 100.00%

		Prediction Value		
		L	**M**	**H**
Actual Value	**L′**	29, 108	7, 54	7, 74
	M′	11, 43	20, 112	11, 72
	H′	8, 31	9, 40	25, 161

Figure 4. Confusion matrix for (class, method) mutation score prediction on JGAP using all feature sets from Table I. The first value in each cell represents the class value while the second represents the method value. **L** = Low, **M** = Medium and **H** = High.

Table IV
THE CROSS VALIDATION (10-FOLDS) ACCURACY OF EACH FEATURE SUBSET (SEE TABLE I) AND ALL OF FOUR FEATURE SUBSETS COMBINED.

Set	Class Accuracy	Method Accuracy
①	53.54%	48.77%
②	49.61%	47.63%
③	45.67%	49.78%
④	54.33%	33.96%
① ② ③ ④	**58.27%**	**54.82%**

Due to the small amount of data acquired from this experiment we decided to evaluate our results by performing a 10-fold cross-validation of our datasets. Our achieved cross-validation accuracy of JGAP (with all features from Table I) was 58.27% for class mutation scores and 54.82% for method mutation scores. Figure 4 illustrates a confusion matrix of the predicted values created from the SVM data for our three categories. It is also worth mentioning that our achieved accuracy outperforms random (which has an accuracy of 33.33% given our category distributions) by 24.94% for classes and 21.49% for methods.

We also conducted the cross-validation using subsets of our feature set and found that the combination of both source code and test suite features slightly improves the prediction accuracy (see Table IV).

V. Related Work

The use of software metrics to locate faults in source code has been well researched. For example, Koru et al. utilized static software measure along with defect data at the class level to predict bugs using machine learning [25]. Similarly, Gyimothy et al. used object-oriented metrics with logistic regression and machine learning techniques to identify faulty classes in open source software [26]. Finally, design level metrics were used with a linear prediction model to determine the estimated maintainability and error prone modules of large software systems [27]. Our work is unique in comparison to these previous works since we not only use source code metrics but we also use test suite metrics to enhance our predication capabilities.

VI. Conclusions & Future Work

Our technique for predicting mutation score using source code and test suite metrics outperforms random with an

achieved accuracy of 58.27% and 54.82% with the JGAP data, for classes and methods respectively. These results have not been optimized and we believe that with further enhancements and a more tailored feature set we may be able to increase the prediction accuracy.

Despite the promising initial results there is an obvious threat to external validity since we have applied our predictive technique to a single open source project – JGAP. Additionally, we performed training and prediction from the same project. As stated by Kitchenham and Mendes *"It is invalid to select one or two datasets to 'prove' the validity of a new technique because we cannot be sure that, of the many published datasets, those chosen are the only ones that favour the new technique"* [28]. Thus, we plan to evaluate more open source projects using our prediction technique to better assess the prediction accuracy. With more data we plan to investigate whether cross-project models are valid for mutation score prediction. We would also like to consider projects with more varied mutation scores to explore the variation in prediction accuracy between strong and weak test suites. A final area of future work is to expand the set of mutation operators used to include object oriented and concurrency operators.

REFERENCES

[1] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *IEEE Computer*, vol. 11, no. 4, pp. 34 –41, Apr. 1978.

[2] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Trans. on Soft. Eng.*, vol. 37, no. 5, pp. 649–678, Sep.–Oct. 2011.

[3] M. E. Delamaro and J. C. Maldonado, "Proteum – a tool for the assessment of test adequacy for C programs," in *Proc. of the Conf. on Performability in Computing Sys. (PCS'96)*, 1996, pp. 79–95.

[4] Y. Jia and M. Harman, "MILU: A customizable, runtime-optimized higher order mutation testing tool for the full C language," in *Testing: Academic & Industrial Conf. – Practice and Research Techniques (TAIC PART 2008)*, 2008, pp. 94 – 98.

[5] K. N. King and A. J. Offutt, "A Fortran language system for mutation-based software testing," *Software: Practice and Experience*, vol. 21, no. 7, pp. 685–718, 1991.

[6] Y.-S. Ma, Y.-R. Kwon, and J. Offutt, "Inter-class mutation operators for Java," in *Proc. of the 13th Int. Symp. on Soft. Reliability Eng. (ISSRE 2002)*, 2002, pp. 352–363.

[7] J. S. Bradbury, J. R. Cordy, and J. Dingel, "Mutation operators for concurrent Java (J2SE 5.0)," in *Proc. of the 2nd Work. on Mutation Analysis (Mutation 2006)*, Nov. 2006, p. 11.

[8] P. E. Ammann, P. E. Black, and W. Majurski, "Using model checking to generate tests from specifications," in *Proc. of the 2nd IEEE Int. Conf. on Formal Eng. Methods (ICFEM'98)*, 1998, pp. 46–54.

[9] R. Abraham and M. Erwig, "Mutation Operators for Spreadsheets," *IEEE Trans. on Soft. Eng.*, vol. 35, no. 1, pp. 94–108, Jan.–Feb. 2009.

[10] J. Offutt and R. H. Untch, "Mutation 2000: Uniting the orthogonal," in *Proc. of Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries*, 2000, pp. 45–55.

[11] A. J. Offutt, A. Lee, and G. Rothermel et al., "An experimental determination of sufficient mutant operators," *ACM Trans. Softw. Eng. Methodol.*, vol. 5, no. 2, pp. 99–118, Apr. 1996.

[12] E. Alpaydin, *Introduction to Machine Learning*. The MIT Press, 2004.

[13] C. Cortes and V. Vapnik, "Support-vector networks," *Mach. Learn.*, vol. 20, no. 3, pp. 273–297, Sept. 1995.

[14] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Trans. Intell. Syst. Technol.*, vol. 2, pp. 27:1–27:27, 2011, software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

[15] D. Schuler and A. Zeller, "Javalanche: Efficient mutation testing for Java," in *Proc. of ESEC/FSE '09*, 2009, pp. 297–298.

[16] C. Stein, G. Cox, and L. Etzkorn, "Exploring the relationship between cohesion and complexity," *J. of Computer Science*, vol. 1, no. 2, pp. 137–144, Apr. 2005.

[17] T. McCabe, "A complexity measure," *IEEE Trans. on Soft. Eng.*, no. 4, pp. 308–320, Dec. 1976.

[18] S. H. Kan, *Metrics and Models in Software Quality Eng.*, 2nd ed. Addison-Wesley Longman Publishing Co., 2002.

[19] T. Honglei, S. Wei, and Z. Yanan, "The research on software metrics and software complexity metrics," in *Proc. of Int. Forum on Computer Science-Technology and Applications (IFCSTA '09)*, 2009, pp. 131–136.

[20] B. Henderson-Sellers, *Object-oriented metrics: measures of complexity*. Prentice-Hall, Inc., 1996.

[21] A. Shaik, D. C. R. K. Reddy, and D. A. Damodaram, "Object oriented software metrics and quality assessment: Current state of the art," *Int. J. of Computer Applications*, vol. 37, no. 11, pp. 6–15, Jan. 2012.

[22] "Eclipse Metrics plugin," web page: http://metrics2.sourceforge.net/, (last accessed Mar. 1, 2012).

[23] V. Roubtsov, "EMMA: A free Java code coverage tool," web page: http://emma.sourceforge.net/, (last accessed Mar. 1, 2012).

[24] K. Meffert et al., "JGAP – Java genetic algorithms and genetic programming package," web page: http://jgap.sourceforge.net/, (last accessed Mar. 1, 2012).

[25] A. G. Koru and H. Liu, "Building effective defect-prediction models in practice," *IEEE Software*, vol. 22, no. 6, pp. 23–29, Nov.–Dec. 2005.

[26] T. Gyimothy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," *IEEE Trans. of Soft. Eng.*, vol. 31, no. 10, pp. 897–910, Oct. 2005.

[27] S. Muthanna and K. Kontogiannis et al., "A maintainability model for industrial software systems using design level metrics," in *Proc. of the 7th Working Conf. on Reverse Eng. (WCRE 2000)*, 2000, pp. 248–256.

[28] B. Kitchenham and E. Mendes, "Why comparative effort prediction studies may be invalid," in *Proc. of the 5th Int. Conf. on Predictor Models in Soft. Eng. (PROMISE '09)*, 2009, pp. 1–5.