Predictive Mutation Testing

Jie Zhang[1], Ziyi Wang[1], Lingming Zhang[2], Dan Hao[1]*, Lei Zang[1], Shiyang Cheng[2], Lu Zhang[1]
[1]Key Laboratory of High Confidence Software Technologies (Peking University), MoE
Institute of Software, EECS, Peking University, Beijing, 100871, China
[2]Department of Computer Science, University of Texas at Dallas, 75080, USA
[1]{ zhangjie_marina,ziyiwang,haodan,lzang,zhanglucs}@pku.edu.cn
[2]{lingming.zhang,sxc145630}@utdallas.edu

ABSTRACT

Mutation testing is a powerful methodology for evaluating test suite quality. In mutation testing, a large number of mutants are generated and executed against the test suite to check the ratio of killed mutants. Therefore, mutation testing is widely believed to be a computationally expensive technique. To alleviate the efficiency concern of mutation testing, in this paper, we propose *predictive mutation testing* (PMT), the first approach to predicting mutation testing results without mutant execution. In particular, the proposed approach constructs a classification model based on a series of features related to mutants and tests, and uses the classification model to predict whether a mutant is killed or survived without executing it. PMT has been evaluated on 163 real-world projects under two application scenarios (i.e., cross-version and cross-project). The experimental results demonstrate that PMT improves the efficiency of mutation testing by up to 151.4X while incurring only a small accuracy loss when predicting mutant execution results, indicating a good tradeoff between efficiency and effectiveness of mutation testing.

CCS Concepts

•Software and its engineering → Software testing and debugging;

Keywords

software testing; mutation testing; machine learning

1. INTRODUCTION

Mutation testing [14, 22] is a powerful methodology for checking test suite quality. In (first order [26]) mutation testing, a set of program variants (i.e., *mutants*) are generated from the original program based on a set of transformation rules (i.e., *mutation operators*) that seed one syntactic change (e.g., deleting a statement) at a time to generate one mutant. A mutant is said to be *killed* by a test suite if at least one test from the suite has different execution results on the mutant and the original program. Otherwise, the mutant is said to be *survived* (i.e., or *alive*). Based on such mutant execution results, the ratio of killed mutants to all the non-equivalent mutants[1] is defined as *mutation score*, which is widely used to evaluate a test suite's effectiveness.

Besides evaluating test suite quality, mutation testing has also been shown to be suitable for simulating real faults in software testing experimentation [4, 36, 42], localizing faults [55, 83, 48], model transformation [49], and guiding test generation [8, 82, 56, 16, 74, 24].

Although useful in various areas, mutation testing is extremely expensive [32] because it requires generating and executing each mutant against the test suite. The generation and execution of the large number of mutants can both be costly. Mutant generation cost has been greatly reduced by various techniques [13, 67], while mutant execution remains expensive in spite of various refinement techniques [32], e.g., selective mutation testing [54, 18], weak mutation testing [27], high-order mutation testing [25], optimized mutation testing [80, 81], and so on. Moreover, developers in industry also complain about such high cost of mutation testing, and wish to have a compromise with respect to early feedback and efficiency [1].

Due to the high cost of mutant execution in mutation testing, we come up with the question – **"Can we obtain mutation testing results without mutant execution?"**. Motivated by this, we propose *predictive mutation testing* (PMT), the first mutation testing approach that predicts mutant execution results (i.e., killed or survived) without any execution. Specifically, in this paper, PMT applies machine learning to build a predictive model by collecting a series of easy-to-access features (e.g., coverage and mutation operator) on already executed mutants of earlier versions of the project (i.e., *cross-version* prediction) or even other projects (i.e., *cross-project* prediction). That is, the classification model is built offline based on the mutation-testing information (including the features of each mutant and its execution result) of existing versions or projects. Based on this model, PMT predicts the mutation testing results (i.e., whether each mutant is killed or not) of a new version or project without executing its mutants at all.

PMT has been evaluated on 163 projects under the two application scenarios: *cross-version* scenario and *cross-project* scenario. Under each scenario, we evaluate PMT's effectiveness and efficiency. First, we use a set of metrics (i.e.,

* Corresponding author

[1] The mutants semantically equivalent to the original program are called equivalent mutants, which can never be killed.

precision, recall, F-measure, and AUC) to evaluate the effectiveness of PMT in predicting the mutant execution results (i.e., killed or survived). Note that PMT can also be used to predict the mutation score of the whole project based on the ratio of the mutants predicted as killed to all the available mutants. Therefore, we also use the prediction error (i.e., difference between predicted and real mutation scores) to evaluate the effectiveness of PMT in predicting mutation scores. Second, we record the overhead of PMT to evaluate its efficiency.

The experimental results show that PMT performs surprisingly well: for the cross-version scenario, PMT achieves over 0.90 precision, recall, F-measure, and AUC for 36 out of 39 predictions; for the cross-project scenario, PMT achieves over 0.85 AUC and lower than 15% error on predicted mutation scores for 146 out of the 163 projects. Furthermore, PMT is shown to be more accurate than coverage-based testing in predicting mutation scores, demonstrating a promising future for using predictive mutation testing to measure test suite quality. On the other side, PMT is shown to be much more efficient than traditional mutation testing (with speedups between 15.2X to 151.4X). Therefore, when predicting mutation testing results, PMT improves the efficiency of mutation testing to a large extent while incurring only a small accuracy issue, demonstrating a good tradeoff between efficiency and effectiveness in mutation testing. The paper makes the following contributions.

- **Dimension.** The paper opens a new dimension in mutation testing which predicts mutation testing results without mutant execution.
- **Approach.** The paper proposes PMT, a machine-learning-based approach to predicting mutation testing results using easy-to-access features, e.g., coverage information, oracle information, and code complexity.
- **Study.** The paper includes an extensive empirical study of PMT on 163 real-world Java projects under two application scenarios.
- **Analysis.** The paper includes detailed analysis of the effectiveness of the various choices in implementing PMT, as well as the impact of its application factors.

2. APPROACH

As it is costly to execute mutants, mutation testing is widely recognized to be expensive [32]. To alleviate this issue, we present a new dimension of mutation testing - predictive mutation testing (PMT) - which predicts mutation testing results without mutant execution. Typically, a mutant has two alternative execution results, killed or survived, and thus predicting the results of mutant execution can be simplified as a binary classification problem.

In this paper, we use machine learning to solve this problem. In particular, PMT builds a classification model by analyzing the training mutants each of which has a label representing its mutant execution result (i.e., killed or survived) and some easy-to-access features that are related to the label. Then the classification model built by PMT can be used to predict whether any new mutant is killed or survived based on the same types of features.

To build a classification model, it is necessary to make clear the following two aspects: (1) what features are used to build the classification model, in terms of which we should use information that is related to mutation testing results, and is easy to access due to the efficiency concern (see Section 2.1); (2) what classification algorithm is used to build the classification model (see Section 2.2). Furthermore, we consider the impact of imbalanced data (see Section 2.3).

2.1 Identified Features

We identify various features that are related to mutation testing results based on the PIE theory [68, 35], according to which a mutant can be killed by a test only if it satisfies the following three conditions: (1) *execution,* the mutated statement is executed by the test; (2) *infection,* the program state is affected immediately after the execution of the mutated statement so as to distinguish the mutant from the original program; and (3) *propagation,* the infected program state propagates to the test output so that the test output of the mutant is different from that of the original program. Instead of executing the mutant, we may predict whether a mutant is killed by statically analyzing whether it may satisfy these conditions. In other words, in PMT, we should identify features that represent the preceding three conditions and predict whether a mutant is killed based on its values of these features. Therefore, we use the following three groups of features:

2.1.1 Execution Features

For the first condition, we must identify features that are necessarily related to the execution of the mutated statements, which are grouped as a set of execution features. In particular, we consider the following two specific features.

- ***numExecuteCovered***, which refers to how many times the mutated statement is executed by the whole test suite.
- ***numTestCovered***, which refers to how many tests from the test suite cover the mutated statement.

These two features characterize the execution of mutated statements by the test suite as a whole and by each individual test analyzing the execution information of the original program rather than the mutants themselves. In particular, the original program is executed against the given test suite, recording whether each statement is executed by each test and how many times each statement is executed by each test, which is used to calculate *numExecuteCovered* and *numTestCovered*. Clearly, if a mutant is killed, it must be covered by a test, and thus these two execution features may be dominating in predicting whether a mutant is survived.

2.1.2 Infection Features

In terms of the second condition, we must identify some features that are necessarily related to the infection of the mutated statements, which are grouped as a set of infection features. Intuitively, modification on a statement does not necessarily change the program state, which depends on the types of statements and the types of changes. We consider the following two specific features:

- ***typeStatement***, which refers to the type of the mutated statement, e.g., expression, conditional statement, return statement, or method invocation[2].
- ***typeOperator***, which refers to the type of the mutation operator.

[2] Note that for Java they can be easily categorized by analyzing bytecode instruction types, e.g., INVOKEVIRTUAL denotes method invocation while IRETURN denotes return statements.

These two features characterize the infection of mutated statements, and are collected through static analysis on generated mutants.

2.1.3 Propagation Features

In terms of the third condition, we must identify some features that are necessarily related to the propagation of infected program states. Intuitively, whether the program state propagates is related to the complexity of the program under test. In particular, when the structural units (i.e., methods, classes) containing the mutation are complex, the infected program state may not propagate. Therefore, PMT uses the following set of complexity features to characterize to what extent infected program states propagate during execution:

- ***infoComplexity***, which refers to the McCabe Complexity [45] of the mutated method.
- ***depInheritance***, which refers to the maximum length of a path from the mutated class to a root class in the inheritance structure.
- ***depNestblock***, which refers to the depth of nested blocks of the mutated method.
- ***numChildren***, which refers to the total number of direct subclasses of the mutated class.
- ***LOC***, which refers to the number of lines of code in the mutated method.
- ***Ca***, which refers to the number of classes outside the mutated package that depend on classes inside the package.
- ***Ce***, which refers to the number of classes inside the mutated package that depend on classes outside the package.
- ***instability***, which is computed based on the *Ca* and *Ce* values, i.e., $Ce/(Ce+Ca)$.

Furthermore, to learn whether a mutant is killed or survived, it is necessary to know whether the tests are equipped with effective oracles so as to be capable of observing the behavior difference between the mutant and the original program. In particular, a test oracle is a mechanism to determine whether the program under test behaves as expected [75]. In practice, developers usually write test assertions in tests, which are actually one type of test oracles. Besides the program output, these assertions aid to distinguish the execution results of mutants from the original program. In the other word, if a test has no assertion and the program has no output, the difference between the execution results of mutants and the original program may not be observed at all. Therefore, to characterize to what extent the behavior difference (between the mutant and the original program) can be observed, our approach uses a set of features that are related to test oracles, which include:

- ***typeReturn***, which refers to the return type of the mutated method.
- ***numMutantAssertion***, which refers to the total number of assertions in the test methods that cover each mutant.
- ***numClassAssertion***, which refers to the total number of test assertions inside the mutated class's corresponding test class.

The feature *typeReturn* characterizes the ability of the program output on observing the execution difference between mutants and the original program, whereas the features *numMutantAssertion* and *numClassAssertion* characterize the ability of test assertions on observing the execution results.

2.2 Classification Algorithm

Machine learning is proposed to learn some models through training data and then make prediction, and has been widely used in solving software testing problems [9, 76]. Considering the output of machine learning, in this paper we use classification methodology, which learns a classification model from some instances and then classifies new instances into different categories. There are many classification algorithms, e.g., decision tree classifiers, (deep) neural networks, support vector machines (SVM), and so on [47, 50, 58]. In this paper, we adopt *Random Forest*, which is a generalization of tree-based classification, as our default classification technique, because *Random Forest* greatly improves the robustness of classification models by maintaining the power, flexibility, and interpretability of tree-based classifiers [6], and has been shown to be more effective in dealing with imbalanced data [41].

Random Forest constructs a multitude of decision trees at training time and outputs the class based on the voting of these decision trees. In particular, *Random Forest* first generates a series of decision trees, each of which is generated based on the values of a set of random vectors (denoted as Θ_k (k=1,...)), independent of the past random vectors $\Theta_1, \ldots, \Theta_{k-1}$ but with the same distribution. The decision tree is denoted as $H(X, \Theta_k)$, where X is an input vector representing the set of instances used to build the decision tree. For any input X, each decision tree of *Random Forest* gives its classification result and *Random Forest* decides the final classification result of X based on these decision trees. That is, *Random Forest* can be viewed as a classifier on a set of decision tree classifiers $\{H(X, \Theta_k), k = 1, \ldots\}$[6].

Note that although we use *Random Forest* as the default algorithm in this work, PMT is not specific to *Random Forest* and our experimental study (in Section 4) also investigates the choice of other classification algorithms, such as *Naive Bayers*, *SVM*, and *J48*.

2.3 Imbalanced Data Issue

In practical mutation testing, the number of killed mutants may differ a lot with that of survived mutants, as reported by Table 1. This is a typical imbalanced data problem in machine learning, which is mostly solved through the following two strategies. First, *cost sensitive*, which assigns different costs to labels (e.g., killed or survived) according to the distribution of the training data. Second, *undersampling*, which gets balanced data by removing some instances of the dominant category from the training set. However, these strategies also have weak points: they may decrease the effectiveness of a classification model by assigning higher costs to essential instances or removing essential instances.

In our approach, we use the *naive* strategy by default, which uses imbalanced data directly without any further handling. We also investigate the choice of these strategies (i.e, *naive*, *cost sensitive*, and *undersampling*) in Section 4.

3. APPLICATION MODE

For any project, PMT can predict the execution results of all its mutants based on a classification model that is built ahead of time. The classification model can be built either based on other projects or based on the previous versions of this project. Thus, there are two main application scenarios for PMT: (1) *cross-version*, where PMT uses the mutation testing results of old versions to predict the mutation testing results of a new version. During the development cycle of the new version, the classification model can be built beforehand based on the earlier version(s). When the new version is ready, only those easy-to-access features need to be collected for the new version and the mutation testing results can be directly predicted for the new version. (2) *cross-project*, where PMT uses the mutation testing results of other projects to predict the mutation testing results in a new project. The users can train a classification model based on other projects beforehand, and then use that model to predict the test suite quality for a new project. Under both scenarios, the cost for a new project is collecting those easy-to-access features and making prediction.

Furthermore, in the near future we will also release the online PMT service in the cloud, in which a large number of real-world projects from various open-source communities (e.g., Github and SourceForge) are used to build a classification model, so that developers across the world can get the predictive mutation testing results of a project by only uploading its source code. Concerned with the property of source code, developers can also use this service by uploading the required features of a project (even a subset of features) rather than its source code.

4. EXPERIMENTAL SETUP

Our experimental study is designed to answer the following research questions.

- **RQ1:** How does PMT perform in predicting mutation testing results under the two application scenarios in terms of effectiveness and efficiency? This research question is to investigate whether PMT predicts mutation testing results accurately and efficiently.
- **RQ2:** How do different application factors (i.e., mutation tools, and types of tests) influence the effectiveness of PMT? This research question is to investigate the application artifacts that PMT fits for.
- **RQ3:** How do different configurations (i.e., features, classification algorithms, and imbalance strategies) of PMT influence its effectiveness? This research question is to investigate the impacts of different configurations on PMT.

4.1 Implementation and Supporting Tools

Mutation testing tools. We use two popular Java mutation tools: *PIT*[3] and *Major*[4], both of which have been widely used in mutation testing research [64, 29]. As our work targets at solving the cost problem of mutation testing, we choose PIT as the primary mutation testing tool since it is evaluated to be the most efficient mutation testing tool [11]. Moreover, PIT has been demonstrated to be more robust [11], enabling large-scale experimental study. Besides, to further investigate PMT's performance on different tools, we use Major as an auxiliary tool, since it is also widely used and can generate mutants representative for real faults [36]. Note that we use all the 14 operators of PIT and 8 operators of Major .

[3] http://pitest.org/

[4] http://mutation-testing.org/

Feature collecting tools. We use *Cobertura*[5] to collect information for the coverage-related features (i.e., *numExecuteCovered* and *numTestCovered*). For the infection features, we directly obtain them from the used mutation testing tools. Note that when collecting the *typeStatement* and *typeOperator* features, we find that different mutation operators of mutation tools correspond to different statement types perfectly. Thus, we collect only the mutation operation type in this study. For the various static-metric-related features (e.g., *depInheritance* and *numChildren*), we implement our own tool based on the abstract syntax tree (AST) analysis provided by the *Eclipse JDT toolkit*[6]. To obtain the remaining features related to the oracle information (i.e., *numMutantAssertion* and *numClassAssertion*), we develop our own tool based on bytecode analysis using the *ASM bytecode manipulation and analysis framework*[7]. The detailed information about our own tools is disclosed online.
Machine learning framework. We use *Weka*[8], the most widely-used machine learning library, to build and evaluate our classification model.

All our experiments were performed on a platform with 4-core Intel Xeon E5620 CPU (2.4GHz) and 24 Gigabyte RAM running JVM 1.8.0-20 on Ubuntu Linux 12.04.5.

4.2 Subject Systems

To evaluate our approach, we use 9 base projects that have been widely used in software testing research [64, 78, 65] as the base subjects. More specifically, we use the latest versions (i.e., the HEAD commit) of these projects as the base subjects. To facilitate the evaluation of PMT in the cross-version scenario, we prepare multiple versions for each base subject following existing work [64]. That is, we select each version by counting backwards 30 commits at a time from the latest version commit of each project and generate up to 10 versions per project. Note that projects may have less than 10 versions due to the limited version history or execution problems for our mutation testing or feature extraction tools. The information of these base projects is shown in Table 1. In the table, Column "#Ver." lists the number of versions that we used; Column "Size (LOC)" lists the minimum and maximum numbers of lines of executable code[9] for each subject considering its various versions; Column "#Tests" shows the minimum and maximum numbers of manual tests of each version; Column "# All Mutants" shows the minimum and maximum numbers of mutants generated by PIT for each subject; Column "# Killed Mutants" shows the minimum and maximum numbers of mutants killed by tests for different versions of each subject; Column "Time" shows the commit time of the latest version of each subject; finally, Column "Distri" shows the data distribution (i.e., proportion of killed mutants to all mutants) of the latest version of each subject.

Besides, to fully evaluate PMT, we further collect another 154 projects, ranging from 172 to 92,176 lines of code. Note that we started with the first 1000 most popular Java projects from Github in June 2015; 388 of these projects were successfully built with Maven[10] and passed all their

[5] http://cobertura.github.io/cobertura/

[6] http://www.eclipse.org/jdt/

[7] http://asm.ow2.org/

[8] https://weka.wikispaces.com/

[9] Reported by LocMetrics: http://www.locmetrics.com/

[10] https://maven.apache.org/

Table 1: Base subjects

Abbr.	Subjects	#Ver.	Size (LOC)		# Tests		# All Mutants		# Killed Mutants		Time	Distri
			Min.	Max.	Min.	Max.	Min.	Max.	Min.	Max.		
apns	Java APNS	5	666	1,503	65	87	434	1,109	249	412	01/2013	0.46
assj	AssertJ	6	11,326	13,372	4,708	5,229	8,500	10,000	7,228	8,742	02/2014	0.84
joda	Joda-Time	4	29,205	29,702	3,834	4,033	27,707	28,133	20,102	20,882	01/2014	0.64
lafj	Linear Algebra for Java	7	5,810	7,016	245	625	7,062	8,585	3,019	4,987	02/2014	0.67
lang	Apache Commons Lang	7	22,357	25,774	2,309	2,376	26,369	26,411	20,315	21,197	02/2014	0.71
msg	Message Pack for Java	7	8,171	13,481	658	1,145	6,408	11,357	3,407	4,235	06/2012	0.53
uaa	UAA	4	5,691	11,767	223	470	4,515	6,534	2,084	2,965	02/2014	0.25
vrap	Vraptor	3	13,636	14,093	985	1,124	9,259	9,586	6,969	7,189	01/2014	0.69
wire	Wire Mobile Protocol Buffers	4	1,788	2,382	19	61	2,088	2,588	1,528	1,785	01/2014	0.35

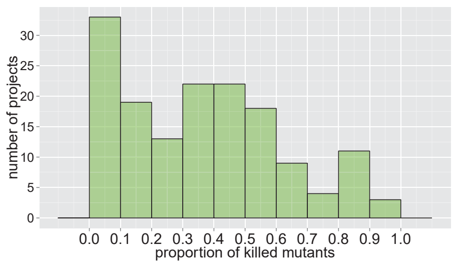

Figure 1: Distribution for 154 Github projects

JUnit tests; then, 227 projects were further removed because they cannot be handled by PIT or the other supporting tools used in our study. Due to space limit, we show only the mutation score distribution of these projects rather than their other information in Figure 1. The x axis represents the range on the proportion of killed mutants. The y axis represents the number of projects belonging to each distribution range. From the figure, the 154 projects have various distribution on the proportion of killed mutants. More specifically, about 66 projects have extremely imbalanced data that the proportion of killed mutants is below 0.2 or above 0.8. More details of all these subjects (e.g., the subject name, version number and the statistics) are available at our project homepage [59].

4.3 Independent Variables

In this study, we consider two independent variables related to the application of PMT, which are mutation testing tools (i.e., PIT and Major) and types of tests (i.e., manually written and automatically generated tests). Besides, we consider another two independent variables related to the implementation of PMT, which are classification algorithms (i.e., *Random Forest*, *Naive Bayes* [57], *SVM* [34], and *J48*[57]) and imbalanced data processing strategies (i.e., the three strategies mentioned in Section 2.3).

4.4 Dependent Variables

As our test sets are sometimes imbalanced, we consider the following common metrics. Note that TP, FP, FN, and TN denote true positive, false positive, false negative, and true negative, respectively.

DV1: ***Precision.*** The fraction of true positive instances in the instances that are predicted to be positive: $TP/(TP+FP)$. Higher precision means fewer false positive errors.

DV2: ***Recall.*** The fraction of true positive instances in the instances that are actual positive: $TP/(TP+FN)$. The higher the recall is, the fewer false negative errors there are.

DV3: ***F-measure.*** The harmonic mean between recall and precision: $2*precision*recall/(precision+recall)$. A high F-measure insures that both precision and recall are reasonably high.

DV4: ***AUC.*** The area of ROC curve [23][11], which measures the overall discrimination ability of a classifier. It is the most popular and widely used measurements in evaluating classification algorithms on imbalanced data [69, 28]. The AUC score for a perfect model would be 1, for a random guessing would be 0.5, and for a model predicting all instances as true or false would be 0.

Note that all these metric values are between 0 and 1[12]. Besides those general metrics for classification algorithms, we also consider the following metric specific to mutation testing since mutation score calculation is an important goal of mutation testing.

DV5: ***Prediction Error.*** The error of the predicted mutation score, which is the difference between the actual mutation score MS and the predicted mutation score MS_p (i.e., $MS - MS_p$)[13]. We also use **absolute prediction error** to refer to the absolute value on prediction error.

4.5 Threats to Validity

The threat to internal validity lies in the implementation of PMT. To reduce this threat, we used the widely used libraries and frameworks (e.g., *Weka*, *ASM*, and *Eclipse JDT*) to aid the implementation and reviewed all our code.

The threats to external validity mainly lie in the subjects and mutants. To reduce the threat resulting from subjects, we selected our base project versions following the same procedure as the prior work [64] and used a large number of Java projects. In the future, we will further reduce this threat by using more programs with more versions in various programming languages (e.g., C, C++, and C#) besides Java. To reduce the threat of mutants, we used both PIT and Major in our study because the former is efficient whereas the latter is proved to be effective in generating representative mutants. In our future, we will further reduce this threat by using more mutation testing tools.

The threat to construct validity lies in the metrics used to measure the classification model. To reduce this threat, we used various widely used measurement metrics for classification models (e.g., *precision, recall, F-measure*, and AUC).

5. RESULT ANALYSIS

5.1 RQ1: Performance of PMT

To answer this RQ, for each application scenario (i.e., cross-version and cross-project), we present the performance

[11]The ROC curve is created by plotting the true positive rate against the false positive rate at various threshold settings.

[12]As PMT predicts both killed and survived mutants, following previous work [21], we use the weighted average of all metrics, each metric is weighted according to the number of instances with the particular class label.

[13]As equivalent mutants are widely recognized to hard to automatically identify [3], similar to prior work [62, 38, 70, 44], we calculate the mutation score for traditional mutation testing and predictive mutation testing by ignoring the influence of equivalent mutants.

Table 2: Cross-version scenario results

Sub.	Ver.	changes	Prec.	Recall	F.	AUC	Err.
lang	v0-v1	49	0.968	0.968	0.968	0.997	1.207%
	v1-v2	120	0.968	0.968	0.968	0.996	1.463%
	v2-v3	13	0.969	0.969	0.968	0.997	1.251%
	v3-v4	14	0.969	0.969	0.968	0.996	1.309%
	v4-v5	171	0.965	0.965	0.965	0.995	1.908%
	v5-v6	6,556	0.963	0.963	0.963	0.994	1.809%
uaa	v0-v1	1,042	0.943	0.940	0.941	0.986	2.797%
	v1-v2	97	0.957	0.956	0.956	0.987	1.580%
	v2-v3	1,162	0.895	0.894	0.894	0.961	0.810%
joda	v0-v1	85	0.915	0.915	0.915	0.962	0.811%
	v1-v2	355	0.956	0.956	0.956	0.993	1.526%
	v2-v3	169	0.937	0.936	0.935	0.983	3.572%
assj	v0-v1	997	0.965	0.964	0.964	0.994	2.469%
	v1-v2	20	0.962	0.962	0.962	0.994	0.212%
	v2-v3	964	0.970	0.970	0.969	0.995	1.808%
	v3-v4	555	0.947	0.948	0.947	0.974	0.704%
	v4-v5	738	0.963	0.963	0.962	0.988	2.662%
msg	v0-v1	240	0.934	0.933	0.933	0.983	2.933%
	v1-v2	519	0.947	0.945	0.945	0.988	2.947%
	v2-v3	4,551	0.953	0.951	0.951	0.987	2.607%
	v3-v4	4,302	0.954	0.953	0.953	0.988	3.165%
	v4-v5	331	0.954	0.954	0.954	0.989	1.207%
	v5-v6	228	0.966	0.966	0.966	0.996	1.211%
apns	v0-v1	193	0.949	0.949	0.949	0.988	2.048%
	v1-v2	362	0.915	0.914	0.914	0.983	2.103%
	v2-v3	0	0.966	0.966	0.966	0.997	0.350%
	v3-v4	67	0.947	0.946	0.946	0.992	1.887%
	v4-v5	215	0.925	0.925	0.925	0.981	1.362%
lafj	v0-v1	994	0.913	0.911	0.911	0.968	2.728%
	v1-v2	270	0.945	0.945	0.945	0.991	1.896%
	v2-v3	190	0.912	0.908	0.908	0.968	5.269%
	v3-v4	938	0.841	0.826	0.829	0.935	-7.484%
	v4-v5	167	0.914	0.913	0.911	0.960	4.502%
	v5-v6	949	0.927	0.926	0.925	0.972	4.333%
vrapt	v0-v1	553	0.959	0.959	0.958	0.991	2.526%
	v1-v2	87	0.914	0.915	0.911	0.950	5.095%
wire	v0-v1	527	0.869	0.870	0.869	0.953	-2.088%
	v1-v2	28	0.971	0.970	0.970	0.997	1.068%
	v2-v3	39	0.960	0.958	0.959	0.991	2.164%

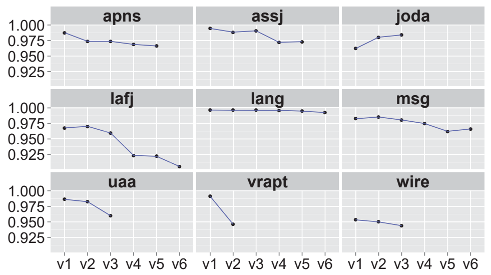

Figure 2: Impacts of version intervals

of PMT using the default configuration (i.e., using the *Random Forest* algorithm and the *naive* imbalanced data processing strategy) in terms of effectiveness and efficiency.

5.1.1 *Effectiveness*

• **Cross-version.** In this scenario, for each of the 9 base projects, we apply PMT to predict the mutation testing results of each version based on the data of the previous versions.

First, for each version, we use its immediate previous version to build the classification model. Thus, for a project with v versions, we perform $v - 1$ predictions – using one version as the training set, and the next version as the test set. The detailed experimental results are shown in Table 2. From the table, PMT performs extremely well under this scenario. More specifically, the absolute values of prediction errors are all below 5% except three predictions, and almost all the other metric values are above 0.9.

Second, to investigate how version intervals impact the performance of PMT, for each project, we use the first version (i.e., v_0) as the training set, and each of the rest versions as the test set. The results are shown in Figure 2, where the x axis represents the versions used as the test set and the y axis represents the AUC values[14]. From the figure,

[14]Due to space limit, we present only the AUC values here. The other metric values are published on our homepage [59].

Table 3: Cross-project scenario results

Sub.	Prec.	Recall	F.	AUC	Err.
apns	0.897	0.884	0.884	0.935	8.719%
assj	0.916	0.919	0.911	0.839	5.641%
joda	0.891	0.877	0.870	0.898	11.168%
lafj	0.888	0.876	0.869	0.876	10.716%
lang	0.904	0.901	0.896	0.896	7.149%
msg	0.919	0.909	0.908	0.924	8.048%
uaa	0.924	0.906	0.910	0.957	7.307%
vrapt	0.900	0.895	0.890	0.882	7.990%
wire	0.858	0.853	0.855	0.919	3.400%

when the version intervals increase, the AUC values decline. This observation is as expected: large version intervals usually lead to large difference between the two versions, which may decrease the effectiveness of PMT. However, AUC values decline in a very low speed and are still all above 0.90. For example, when using v_0 to predict v_3 for `uaa`, the AUC value is still above 0.95 although the project size has evolved from 5,691 to 11,767 and the test suite size has evolved from 223 to 470. This finding indicates that PMT is effective for cross-version prediction even when using the mutation information of an old version far away from the current version in the code repository.

• **Cross-project.** To investigate the effectiveness of PMT in the cross-project scenario, for the latest versions of the 9 base projects, we use PMT to build a classification model based on any 8 base projects to predict the mutation testing results of the remaining base project. Note that when constructing the training data, in case that the data of large projects may overwhelm those of small projects in building the classification model, we assign weights to each instance according to the sizes of projects following the standard way used in machine learning[15]. The results are shown in Table 3. From the table, PMT performs well for all the 9 base projects: all the metric values are above 0.85, and almost all the absolute values of prediction errors of mutation scores are below 10.0%.

In summary, compared to traditional mutation testing, PMT incurs only a small accuracy issue for most of the projects in both cross-version and cross-project scenarios.

5.1.2 *Efficiency*

Next, we present the cost of PMT, and make comparison with traditional mutation testing. As discussed in Section 3, the classification model can be built off-line ahead of time. Thus, the cost of PMT in predicting any new upcoming project contains two parts: feature collection time and prediction time.

Table 4: Performance comparison on PIT

Sub	Time			Error		
	PIT	PMT:cv	PMT:cp	PMT:cv	PMT:cp	Cov.
apns	803s	25s (1s) 32.1X	28s (4s) 28.7X	1.4%	8.7%	21.0%
assj	2667s	41s (1s) 65.0X	43s (3s) 62.0X	2.7%	5.6%	7.3%
joda	3120s	60s (2s) 52.0X	63s (5s) 49.5X	3.6%	11.2%	26.0%
lafj	786s	15s (1s) 52.4X	19s (5s) 41.4X	4.3%	10.7%	14.4%
lang	9540s	63s (2s) 151.4X	66s (5s) 144.5X	1.8%	7.1%	24.2%
msg	4980s	45s (1s) 110.7X	49s (5s) 101.6X	1.2%	8.0%	14.8%
uaa	939s	26s (1s) 36.1X	29s (4s) 32.4X	0.8%	7.3%	17.8%
vrapt	6000s	40s (1s) 150.0X	42s (3s) 142.8X	5.1%	8.0%	16.4%
wire	228s	12s (1s) 19.0X	15s (4s) 15.2X	2.2%	3.4%	23.2%

Table 4 lists the main findings regarding the latest version of each 9 base project. Column 2 lists the execution time by PIT with the default execution setting that the remaining tests will not run against a mutant once the mutant has already been killed. Columns 3 and 4 list the time by

[15]Suppose the total training data contain s instances, while training project A contains a instances, we then set the weight of project A as s/a, thus each project's weight multiplying its number of instances is a constant [2].

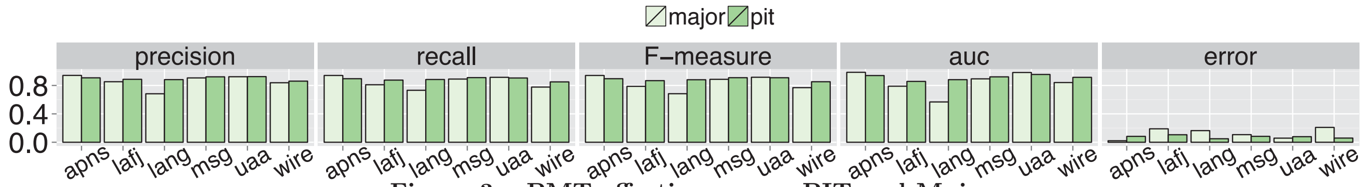

Figure 3: PMT effectiveness on PIT and Major

PMT under two application scenarios (i.e., "cv" represents cross-version, and "cp" represents cross-project[16]), including feature collection and prediction time (i.e., shown in brackets). We also list the speedup that PMT achieves compared with PIT in the right side of each column. From the table, compared with traditional mutation testing using PIT, PMT costs much less time under both scenarios. Especially, PMT costs less than one minute almost for all subjects, while the mutation testing tool PIT costs from 228 seconds to as much as 9,540 seconds (i.e., over two and a half hour). What's more, the time of PIT increases dramatically when the program size increases, while the time of PMT remains almost stable, indicating a much better scalability of PMT. Note that PIT has embodied a number of optimizations and is currently one of the fastest mutation testing tools [11], which indicates that PMT may outperform other mutation testing tools even more. One thing worth mentioning is that there is still much space for further improving the efficiency of PMT. For example, the prediction time is less than or equal to 5 seconds for all projects, and we can further speed up the feature collection time (details shown in Section 6).

5.1.3 *Comparison with Coverage-based Testing*

In addition, as statement coverage has been shown to be effective in predicting test effectiveness in terms of mutation score [19], we directly apply statement coverage to predict mutation scores for the 9 base subjects and record absolute prediction errors in the last column of Table 4. We also show the absolute prediction errors of PMT under cross-version and cross-project scenarios in Columns 5 and 6, respectively. We can find that the errors for both scenarios of PMT are much smaller and more stable than those of coverage-based testing: the largest error for PMT under cross-version/cross-project scenario is only 5.1%/11.2% while it is 26.0% for statement coverage. As an extreme case, for subject `wire`, PMT is able to predict the mutation score with only 2.2%/3.4% error while statement coverage has an error of 23.2%. The reason is that PMT utilizes more detailed coverage information (including the coverage frequency and covering test set) as well as test oracle and static metric information to ensure more accurate mutation testing result prediction. That is, PMT performs much better than coverage-based testing in test effectiveness evaluation.

5.2 RQ2: Application Factors

In this section, we investigate how application factors influence the effectiveness of default PMT, indicating what application artifacts PMT fits for.

5.2.1 *Mutation Testing Tools*

Different mutation testing tools vary in several aspects (e.g., operators, environment support) [11], and may even give different mutation scores for the same project. Therefore, it is important to learn whether PMT is also effective when using mutation tools besides PIT. In this paper, we take the widely used Major as a representative of other mutation tools and evaluate its effectiveness based on the same base projects. As Major fails to generate mutants for `joda`, `vrapt`, and `assj`, we use the remaining 6 base subjects instead, and predict the mutation testing results of each subject using the rest 5 subjects. The results are shown in Figure 3. From the figure, almost all the values of the metrics are above 0.8 for both PMT and Major. Also, there is no obvious difference between the performance of PMT using PIT and Major.

In general, the results above indicate that PMT may achieve good performance on different mutation tools.

5.2.2 *Types of Tests*

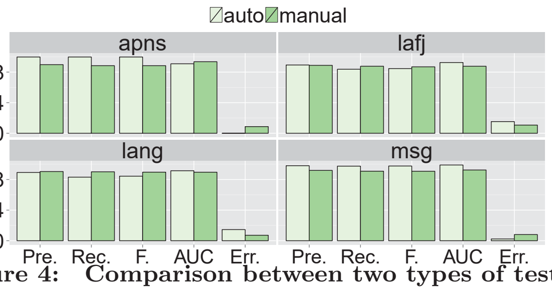

Figure 4: Comparison between two types of tests

We also investigate how PMT performs on predicting the mutation testing results of manually written and automatically generated tests. For each of the 9 base projects, we first generate tests using *randoop*[17], collect related features, and construct a new test set based on those automatically generated tests. Then, we use the other 8 projects with manually written tests to construct the training set, which is used to predict both the new test set (constructed with automatically generated tests) and the original test set (constructed with manually written tests). As *randoop* fails to generate tests for 5 base subjects due to uncompilable generated tests, we compare only the results on the remaining 4 base subjects.

The comparison results are shown by Figure 4. From the figure, PMT performs almost comparably between automatically generated and manually written tests. Manually written tests are a bit inferior. We suspect the reason to be that automatically generated tests are more uniform and less sophisticated (e.g., with standard test body and assertion usage), and thus are easier to predict.

5.3 RQ3: Configuration Impact

In this section, we further extend our experiments with another 154 Github projects to investigate the influence of internal factors of PMT so as to investigate how to make

[16]For the cross-version scenario, the classification model is built from the last nearest version.

[17]http://mernst.github.io/randoop/

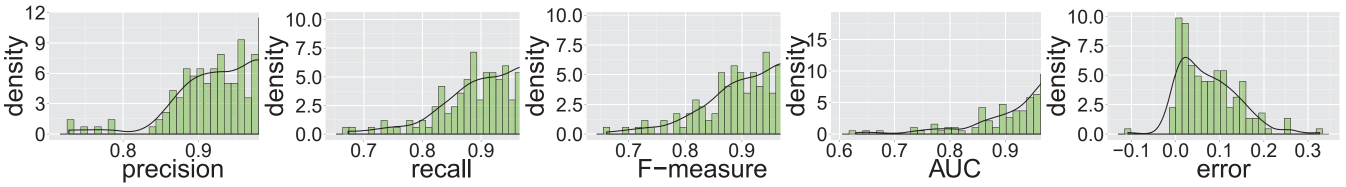

Figure 5: Result distribution under the cross-project scenario

PMT achieve better performance. The latest versions of the 9 base subjects are used to build a classification model, whereas the remaining 154 Github subjects are used to evaluate the classification model. The detailed experimental results are shown in Figure 5. From Figure 5, for the vast majority of the 154 Github projects PMT is able to predict the mutation testing results with over 0.85 precision, recall, and F-measure, and over 0.90 AUC, and is able to predict the mutation scores with errors lower than 15%. This further demonstrates PMT's effectiveness.

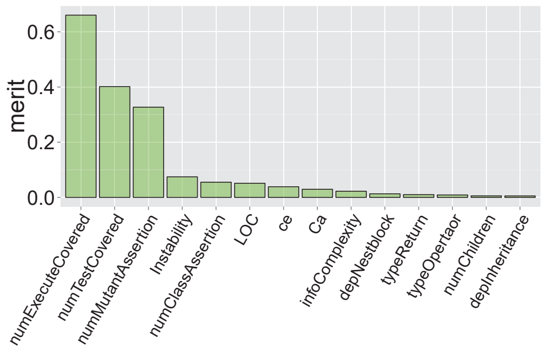

Figure 6: Contribution of features

• **Features.** PMT builds a classification model based on a set of features, which differ in their contribution to classification. In this section, we study the contribution of these features so as to facilitate further improvement of PMT. More specifically, we rank all the features based on their information gain, which measures the worth of features in decision trees [39]. The higher information gain a feature has, the more preferred it is to other features. More specifically, for a feature, we used *InfoGainAttributeEval*, the most popular way for feature ranking in *Weka*, to get the information gain of a dataset constructed by all our projects. The detailed results are shown in Figure 6, where the x axis lists all the features used in our work and the y axis lists all merit values for each feature. According to the figure, the coverage features (i.e., *numExecuteCovered* and *numTestCovered*) are the most important features in predicting mutant execution results. This observation is as expected because coverage is a necessary condition for mutant killing according to the PIE theory [68]. The oracle-related features, *numMutantAssertion* and *numClassAssertion* are also essential, since they show detailed information about how many assertions may have checked a mutant. Furthermore, the *instability* feature, which describes how the mutated class interacts with outside classes, also ranks high, since mutated classes with more dependents may have more tunnels to propagate the internal state change to final output.

• **Classification algorithms.** We first investigate whether PMT predicts better than random guess. We compare the prediction results of the default PMT classification algorithm (i.e., *Random Forest*) with those of the baseline model in machine learning (i.e., *ZeroR* [43]) on our 154 Github projects. The results are shown in Figure 7, where the x axis represents the 154 Github projects, bars represent the metric values of PMT, and black points represent the metric values of random guess. From the first three subfigures, most of the black points are much lower than bars, indicating that PMT is much better than random guess in precision, recall, and F-meausre; from the last subfigure, random guess always has a AUC value of 0.5, while PMT is much better than that. Note that we did not construct the training set intentionally, but still got impressive prediction results, indicating the effectiveness of PMT.

We then investigate the effectiveness of PMT by varying its classification algorithms, including *Random Forest*, *Naive Bayes* [46], *SVM* [34], and *J48* [57]. The comparison results are shown by Figure 8, where the x axis represents all the four algorithms and the y axis represents different metric values. The width of each metric value represents the density of this value (i.e., larger density represents that more projects have this value) among all the 154 projects. From the figure, *Random Forest* clearly performs the best; *J48* is slightly inferior to *Random Forest*; *SVM* performs worse than *J48*; *Naive Bayes* performs the worst. As both *Random Forest* and *J48* belong to decision tree algorithms, we suspect that as different features in PMT obviously contribute differently (e.g., execution features contribute most), decision tree algorithms have the advantage by putting these key features on the top of decision trees, and thus will get definite decision solutions based on those key feature conditions [60], which is different from other algorithms.

• **Imbalanced data processing strategies.** To learn how PMT performs on very imbalanced test sets, we choose the 66 Github subjects whose proportion of killed mutants are below 0.2 or above 0.8 as representatives of imbalanced test sets. The AUC results of PMT are given by Figure 10. From the figure, the default *naive* imbalanced data processing strategy of PMT already performs well even for those very imbalanced projects. We further compare all the three imbalanced data processing strategies mentioned in Section 2.3 to deal with the training data, whose results are shown by Figure 9, where the x axis represents the imbalanced data processing strategies and the y axis represents different metric values. From the figure, there is little difference among the three strategies. This indicates that PMT is robust to imbalanced data, as the *Random Forest* algorithm handles imbalanced data well [41].

6. DISCUSSION

6.1 Implementation

Feature optimization may be needed to refine the effectiveness or reduce PMT's cost. In this paper, we use 14 features. More effective features (e.g., semantic information based on advanced control/data flow analysis) may be considered to make a more effective classification model. Also, we choose features without considering their overlap. Fea-

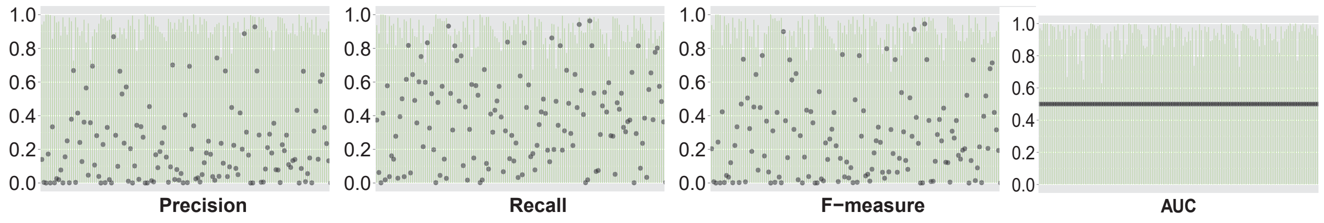

Figure 7: Comparison with random guess

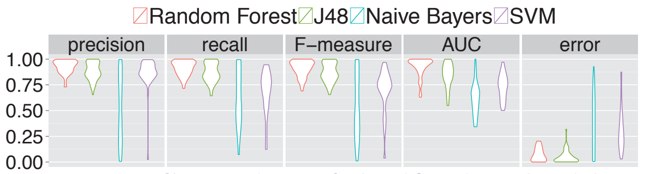

Figure 8: Comparison of classification algorithms

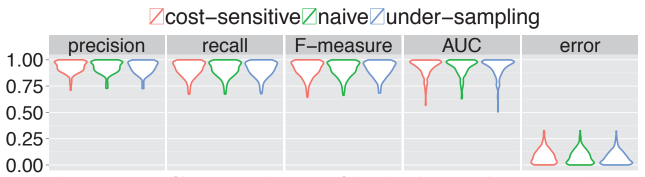

Figure 9: Comparison of imbalanced strategies

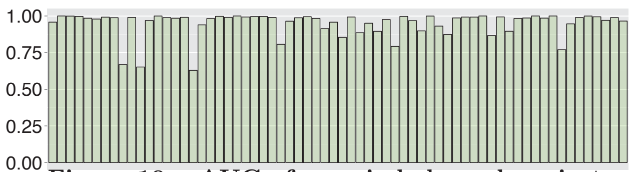

Figure 10: AUC of very imbalanced projects

ture selection techniques can be applied, and thus developers may use fewer features to achieve the same effectiveness. Furthermore, the current feature collection of PMT can also be sped up to make the process faster, e.g., the current static metric collection is based on the costly Eclipse JDT analysis and we plan to reimplement that based on bytecode analysis.

Training set size is another internal factor of PMT, which may influence its effectiveness. Intuitively, a bigger training set may improve the effectiveness of PMT by sacrificing some efficiency on classification model construction, which actually occurs before prediction and can be constructed offline. However, from Table 3, the prediction results of PMT on a training set constructed by only 8 projects are already satisfactory. The reason is that even a single training project can provide a tremendous number of training instances for PMT. For example, the `lang` project alone already contains more than 26,000 training instances. In the future, we will further investigate the influence of training set size, as well as the ways to construct more effective training sets that may produce better classification models.

Note that currently PMT is implemented for Java programs with JUnit tests. Although the implementation frameworks and the detailed feature sets may vary across different languages (e.g., the *numMutantAssertion* feature may not be available for C programs since they usually do not have test code with assertions), the basic idea can be applied to programs written in different languages with different test paradigms, and we plan to explore that in the near future.

6.2 Tradeoff: Efficiency v.s. Effectiveness

Although PMT is proposed to address the efficiency problem in mutation testing, it suffers from a new accuracy concern. Fortunately, PMT makes a good tradeoff between efficiency and accuracy because it improves the efficiency of mutation testing largely (i.e., 15.2X-151.4X) while only incurring a small accuracy issue (e.g., over 0.85 AUC in most cases). Therefore, for projects that traditional mutation testing is unbearable long, PMT is typically useful and applicable. For such projects, developers may tend to give up mutation testing in the past, but now can use PMT as an alternative. Furthermore, if developers are urgent to obtain the mutation execution results and allow some accuracy loss, they may tend to use PMT rather than traditional mutation testing. Besides, among all the applications of mutation testing, some may care much more about efficiency rather than accuracy loss, and thus may be in more need of such a tradeoff provided by PMT. For example, when using mutation testing to filter extremely large test suites generated by test generation tools, PMT may be a good choice, since it can help choose high-quality test suites within minutes.

In the future, we will further investigate each mutant's class probability distribution[18] provided by classifiers, with which developers may choose the mutants with proper probability distribution to get better prediction results. We will also investigate how PMT performs on different types of mutants, with which PMT can provide more accurate and find-grained prediction results.

6.3 Impacts in Software Testing

From the microscopic view, PMT can be viewed as an improvement over traditional mutation testing. In particular, the machine-learning based process in PMT facilitates assessment of mutation testing results so that it is not necessary to obtain mutation testing results through mutant execution. Thus, predictive mutation testing is much more light-weight than traditional mutation testing, and still has accurate results. From the macroscopic view, PMT may be viewed as a new measurement for test suite quality. That is, the predicted mutation score based on PMT can be directly used to evaluate the test suite quality rather than being treated as a replacement of the traditional mutation testing score. Compared with the widely used coverage criterion, the predicted mutation score is more effective because it characterizes the execution, infection, and propagation of faults whereas coverage criterion characterizes only the execution (confirmed by experimental results in Section 5.2.1). Moreover, PMT does not incur much extra cost comparing with the coverage criterion. Compared with the traditional mutation score, the predicted mutation score is more lightweight because it does not require the costly mutant execution.

[18]The confidence level at which this mutant is classified correctly [7].

7. RELATED WORK

7.1 Mutation Testing

Mutation testing is a powerful methodology to evaluate the effectiveness of test suites. It was first proposed by Demillo et al. [14] and Hamlet [22], and is gaining more and more popularity [16, 12, 18]. Despite its effectiveness, mutation testing has a main limitation, i.e., it is extremely expensive since it needs to execute the test suite on each mutant. Therefore, many researchers have focused on presenting various techniques to reduce the cost of mutation testing. Some techniques focus on reducing the number of mutants. In particular, Mathur and Wong [71] analyzed 22 mutation operators used by Mothra [10] and observed that several operators contribute to most of the generated mutants. Based on this, Offutt et al. [54] proposed *N-selective mutation*, which reduces the number of mutants by omitting the N most prevalent operators. Following their work, many researchers focused on detecting *sufficient mutation operators* [72, 5, 54, 66, 33, 18]. Zhang et al. [79] conducted an empirical study on comparing random mutant selection and operator-based mutant selection, and found that random mutant selection is as effective as operator-based mutation selection. Other techniques focus on reducing the mutant execution time. Howden [27] proposed the concept of *weak mutation testing*, which executes a mutant partially to speed up mutant execution. Later, Woodward and Halewood [73] proposed *firm mutation testing*, which is a compromise of weak mutation testing and traditional mutation testing. Offutt and Lee [52] conducted an empirical study on weak mutation testing and found that weak mutation testing can be better applied in unit testing of non-critical applications. Emillo et al. [13] and Untch et al. [67] changed a compiler to make it able to compile all mutants at one time. Krauser et al. [40] and Offutt et al. [53] ran mutants in parallel to speed up mutation testing. Just et al. [37] found that redundant mutants would affect mutation score as well as increasing the cost. Zhang et.al. [77] found that selective mutation testing has good scalability, and the proportion of selected mutants can be predicted. To facilitate mutation testing for evolving programs, Zhang et al. [81] proposed to speed up mutation testing for the current program by reusing the execution results of some mutants on the previous program.

Although the existing techniques can speed up mutation testing, they still need to execute the mutants to get each mutant's execution result and are still costly. In contrast, PMT opens a new dimension in mutation testing which does not require mutant execution, and has been shown to be more efficient than state-of-the-art techniques that embody various existing optimizations. Note that Jalbert [31] also applied machine learning to mutation testing, while their technique only classifies the mutation score of a source code unit into three categories: low, medium, and high.

7.2 Coverage-based Testing

Besides mutation testing, code coverage is another widely used methodology for measuring test suite effectiveness in both academia and industry [61, 15, 20, 17, 19]. A huge body of research has been dedicated to study the relationship between test coverage and test effectiveness. Namin and Andrews [51] reported that block coverage, decision coverage, and other two data-flow criteria can influence test suite effectiveness. Gligoric et al. [17] found that branch coverage correlates well with test effectiveness. Later on, Gopinath et al. [19] performed a larger scale empirical study, and observed that statement coverage correlates the best with test effectiveness. However, recently, more and more people realized that code coverage may not have high correlation with test suite effectiveness. Inozemtseva and Holmes [30] conducted several studies to evaluate such correlation, and found the correlation to be weak. They suggest that code coverage should not be used as a quality target. Later on, Zhang and Mesbah [84] further found that one possible reason for the low correlation is that code coverage usually does not consider assertion information. In industry, many developers also start to doubt test coverage. In fact, though code coverage is widely used, it does not consider the oracle information at all, while a test suite could be useless if it has no oracle, even if it achieves 100% coverage. Considering test oracle information, Schuler and Zeller [63] proposed checked coverage (i.e., the proportion of statements dynamically checked by test assertions) to indicate test effectiveness. Although powerful at measuring test effectiveness in detecting faults captured by assertions, checked coverage cannot precisely measure test effectiveness in detecting faults captured by other exceptions. In the near future, we plan to include checked coverage as one feature in PMT.

PMT costs comparatively with code coverage, but has been demonstrated to be effective in measuring test effectiveness (also more powerful than the most effective statement coverage identified by Gopinath et al. [19] recently).

8. CONCLUSION

In this paper, we propose *predictive mutation testing*, the first approach that predicts mutation testing results without any mutant execution. We have implemented the proposed approach using the *Random Forest* algorithm. The experimental results on 163 real-world Java projects demonstrate that PMT can predict mutant execution results accurately. The comparison with traditional testing methodologies also shows that PMT is able to predict mutation testing results accurately with small overhead, demonstrating a good tradeoff between efficiency and effectiveness of mutation testing.

9. ACKNOWLEDGMENTS

This work is supported by the National 973 Program of China No.2015CB352201, the National Natural Science Foundation of China under Grants No.61421091, 61225007, 61522201, 61272157, and 61529201. The authors from UT Dallas are supported in part by NSF grant CCF-1566589, UT Dallas start-up fund, and Google Faculty Research Award.

10. REFERENCES

[1] http://www.codeaffine.com/2015/10/05/what-the-heck-is-mutation-testing/.

[2] http://nestor.coventry.ac.uk/~nhunt/meths/strati.html.

[3] K. Adamopoulos, M. Harman, and R. M. Hierons. How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution. In *Proc. GECCO*, pages 1338–1349, 2004.

[4] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proc. ICSE*, pages 402–411, 2005.

[5] E. F. Barbosa, J. C. Maldonado, and A. M. R. Vincenzi. Toward the determination of sufficient mutant operators for C. *STVR*, 11(2):113–136, 2001.

[6] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[7] J. S. Bridle. Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition. In *Neurocomputing*, pages 227–236. Springer, 1990.

[8] A. Brillout, N. He, M. Mazzucchi, D. Kroening, M. Purandare, P. Rümmer, and G. Weissenbacher. Mutation-based test case generation for Simulink models. In *Proc. FMCO*, pages 208–227, 2010.

[9] Y. Brun and M. D. Ernst. Finding latent code errors via machine learning over program executions. In *Proc. ICSE*, pages 480–490, 2004.

[10] B. Choi, R. A. DeMillo, E. W. Krauser, R. Martin, A. Mathur, A. J. Offutt, H. Pan, and E. H. Spafford. The mothra tool set (software testing). In *Proc. ICSS*, pages 275–284, 1989.

[11] M. Delahaye and L. du Bousquet. A comparison of mutation analysis tools for Java. In *Proc. QSIC*, pages 187–195, 2013.

[12] M. Delamaro, M. Pezzè, A. M. R. Vincenzi, and J. C. Maldonado. Mutant operators for testing concurrent java programs. In *Proc. SBES*, pages 272–285, 2001.

[13] R. A. DeMillo, E. W. Krauser, and A. P. Mathur. Compiler-integrated program mutation. In *Proc. COMPSAC*, pages 351–356, 1991.

[14] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

[15] S. Fine and A. Ziv. Coverage directed test generation for functional verification using bayesian networks. In *Proc. DAS*, pages 286–291, 2003.

[16] G. Fraser and A. Arcuri. Achieving scalable mutation-based generation of whole test suites. *Empirical Software Engineering*, pages 1–30, 2014.

[17] M. Gligoric, A. Groce, C. Zhang, R. Sharma, M. A. Alipour, and D. Marinov. Comparing non-adequate test suites using coverage criteria. In *Proc. ISSTA*, pages 302–313, 2013.

[18] M. Gligoric, L. Zhang, C. Pereira, and G. Pokam. Selective mutation testing for concurrent code. In *Proc. ISSTA*, pages 224–234, 2013.

[19] R. Gopinath, C. Jensen, and A. Groce. Code coverage for suite evaluation by developers. In *Proc. ICSE*, pages 72–82, 2014.

[20] R. Gupta, A. P. Mathur, and M. L. Soffa. Generating test data for branch coverage. In *Proc. ASE*, pages 219–227, 2000.

[21] T. Gyimothy, R. Ferenc, and I. Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *TSE*, 31(10):897–910, 2005.

[22] R. G. Hamlet. Testing programs with the aid of a compiler. *TSE*, (4):279–290, 1977.

[23] J. A. Hanley and B. J. McNeil. The meaning and use of the area under a receiver operating characteristic (ROC) curve. *Radiology*, 143(1):29–36, 1982.

[24] D. Hao, L. Zhang, M.-H. Liu, H. Li, and J.-S. Sun. Test-data generation guided by static defect detection. *JCST*, 24(2):284–293, 2009.

[25] M. Harman, Y. Jia, and W. B. Langdon. Strong higher order mutation-based test data generation. In *Proc. FSE*, pages 212–222, 2011.

[26] M. Harman, Y. Jia, P. Reales Mateo, and M. Polo. Angels and monsters: An empirical investigation of potential test effectiveness and efficiency improvement from strongly subsuming higher order mutation. In *Proc. ASE*, pages 397–408, 2014.

[27] W. E. Howden. Weak mutation testing and completeness of test sets. *TSE*, (4):371–379, 1982.

[28] J. Huang and C. X. Ling. Using AUC and accuracy in evaluating learning algorithms. *TKDE*, 17(3):299–310, 2005.

[29] L. Inozemtseva, H. Hemmati, and R. Holmes. Using fault history to improve mutation reduction. In *Proc. FSE*, pages 639–642, 2013.

[30] L. Inozemtseva and R. Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proc. ICSE*, pages 435–445, 2014.

[31] K. Jalbert and J. S. Bradbury. Predicting mutation score using source code and test suite metrics. In *Proc. RAISE'*, pages 42–46, 2012.

[32] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *TSE*, 37(5):649–678, 2011.

[33] Y. Jiang, S.-S. Hou, J. Shan, L. Zhang, and B. Xie. An approach to testing black-box components using contract-based mutation. *ISSRE*, pages 93–117, 2008.

[34] T. Joachims. Advances in kernel methods. chapter Making Large-scale Support Vector Machine Learning Practical, pages 169–184. MIT Press, 1999.

[35] R. Just, M. D. Ernst, and G. Fraser. Efficient mutation analysis by propagating and partitioning infected execution states. In *Proc. ISSTA*, pages 315–326, 2014.

[36] 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 *Proc. FSE*, pages 654–665, 2014.

[37] R. Just, G. M. Kapfhammer, and F. Schweiggert. Do redundant mutants affect the effectiveness and efficiency of mutation analysis? In *Proc. ICST*, pages 720–725. IEEE, 2012.

[38] R. Just, F. Schweiggert, and G. M. Kapfhammer. Major: An efficient and extensible tool for mutation analysis in a java compiler. In *Proc. ASE*, pages 612–615, 2011.

[39] J. T. Kent. Information gain and a general measure of correlation. *Biometrika*, 70(1):163–173, 1983.

[40] E. W. Krauser, A. P. Mathur, and V. J. Rego. High performance software testing on simd machines. *TSE*, 17(5):403–423, 1991.

[41] M. Liu, M. Wang, J. Wang, and D. Li. Comparison of random forest, support vector machine and back propagation neural network for electronic tongue data classification. *SABC*, 177:970–980, 2013.

[42] Y. Lou, D. Hao, and L. Zhang. Mutation-based test-case prioritization in software evolution. In *Proc. ISSRE*, pages 46–57, 2015.

[43] L. Lu, H. Jiang, and H. Zhang. A robust audio

classification and segmentation method. In *Proc. ACMMM*, pages 203–211. ACM, 2001.

[44] L. Madeyski. The impact of test-first programming on branch coverage and mutation score indicator of unit tests: An experiment. *IST*, 52(2):169–184, 2010.

[45] T. J. McCabe. A complexity measure. *TSE*, (4):308–320, 1976.

[46] A. McCallum, K. Nigam, et al. A comparison of event models for naive bayes text classification. In *AAAI*.

[47] D. Michie, D. J. Spiegelhalter, and C. C. Taylor. Machine learning, neural and statistical classification. 1994.

[48] S. Moon, Y. Kim, M. Kim, and S. Yoo. Ask the mutants: Mutating faulty programs for fault localization. In *Proc. ICST*, pages 153–162, 2014.

[49] J.-M. Mottu, B. Baudry, and Y. Le Traon. Mutation analysis testing for model transformations. In *Proc. ECMDA*, pages 376–390. Springer, 2006.

[50] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin. Convolutional neural networks over tree structures for programming language processing. In *AAAI*, 2016.

[51] A. S. Namin and J. H. Andrews. The influence of size and coverage on test suite effectiveness. In *Proc. ISSTA*, pages 57–68, 2009.

[52] A. J. Offutt and S. D. Lee. An empirical evaluation of weak mutation. *TSE*, 20(5):337–344, 1994.

[53] A. J. Offutt, R. P. Pargas, S. V. Fichter, and P. K. Khambekar. Mutation testing of software using a mimd computer. In *Proc. ICPP*, 1992.

[54] A. J. Offutt, G. Rothermel, and C. Zapf. An experimental evaluation of selective mutation. In *Proc. ICSE*, pages 100–107, 1993.

[55] M. Papadakis and Y. Le Traon. Using mutants to locate "unknown" faults. In *Proc. ICSTW*, pages 691–700, 2012.

[56] M. Papadakis, N. Malevris, and M. Kallia. Towards automating the generation of mutation tests. In *Proc. AST*, pages 111–118, 2010.

[57] T. R. Patil and S. Sherekar. Performance analysis of naive bayes and J48 classification algorithm for data classification. *IJCSA*, 6(2):256–261, 2013.

[58] H. Peng, L. Mou, G. Li, Y. Liu, L. Zhang, and Z. Jin. Building program vector representations for deep learning. In *Knowledge Science, Engineering and Management*, pages 547–553. 2015.

[59] PMT homepage. https://github.com/SEITest/PMT.

[60] J. R. Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[61] S. Rayadurgam and M. P. E. Heimdahl. Coverage based test-case generation using model checkers. In *Proc. ECBS*, pages 83–91, 2001.

[62] D. Schuler and A. Zeller. Javalanche: efficient mutation testing for java. In *FSE*, pages 297–298, 2009.

[63] D. Schuler and A. Zeller. Assessing oracle quality with checked coverage. In *Proc. ICST*, pages 90–99, 2011.

[64] A. Shi, A. Gyori, M. Gligoric, A. Zaytsev, and D. Marinov. Balancing trade-offs in test-suite reduction. In *Proc. FSE*, pages 246–256, 2014.

[65] A. Shi, T. Yung, A. Gyori, and D. Marinov. Comparing and combining test-suite reduction and regression test selection. In *FSE*, pages 237–247, 2015.

[66] A. Siami Namin, J. H. Andrews, and D. J. Murdoch. Sufficient mutation operators for measuring test effectiveness. In *Proc. ICSE*, pages 351–360, 2008.

[67] R. H. Untch, A. J. Offutt, and M. J. Harrold. Mutation analysis using mutant schemata. In *Proc. ISSTA*, pages 139–148, 1993.

[68] J. M. Voas. Pie: A dynamic failure-based technique. *TSE*, 18(8):717–727, 1992.

[69] C. G. Weng and J. Poon. A new evaluation measure for imbalanced datasets. In *Proc. AusDM*, pages 27–32, 2008.

[70] W. E. Wong, editor. *Mutation Testing for the New Century*. Kluwer Academic Publishers, 2001.

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

[72] W. E. Wong, A. P. Mathur, and J. C. Maldonado. Mutation versus all-uses: An empirical evaluation of cost, strength and effectiveness. In *Software Quality and Productivity: Theory, Practice and Training*, pages 258–265, 1995.

[73] M. Woodward and K. Halewood. From weak to strong, dead or alive? an analysis of some mutation testing issues. In *Proc. STVA*, pages 152–158, 1988.

[74] J. Xuan, X. Xie, and M. Monperrus. Crash reproduction via test case mutation: Let existing test cases help. In *Proc. FSE*, pages 910–913, 2015.

[75] J. Zhang, J. Chen, D. Hao, Y. Xiong, B. Xie, L. Zhang, and H. Mei. Search-based inference of polynomial metamorphic relations. In *Proc. ASE*, pages 701–712, 2014.

[76] J. Zhang, X. Wang, D. Hao, B. Xie, L. Zhang, and H. Mei. A survey on bug-report analysis. *Science China Information Sciences*, 58(2):1–24, 2015.

[77] J. Zhang, M. Zhu, D. Hao, and L. Zhang. An empirical study on the scalability of selective mutation testing. In *Proc. ISSRE*, pages 277–287. IEEE, 2014.

[78] L. Zhang, M. Gligoric, D. Marinov, and S. Khurshid. Operator-based and random mutant selection: Better together. In *Proc. ASE*, pages 92–102, 2013.

[79] L. Zhang, S.-S. Hou, J.-J. Hu, T. Xie, and H. Mei. Is operator-based mutant selection superior to random mutant selection? In *Proc. ICSE*, pages 435–444, 2010.

[80] L. Zhang, D. Marinov, and S. Khurshid. Faster mutation testing inspired by test prioritization and reduction. In *Proc. ISSTA*, pages 235–245, 2013.

[81] L. Zhang, D. Marinov, L. Zhang, and S. Khurshid. Regression mutation testing. In *Proc. ISSTA*, pages 331–341, 2012.

[82] L. Zhang, T. Xie, L. Zhang, N. Tillmann, J. De Halleux, and H. Mei. Test generation via dynamic symbolic execution for mutation testing. In *Proc. ICSM*, pages 1–10, 2010.

[83] L. Zhang, L. Zhang, and S. Khurshid. Injecting mechanical faults to localize developer faults for evolving software. In *OOPSLA*, pages 765–784, 2013.

[84] Y. Zhang and A. Mesbah. Assertions are strongly correlated with test suite effectiveness. In *Proc. FSE*, pages 214–224, 2015.