An Extensive Study on Cross-project Predictive Mutation Testing

Dongyu Mao
Department of Computer Science
The University of Texas at Dallas
maody@utdallas.edu

Lingchao Chen
Department of Computer Science
The University of Texas at Dallas
lxc170330@utdallas.edu

Lingming Zhang
Department of Computer Science
The University of Texas at Dallas
lingming.zhang@utdallas.edu

***Abstract*—Mutation testing is a powerful technique for evaluating the quality of test suite which plays a key role in ensuring software quality. The concept of mutation testing has also been widely used in other software engineering studies, e.g., test generation, fault localization, and program repair. During the process of mutation testing, large number of mutants may be generated and then executed against the test suite to examine whether they can be killed, making the process extremely computational expensive. Several techniques have been proposed to speed up this process, including selective, weakened, and predictive mutation testing. Among those techniques, Predictive Mutation Testing (PMT) tries to build a classification model based on an amount of mutant execution records to predict whether coming new mutants would be killed or alive without mutant execution, and can achieve significant mutation cost reduction. In PMT, each mutant is represented as a list of features related to the mutant itself and the test suite, transforming the mutation testing problem to a binary classification problem. In this paper, we perform an extensive study on the effectiveness and efficiency of the promising PMT technique under the cross-project setting using a total 654 real world projects with more than 4 Million mutants. Our work also complements the original PMT work by considering more features and the powerful deep learning models. The experimental results show an average of over 0.85 prediction accuracy on 654 projects using cross validation, demonstrating the effectiveness of PMT. Meanwhile, a clear speed up is also observed with an average of 28.7X compared to traditional mutation testing with 5 threads. In addition, we analyze the importance of different groups of features in classification model, which provides important implications for the future research.**

***Index Terms*—software quality, software testing, mutation testing, machine learning, deep learning**

I. Introduction

Mutation testing [1], [2] is a powerful technique for evaluating test suite quality. In the process of mutation testing, numbers of mutants will be generated. Every *mutant* is a program variant that is generated from the original program based on a set of transformation rules which are called *mutation operators*. Then those mutants (program variants) will be executed against the test suite associated with the original program to see if the test suite can kill them. A mutant is said to be *killed* if there exist any test in the test suite showing a different execution result between the mutant and the original program; otherwise, the mutant is said to be *alive*, or *survived*. By calculating the ratio between killed and non-equivalent (by *equivalent*, it means that the mutant is semantically equivalent to the original program, which can never be killed) mutants, we obtain a metric called *mutation score*, which is widely used to evaluate the powerfulness of the test suite.

Mutation testing is considered as a powerful technique in test suite quality evaluation [3], [4] and is gaining more and more attentions in both software engineering research [5]–[8] and real-world applications [9]–[11]. Besides the directly usage in evaluation of test suite quality, mutation testing has also shown it powerfulness in other testing and debugging problems, such as real fault simulation [12]–[14], fault localization [15]–[18], test generation [19]–[22], and program repair [23]–[29].

One of the main limitations of mutation testing resides in its computational expenses [5], [30], as usually large numbers of mutants are generated for the original program and then executed against the test suite. Both the generation and execution of mutants can be costly. The mutant generation cost has been reduced by various techniques [31], [32] to an acceptable level. While for mutant execution, researchers have proposed many techniques to reduce its cost, e.g., selective mutation testing [33]–[36], weak mutation testing [37], high-order mutation testing [38], optimized mutation testing [39], [40], etc. However, the cost of running mutants is still extremely high despite of those refinement techniques [5].

Recently, an interesting technique called *predictive mutation testing* (PMT) [41] was proposed to try to "*obtain mutation testing results without mutant execution*". The key idea of PMT is to train a classification model based on an amount of mutant execution records to predict whether coming new mutants would be killed or alive without execution. In PMT, each mutant is represented as a list of features related to the mutant itself and the test suite, and the prediction phase is usually very fast, making it a very efficient approach. The experiment results in [41], [42] showed that the PMT framework is both effective and efficient under both *cross-version* and *cross-project* settings. However, the prior work only evaluated PMT with a limited set of features and classification algorithms on a limited number of projects.

The key part of the PMT is *feature design*, which is a crucial step in most machine learning tasks [43]. In the original PMT work [41], a list of 14 features are identified based on PIE theory [44]. In PIE theory, a mutant can be killed if all following three conditions are satisfied: 1) ***Execution***, the mutated statement is executed by the test; 2) ***Infection***, the program state changes after the execution of the mutated statement; 3) ***Propagation***, the infected program state propagates to test output causing the output to be different from original program. Then for each condition, a list of features is identified or designed based on the analysis of the possible relation of each feature to the condition definition. Those features can be further categorized into two groups, i.e., static and dynamic features. Static features are selected from list of software metrics, while dynamic features are manually designed based on the execution traces of test suite. The main limitation of the prior PMT implementation in [41], [42] lies in the feature design part. Although it seems reasonable to manually inspect the definition of software metrics and then relate them to one of the PIE theory aspect to make feature selection, there is no guarantee that when using other software metrics, PMT will perform worse. In fact, there are many other software metrics in different granularity, i.e., package, class, or method level metrics, that may *relate* to whether a mutant can be killed.

In this paper, we enrich the PMT feature list by adding more static software metrics in different granularity. Especially, *JHawk* tool[1] is used to extract a total of 89 static software metrics from package, class, and method level. By combining those static metrics with 4 dynamic features and mutator type that also used in [41], a total 95 features are considered in our evaluation. Using dataset with the larger feature list, individual feature's importance is calculated based on *Random Forest* [45] classification model, which is shown to perform the best in the original PMT work [41], and then those importance scores are further used to perform feature selection to try to improve PMT performance. Also, inspired by the recent successes of deep learning in software engineering studies [46]–[50], 11 popular machine learning models, including three widely used deep learning models such as *Multi-Layer Perceptron* [51], *Convolutional Neural Network* (CNN) [52], and *Cascading Forest* (caForest) [53], are studied in PMT.

In this work, we focus on evaluating PMT under cross-project setting because the performance of PMT in cross-version setting has been shown to be quite good [41] and cross-version setting can be viewed as an easier instance of the cross-project setting if we take different versions as different projects; furthermore, the cross-version setting requires mutation testing results on earlier versions which may not always be available in practice. In summary, a total of 654 Java projects with over 4M mutants are used in this study. Different from [41] where only 9 base projects are used in the training phase, a more rigorous 5-fold cross validation is used to evaluate the effectiveness of PMT.

Our study has found various interesting findings that may further advance future PMT, such as:

- PMT performs well on extensive cross-project prediction, achieving a mean error rate of around 0.15 with a 28.7X mean speedup compared with traditional mutation testing using 5 threads (our study also demonstrates a theoretically best mean speedup of 51.7X).
- The simple *Random Forest* algorithm achieves nearly the same effectiveness for PMT compared to the advanced deep learning models such as *CNN*, and *caForest*.
- Larger feature sets benefits deep learning more than traditional learning algorithms for PMT.
- Dynamic features dominate the prediction process and cause high recalls (near 1.0) for most subjects.
- Package level static software metrics are more important than class and method level metrics for PMT and deserve more investigations.

To sum up, this paper makes the following contributions:

- An extensive study of PMT on 654 real world Java projects with more than 4 Million mutant records under cross-project setting.
- A total of 11 popular classification algorithms including 3 state-of-the-art deep learning ones are evaluated in PMT framework using various metrics related to prediction quality and time under their preferred data and model settings.
- A large list of features' contributions in the classification model are analyzed, which provide important guidance to refine PMT implementation in the future.

The rest of the paper is organized as follows. Section II introduces the details of the studied PMT approach. Section III introduces experimental setup. Section IV presents the experimental results and analysis. Related work will be introduced in section V and section VI concludes this paper.

II. Studied Approach

Mutation testing is costly because usually large number of mutants need to be executed against the test suite [5]. In predictive mutation testing (PMT) framework, the execution of mutants is replaced by a binary classification process using a pre-trained classifier. As the feature collection phase and prediction phase usually cost much less time than running the whole mutants, PMT can achieve high efficiency.

In this section, we introduce the details of our PMT implementation, especially:

- The PMT framework (see Section II-A).
- What features are used to build the classification models: We introduce each type of features used in our implementation in details (see Section II-B).
- What classification algorithms are selected for evaluation: We give a brief introduction to three of them that show good performance (see Section II-C).

A. PMT Framework

The general framework of PMT is shown in Fig. 1. In mutation testing, a list of mutation operators is applied to different locations of program source/byte code to generate different mutants, those mutants are then executed against the test suite associated with the original program to check whether they can be killed or not. While in PMT, we are

[1]http://www.virtualmachinery.com/jhawkprod.htm

interested in predicting the mutant execution results without the execution process.

As shown in Fig. 1, PMT has two main stages:

1) Feature Extraction. In this stage, a list of features from different categories (e.g., static and dynamic features) are extracted for each mutant. Mutation operation location, mutator type, and tests execution traces are needed to prepare those features.
2) Training or Prediction. This stage has two working modes, training and prediction. In the training working mode, classifier is trained with a dataset containing historical mutant execution records, each record is a list of feature values extracted using the former stage with a label (killed or alive) associated with this record. In the prediction working mode, the classifier is used to predict whether a mutant can be killed or not based on the feature value list associated with this mutant.

As extracting more useful features may boost machine learning approach significantly [43], feature extraction is the key stage in PMT. While considering adding more useful features to help improve effectiveness, we also need to be careful to avoid making feature extraction consume too much time and incur efficiency issue.

B. Feature Extraction

In mutation testing, each mutation operation applies to one location of program source/byte code and generates a corresponding program mutant using specific mutation operator (mutator). For example, "●" and "▲" are used to represent some code elements as shown in Fig. 1, the mutation operation generates a mutant by replacing "●" with "▲" in the source/byte code in one location. Therefore, features for each mutant can be designed and extracted based on the mutation operation location, mutator type, tests execution traces in test suite (which test covers which lines in the source/byte code), or other available information.

The PMT implementation in [41] tries to relate designed features to three conditions in PIE theory. The limitation here is that when choosing static software metric features, manually inspecting the relation between definition of software metrics and PIE conditions is not appropriate enough to get the best match. In fact, there are many more software metrics that can be added into consideration. Therefore, in our implementation, we first try to add more software metrics and then conduct feature selection process for a better performance.

In this paper, static and dynamic features are identified as following:

1) Static Features: We use a list of software metrics as static features. A software metric is a measure of a software system characteristic which is quantifiable. Software metrics are important in many applications, including measuring software complexity [54], assessing software maintainability [55], measuring software quality [56], and so on. Recent studies have shown the successes of using software metrics in many research, including defect prediction [57], [58] and time cost reduction in mutation testing [42], [59].

In our implementation, we use *JHawk* tool[1], which has been widely used in prior research [60]–[62], to extract method, class, and package level metrics as features. Totally 89 software metrics are used, as shown in TABLE I. Note that for any enumerate type metric, i.e. *instanceVariables* metric in class level, we replace its enumeration set by the size of the set, making all metrics real-valued. For the complete definitions of the metrics, please refer to the tool's website.

Despite those software metrics, following the implementation in [42], we add return type of the mutated method (*returnType*) as a static feature, which is an important characteristic of a method and should be considered in feature design. Specifically, we use type descriptors in *ASM*[2] to annotate Java types, and "OTHERTYPE" is used to annotate user defined types. Method return type can be directly extracted from PIT[3] mutation testing report, and then matching to the designed annotation style. Note that *returnType* is a categorical feature.

The type of mutator (*MutatorClass*) used to generate the mutant is also used as a static feature. Specifically, PIT[3] is used as mutation testing tool, the mutation operators implemented in PIT are used as feature values. Clearly, *MutatorClass* is also a categorical feature.

2) Dynamic Features: Dynamic features are designed based on mutation operation location and execution traces of tests in the test suite. Those features are shown to be very important in PMT. Following the implementation in [42], we consider 4 dynamic features, including:

- *numExecuted*, which refers to the number of times the mutated statement gets executed by the whole test suite.
- *numTestCover*, which refers to the number of test methods from the test suite covering the mutated statement.
- *numAssertInTM*, which refers to the total number of assertions in the test methods covering the mutated statement.
- *numAssertInTC*, which refers to the total number of assertions in the test classes testing the mutated statement.

Totally, we have 91 static features + 4 dynamic features = 95 features, TABLE II summarizes the features considered in our PMT implementation. Note that those features subsume all features implemented in [42], except *instability* which can be derived by a division using two other features in our features.

C. Classification Algorithm

Machine learning technique has shown its success in many software engineering research [18], [42], [48], [63], [64]. In mutation testing, what we care about is whether a mutant can be killed by the test suite, i.e., there are only two available status, killed or alive for each mutant after execution, reducing PMT to a binary classification problem.

Choosing a good classification algorithm is vital for training and prediction stage in PMT framework. In this paper, a total of 11 popular classification algorithms are investigated when applying them to PMT, including:

[2]https://asm.ow2.io
[3]http://pitest.org

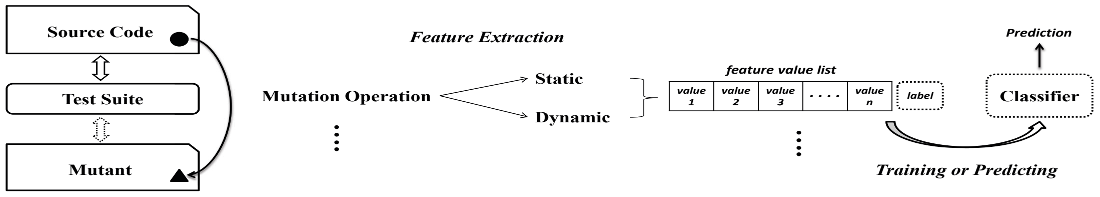


Fig. 1: The general framework of PMT.

TABLE I: List of Software Metrics Used as Features

Level	Metric List:
package	abstractness, avcc, cumulativeNumberOfCommentLines, cumulativeNumberOfComments, distance, fanin, fanout, halsteadCumulativeBugs, halsteadCumulativeLength, halsteadCumulativeVolume, halsteadEffort, instability, loc, maintainabilityIndex, maintainabilityIndexNC, maxcc, numberOfClasses, numberOfMethods, numberOfStatements, RVF, tcc
class	avcc, cbo, coh, cumulativeNumberOfCommentLines, cumulativeNumberOfComments, dit, externalMethodCalls, fanIn, fanOut, halsteadCumulativeBugs, halsteadCumulativeLength, halsteadCumulativeVolume, halsteadEffort, hierarchyMethodCalls, importedPackages, instanceVariables, interfaces, lcom, lcom2, loc, localMethodCalls, maintainabilityIndex, maintainabilityIndexNC, maxcc, messagePassingCoupling, modifiers, numberOfCommands, numberOfMethods, numberOfQueries, numberOfStatements, numberOfSubclasses, numberOfSuperclasses, responseForClass, reuseRatio, REVF, six, specializationRatio, tcc, unweightedClassSize
method	classesReferenced, cyclomaticComplexity, exceptionsThrown, externalMethodsCalled, halsteadBugs, halsteadDifficulty, halsteadEffort, halsteadLength, halsteadVocabulary, halsteadVolume, instanceVariablesReferenced, loc, localInstanceVariablesReferenced, maxDepthOfNesting, modifiers, numberOfArguments, numberOfCasts, numberOfCommentLines, numberOfComments, numberOfExpressions, numberOfLoops, numberOfOperands, numberOfOperators, numberOfStatements, numberOfVariableDeclarations, numberOfVariableReferences, totalNesting, variablesDeclared, variablesReferenced

TABLE II: Features Summary

Type	Metric List:
static	features-in-TABLE I, returnType, MutatorClass
dynamic	numExecuted, numTestCover, numAssertInTM, numAssertInTC

- *Tree Structured Algorithms*: Random Forest [45], Extra Trees [65], Decision Stumps [66], C4.5 [67], caForest [53]
- *Neural Network Structured Algorithms*: Multilayer Perceptron [51], Deep Multilayer Perceptron [51], Convolutional Neural Network [68]
- *Others*: SVM [69] with linear kernel, k-Nearest Neighbors [70], Gaussian Naïve Bayes [71]

For disambiguation, we use *MLP* to denote Multilayer Perceptron and *deep MLP* to denote Deep Multilayer Perceptron in the following discussion.

Here we briefly introduce three algorithms that show best performance in our evaluation:

Random Forest [45] is an ensemble learning method for classification, regression, and other tasks. It works by constructing a bunch of decision trees at training phase. During the training process, each tree grows by randomly selecting $\sqrt{d}$ (d is number of features) number of features as candidates and choosing the one with the best *gini* value to split at each tree node, until leaf node contains only instances with the same class or the tree depth reaches certain pre-defined value. When used for classification, the class (label) for an input instance will be the mode of the classes of the ensembled decision trees.

Convolutional Neural Network [68] (CNN) is an efficient feedforward neural network architecture to extract statistical patterns in large-scale and high-dimensional datasets. It extracts local property of training data by revealing local features that share across the data domain. These similar features are identified with localized convolutional filters that are learnt from the data. CNN's ability of learning local structures in a hierarchical way has led the successes when applying it to image, video, sound, and graph related tasks [68], [72]–[74].

caForest [53] (*gcForest* without multi-grained scanning) is a decision tree ensemble approach that employs a cascade learning structure. Each level of the cascade is a bunch of decision forests that receive feature information processed by its preceding level and output its processing result to the next level. Different types of forests can be chosen in the cascade level to add model diversity. *caForest* has been shown to be competitive to deep neural network models in a board range of tasks. Fewer hyper-parameters and more interpretable are two main advantages of caForest when compared with deep neural network models.

III. Experimental Setup

A. Research Questions

Our experiments investigate the following research questions:

- **RQ1:** How do different data and classifier configurations impact the effectiveness of PMT?
- **RQ2:** How does PMT perform in predicting mutation testing results in terms of efficiency?
- **RQ3:** How does PMT perform in predicting mutation testing results in terms of effectiveness?
- **RQ4:** How do different features impact the effectiveness of PMT in predicting mutation testing results?

RQ1. In this research question, we investigate various settings or configurations in data and classification algorithms that help improve the effectiveness of PMT. For data, we consider if sample weighting, the classical imbalance strategy, or data scaling, help improve the prediction accuracy. In addition, we try to manipulate the dynamic features to refine data scaling. For classification algorithms, we try to find the best data and parameter settings for each classifier model. In RQ1 we focus on finding the best settings for each classifier in a reduced dataset using cross validation. Specifically, 40 subjects' corresponding mutant dataset are used, which contain a total of more than 335k mutant record. Then the classifiers with best setting in our search space are used in the following part to fully evaluate PMT on the whole subjects' dataset.

RQ2. This research question is answered by comparing the running time of executing the whole mutants against the test suite and the running time of feature extraction, mutant generation, and classifier prediction in PMT for each subject, and then summarize the speedup statistics of all the subjects. In PMT, different classifiers differ in their training time. However, the training process can be done in an **offline** mode (i.e., a trained model can be applied to any future projects without retraining), thus it will not be counted into this efficiency comparison. Note that this comparison is conducted on the reduced subject set used in RQ1, where the corresponding running time data are recorded.

RQ3. In this research question, we focus on comparing the effectiveness of different classification algorithms in PMT using various statistics. Specifically, the mutant dataset from all 654 subjects are used. Classification algorithms with the best data and parameter settings gotten from RQ1's experiments are evaluated using cross validation to see if they can generalize well on the full dataset.

RQ4. This research question is answered by investigating the importance of each individual feature in the classification algorithm. Especially, we use Random Forest, which is shown to generalize well in full dataset, to extract the importance score for each feature. The importance calculation is based on the implementation in *scikit-learn*[4]. Then we try to find patterns based on features importance scores and types. We also dive into specific subjects' datasets to see why the classification algorithms perform well or poorly. Considering the importance of feature design, those findings are vital for refining the PMT in the future.

B. Subject Systems

To extensively evaluate different aspects of our PMT implementation, we collect mutant record datasets for 654 real-world Java subjects from GitHub[5]. In details, we collected 8841 unique Java projects with over 20 forks from GitHub; 4928 of them support *Maven*[6] build system; 2953 of them have a green test suite; where *PIT*[3] and our coverage collection tool can be executed successfully on 654 of them. As we focus on the cross-project setting, each mutant record dataset corresponds to one revision of the related project. Summary of statistics of those projects can be find in TABLE III. A total more than 4M mutant records joins our evaluation.

Within the 654 projects' datasets, 40 of them (reduced subjects) are selected to construct a reduced dataset. This

[4]http://scikit-learn.org
[5]https://github.com
[6]https://maven.apache.org

TABLE III: Summary of Subject Statistics

Statistic	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
LOC	25	704	1655	8183	5748	464156
# Packages	1	2	5	12	11	477
# Classes	2	14	37	130	110	5400
# Methods	3	64	173	786	618	33811
# Tests	1	5	15	113	63	10881
# Mutants	5	401	1156	6227	4128	356520
# Killed	0	58	277	1395	1045	78184
Killed Pct.	0.00	0.08	0.32	0.36	0.60	1.00

TABLE IV: Summary Statistics of Reduced Subjects

Statistic	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
LOC	1272	5164	7695	29426	21004	464156
# Packages	1	5	9	16	16	120
# Classes	9	102	128	341	379	3571
# Methods	219	505	752	2080	2298	17341
# Tests	20	106	211	745	393	10881
# Mutants	522	1908	4561	8393	9224	38541
# Killed	349	1341	2304	4660	4975	32272
Killed Pct.	0.04	0.44	0.65	0.60	0.76	0.98

*Those subjects are used for RQ1, RQ2, and RQ4.

reduced dataset is used to help data preprocessing and parameter settings for each classifier (RQ1). Also, the mutation testing time and feature collection time for each project are recorded to evaluate the efficiency of PMT (RQ2) because mutation testing time for other subjects are not collected. In addition, the feature ranking calculated using this reduced dataset is also used for analysis of feature importance (RQ4). Specifically, follow the implementation in [42], we include the 9 base projects in this reduced dataset, those projects have been widely used in previous software testing research [4], [75], [76]. Then we collect another 31 projects to enrich the dataset to make the settings in RQ1 more convincing considering the scale of our experiments. The summary statistics of the reduced subjects are shown in TABLE IV. Note that to get more representative data and parameter settings for the classifiers, we use more complex subjects than average in the reduced subject set (we limit the minimum LOC to be 1k and randomly select the other 31 subjects from all 654 subjects to form the reduced subject set), while in the full subject list, all subjects (including small ones) are included to reduce the threats to external validity, causing the difference in statistics between TABLE III and TABLE IV.

C. Supporting Tools and Implementation

We summarize the tools we used in our PMT implementation as following:

Mutation test tool. We choose the popular Java mutation tool, *PIT*[3], as our mutation testing tool. *PIT* is a state-of-the-art mutation testing system, it is fast, scalable, and can be integrate with modern test and building tool like *Maven*[6]. *PIT* has been shown to be both efficient and robust [77], enabling its usage in large-scale experimental study and making it widely used in software engineering research [18], [42], [76].

Feature collection tools. To collect static and dynamic features, we use the following tools and methods:

- *Static.* To collect static features, both *JHawk*[1] and *PIT* are used. *PIT* report contains the mutated statement information which can be used to match *JHawk* report to get the corresponding package, class, and method metrics. Mutator used to generate each mutant can be directly extracted from *PIT* report. *PIT* report also contains the return type of mutated method, which can be used to match to our type annotations as described in section II.

- *Dynamic.* To collect dynamic features. We implement our own tool based on *ASM*[2]. Specifically, our coverage collection tool is implemented via extending *IntelliFL*[7], on which we add one Maven plugin to extend its original statement coverage collection utilities to collect the detailed coverage information for each test in the test suite. Then the designed four dynamic features can be calculated based on the mutation location and the detailed coverage report.

Machine learning tools. *scikit-learn*[3] and *Keras*[8] are used to implement eight non-deep models and two deep neural network models respectively. They are both popular machine learning libraries and have been used in many research. We use the official implementation of *gcForest*[9] to implement the *caForest* model.

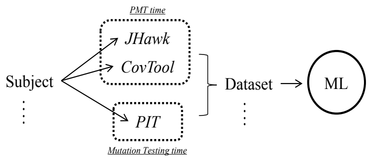


Fig. 2: Tool usage in the experiment.

Fig. 2 shows the tool usage in our experiment. For each subject, we run *JHawk* and the *coverage collection tool* to generate software metric report and statement coverage report, these two tools' running time are recorded and summed as parts of PMT running time for subject in the reduced subject set. *PIT* is used to run mutation testing for each subject to generate mutation testing report, also, for subject in the reduced subject set, we record *PIT*'s running time as mutation testing time. We then generate dataset from these three reports and used it in the machine learning part where we build different classifiers to evaluate the performance of PMT.

More details of all dataset and machine learning model implementation can be found in our project homepage[10]. All the experiments were performed on a platform with 14-core Intel Xeon E5-2660 CPU (2.0GHz) and 220 GiB RAM running Java 1.8.0_181 on Ubuntu 14.04.5 LTS.

D. Measurements

We evaluate the effectiveness of our PMT implementation using prediction *accuracy*, *error*, *precision*, *recall*, *F1-score*, and *AUROC* for *each subject* using 5-fold cross validation and then *summarize* (e.g. use mean, median) those statistics to compare the performance of different classification algorithms. In this paper, we label killed mutants as positive, alive mutants

[7]http://www.intellifl.org
[8]https://keras.io
[9]https://github.com/kingfengji/gcForest
[10]https://github.com/SElab2019/ExtPMT

TABLE V: Accuracy Summary of two Scaling Strategies

Strategy	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
$+$ *log*	**0.6150**	0.8456	**0.8815**	**0.8762**	**0.9187**	0.9870
$-$ *log*	0.6145	**0.8470**	0.8802	0.8748	0.9169	0.9870

*Best ones are shown in bold.

as negative, and calculate the measures for each subject as follows (denote TP as true positive, TN as true negative, FP as false positive, FN as false negative):

- Accuracy = (TP + TN) / (TP + FP + TN + FN).
- Precision = TP / (TP + FP), Recall = TP / (TP + FN).
- F1-score is the harmonic mean of precision and recall.
- AUROC is the area under ROC [78].

IV. Experiment Results

A. RQ1: Data Preprocessing and Classifier Settings

Scaling strategy. In data preprocessing, we consider *whether scaling non-categorical features helps improve the performance of each classifier*. Specifically, we notice that in many datasets, some mutant records have very large dynamic feature values than average. If directly apply scaling to raw data, the dynamic feature values for many mutant records that are very small will be hard to distinguish. Thus we investigate whether applying log to dynamic feature values before scaling helps improve the performance. We use Random Forest with 5-fold cross validation, the accuracy summary is shown in TABLE V. Clearly, we should apply log before scaling.

Feature selection. In the total 95 features, we consider *whether using a subset of these features helps improve the performance of each classifier*. First, the importance score for each feature is calculated based on Random Forest classifier using the reduced subject set, which is shown to be suitable for PMT [41]. Second, we rank the features according to their importance scores. Third, we consider using 10 to 40 top ranked features and try to find the best subset using 5-fold cross validation in the reduced subject set. Note that we choose step size as 2 to reduce the time consumption used in this process. Also, the feature importance scores are calculated 3 times and then averaged considering the randomness when building the Random Forest classifier. Our result shows that using 12 top ranked features can acquire best performance in mean and median of the prediction accuracies.

Classifier settings. For all 11 classification algorithms, we first test if scaling helps improve prediction accuracies. Then we test if sample weighting helps improve the prediction accuracies, note that MLP and caForest have no sample weighing option so we omit it. For tree-based classifiers, we use default depth settings in the supporting tools and search number of trees that help produce the best performance. For neural network-based classifiers, we try different hyper-parameter settings to get the best results. For deep-MLP and CNN, we first tune activation type and optimizer using a reasonable layer setting, and then tune the layers. Specifically, for CNN, we use 1-d convolutional layers. Note that for deep models, considering their representation learning ability, we also consider using full feature list. The details of classifier settings are as following:

- *Random Forest*: no-scaling, no-weighting, 150 trees;
- *Extra Trees*: scaling, no-weighting, 100 trees;
- *Decision Stumps*: no-scaling, weighting, 100 trees;
- *C4.5*: scaling, no-weighting;
- *Naïve Bayes*: scaling, no-weighting;
- *SVM*: scaling, no-weighting;
- *k-Neighbors*: scaling, k=13;
- *MLP*: scaling, hidden_layers=[5, 4];
- *deep-MLP*: scaling, no-weighting, hidden_layers=[95, 45, 45] (full features) or [24, 24, 24] (12 features), sigmoid activation, Adagrad [79] optimizer;
- *CNN*: no-scaling, no-weighting, filters=[36, 36] (for both full and 12 features), kernel and step size are set to 6, Adadelta [80] optimizer;
- *caForest*: no-scaling, use Random Forest and Extra Trees in each layer both with 50 trees.

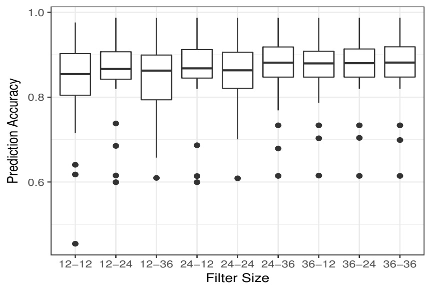


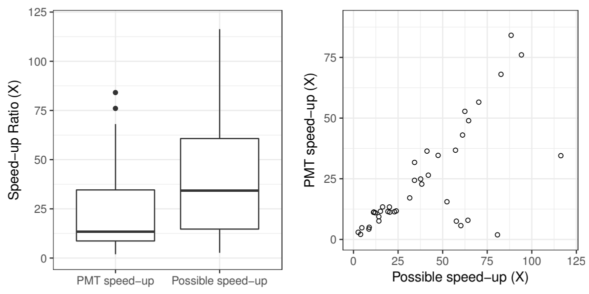
Fig. 3: Accuracy of CNN (full features) with different filters.

From the experiment results, we have the following important findings:

- Data preprocessing helps improve the classification performance of PMT. In this paper, we consider applying log to dynamic features before scaling and get better prediction accuracies as shown in TABLE V.
- Different classifiers have different preferences on data preprocessing. As shown in the classifier setting part, some classifiers prefer scaling, or weighting, while others prefer raw data.
- Hyper-parameter tuning is vital for classifiers in PMT, especially for deep models. For example, Fig. 3 presents the prediction accuracies when we try to find the best filter settings for CNN with the full feature set. We can see clear differences in the boxplots, specifically, the mean prediction accuracy increases by 4% if we choose [36, 36] as filter settings rather than [12, 12].

B. RQ2: Efficiency

We compare the running time of our PMT implementation with mutation testing. PMT running time is calculated by summing *JHawk* and *coverage tool* running time, mutant generation time, feature collection time, and classifier prediction time (use Random Forest with feature list in [41] as example). Note that when running mutation testing, we use 5 threads to speed up the process and record the time for comparison. We



(a) Speed-up ratio comparison. (b) Speed-up relation.

Fig. 4: Speed-up ratios in our PMT implementation v.s. theoretical best speed-up ratios. One outlier subject with very large speed-up ratios (233.7X in our implementation and 519.0X in theoretical best) is omitted for better demonstration.

also calculate the theoretically minimum predictive mutation testing time by only summing up the test execution time and mutation generation time[11]. Fig. 4a shows the comparison between our PMT implementation and the theoretically best speedup. Specifically, in statistics, our implementation shows speed-up ratios with an average of 28.7X and a median of 13.4X, where the theoretically best technique shows an average 51.7X and a median 36.0X.

From the experiment results, we have the following important findings:

- PMT is highly efficient compared with running mutation testing. In fact, 75% subjects in the reduced set get a speed-up more than 9X with our PMT implementation (compared to traditional mutation testing with 5 threads).
- A near linear relation can be found between the speed-up of our PMT implementation and the theoretically fastest implementation (Fig. 4b). Exceptions come from when coverage collection or software metric collection cost much more time than expected, which can be refined to make the implementation more efficient.

C. RQ3: Effectiveness

We evaluate the effectiveness of PMT using 5-fold cross validation on all 654 subjects. Specifically, in each fold, 80% of the subjects' datasets are used for training and the remaining 20% are used for testing. The prediction statistics (accuracy, error, precision, recall, F1 score, and AUROC) are recorded for each subject. The median and mean of these statistics for each classifier are listed in TABLE VI. For *deep structured classifiers* and *Random Forest*, we implement them with subset (12) and full (95) features, denoted as classifier name followed with "(12)" and "(95)". Other *non-deep structured classifiers* are implemented with subset features. In addition, *Random Forest* is also implemented using features designed in [41] as comparison, denoted as "*Random Forest [41]*".

[11]Note that only mutant generation and test execution time are unavoidable for any precise predictive mutation testing, since test execution is necessary to obtain precise test information, while mutant generation is the first step to perform predictive mutation testing.

TABLE VI: PMT Performance Comparison of Different Classifiers

Classifier	Accuracy		Error		Precision		Recall		F1 score		AUROC	
	Median	Mean	Median	Mean	Median	Mean	Median	Mean	Median	Mean	Median	Mean
Random Forest (12)	0.8859	**0.8466**	0.1141	**0.1534**	**0.7695**	0.6554	0.9709	0.7773	0.8299	0.6879	**0.8979**	**0.7779**
Random Forest (95)	0.8841	0.8442	0.1160	0.1558	0.7667	0.6545	0.9726	0.7625	0.8235	0.6764	**0.8981**	**0.7770**
Random Forest [41]	0.8838	0.8447	0.1162	0.1553	0.7656	**0.6591**	0.9408	0.7656	0.8255	0.6877	**0.8993**	**0.7774**
Extra Trees	0.8802	0.8397	0.1198	0.1603	0.7494	0.6407	0.9834	0.8032	0.8343	0.6945	0.8872	0.7635
Decision Stumps	0.8745	0.8358	0.1255	0.1642	0.7573	0.6452	0.9432	0.7653	0.8121	0.6781	0.8843	0.7590
C4.5	0.8000	0.7878	0.2000	0.2122	0.7625	0.6453	0.6714	0.5820	0.6953	0.5911	0.7618	0.6698
Naïve Bayes	0.8533	0.8140	0.1467	0.1860	**0.7675**	**0.6575**	0.8665	0.7391	0.7819	0.6695	0.8752	0.7559
SVM	0.8491	0.8038	0.1509	0.1962	0.7667	0.6294	0.7505	0.6327	0.7209	0.6012	0.8739	0.7493
k-Neighbors	0.8338	0.8131	0.1662	0.1869	**0.7684**	0.6510	0.8063	0.6858	0.7622	0.6454	0.8497	0.7424
MLP	0.8806	0.8362	0.1194	0.1638	0.7427	0.6358	0.9958	0.8185	0.8338	0.6978	0.8685	0.7517
deep-MLP (12)	0.8820	0.8379	0.1180	0.1621	0.7529	0.6467	**0.9984**	**0.8362**	0.8379	**0.7094**	0.8742	0.7481
deep-MLP (95)	0.8748	0.8348	0.1252	0.1652	0.7579	0.6468	0.9823	0.7919	0.8255	0.6898	0.8700	0.7562
CNN (12)	**0.8872**	0.8445	**0.1128**	0.1555	0.7495	0.6334	**0.9995**	0.8047	**0.8392**	0.6913	0.8856	0.7639
CNN (95)	**0.8893**	**0.8453**	**0.1107**	**0.1547**	0.7487	0.6387	**1.0000**	**0.8299**	**0.8432**	**0.7038**	0.8780	0.7551
caForest (12)	0.8761	0.8448	0.1239	0.1552	0.7661	**0.6595**	0.9313	0.7803	0.8225	0.6965	0.8918	0.7742
caForest (95)	**0.8874**	**0.8467**	**0.1126**	**0.1533**	0.7519	0.6454	0.9973	**0.8356**	**0.8466**	**0.7103**	0.8927	0.7722

* Top 3 values for each statistic are shown in **bold**.

For comparison, we conduct a *random guessing with probability*. The probability of predicting a mutant to be killed is set to be the proportion of killed mutants in all datasets. The median and mean prediction accuracy of this random guessing in the subject set are 0.6005 and 0.5756, much lower than all the classifiers used in our evaluation, showing the powerfulness of machine learning approach in PMT.

From TABLE VI, we have the following observations:

- Best classifiers (*Random Forest, CNN, caForest*) provide cross validation accuracies and AUROCs with about 0.89 in median, which demonstrates the effectiveness of PMT.
- The median of prediction recalls is very high for many classifiers, which means that for most subject, the killed mutants can be almost figured out completely.
- Comparing with recall, the statistics for precision are much lower. It means that inside the predicted killed mutants, the real killed mutants' proportion is not very high. The reason lies in that dynamic features can trigger the prediction result to be a killed one. We will investigate it in the next subsection.
- More features tend to benefit the deep learning models more than the traditional *Random Forest* model.
- The gap between median and mean of these statistics tells that for some subjects, PMT shows a bad performance. We discuss several interpretations in the next section.

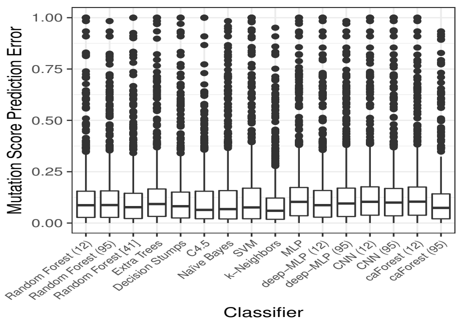


Fig. 5: Comparison of mutation score prediction.

Fig. 5 shows the comparison of different classifiers when used for predicting mutation score. As we can see, for all classifiers, the absolute prediction error has a median of 0.1 or lower. However, it is clear that predicting mutation score is more difficult because the average mutation score for all subjects are about 0.36, meaning that the percentage share of error is larger than that of predicting the label. It is also interesting to note that *k-Neighbors* shows the best performance in this task.

TABLE VII: Pairwise comparison of Random Forest (12) and CNN (95) on 654 subjects

Statistic	Random Forest (12) win	Draw	CNN (95) win
Accuracy	41.6%	25.7%	32.7%
F1 score	27.7%	29.0%	43.3%
AUROC	66.5%	15.1%	18.4%

Next, we consider the comparison between classifiers. We use *Mann-Whitney U test* to test if one classifier is better than the other for a statistic, specifically, we set the confidence level to be 95%. For prediction accuracy, consistent with prior work [42], *Random Forest* based classifiers is shown to be better than *Decision Stumps*, *C4.5*, *Naïve Bayes*, *SVM*, and *k-Neighbors*. However, the differences are not enough to tell that it is better than the others like *CNN* and *caForest*.

TABLE VII shows a pairwise comparison between two representative classifiers, *Random Forest (12)* and *CNN (95)*, in prediction accuracies, F1 scores, and AUROCs on all 654 subjects. We can see that *Random Forest (12)* has advantages in accuracy and AUROC, while *CNN (95)* has advantages in F1 score. Combined with the result showed in Fig. 5, we can get an important finding:

- For different prediction tasks, PMT should consider different classifiers or objectives for better performance. For example, in this study, deep classifiers are designed to optimize accuracy, if we change the focus to F1 score, we may use it as objective to pursue a better result.

D. Q4: Feature Importance Analysis

Random Forest classifier is used to calculate feature importance as it shows good performance in PMT. Fig. 6 shows the importance scores for all the 95 features and the selected 12 features calculated on full and reduced subject set. Note that

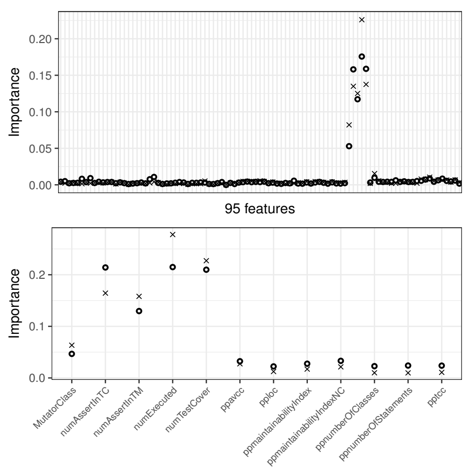


Fig. 6: Feature importance scores for 95 (upper) and 12 (below) features calculated on full subject set (○) and reduced subject set (×). Feature names are omitted due to space limitation for the upper one, while the top-5 features share the same label sequence in both figures.

considering the randomness, we build the classifier 3 times and average the importance for each feature. From the results, we observe that the top-5 features are 4 dynamic features and the mutator type in all 4 calculation settings. Specifically, the sum of importance of 4 dynamic features shares more than 60% of the total importance in all settings, which means that dynamic features dominate the prediction in the fitted classifier.

We also notice an interesting point that package-level software metric features are more important than those of class and method level. When calculating the feature importance using all 654 subjects, more than 95% (20/21) package level features ranked within top-40, while this statistic for class and method level features are only 25% (10/39) and 17% (5/29). The reason may lie in that for each subject, there are usually much smaller number of packages than classes and methods (TABLE III, TABLE IV), making the package-level metrics able to represent more subject-level characteristics than the metrics for individual classes and methods. From the differences between mean and median of the prediction statistics listed in TABLE VI, we see that different subjects have different characteristics, while most of them enjoy good performance when using PMT, some are not. The distinction of each subject may cause the preferences of package level metrics during the classifier training process.

The feature importance for each subject is also analyzed. Fig. 7 summarizes the sum of importance for top-5 features in all 654 subjects. We can see that for majority of subjects, the top-5 features are much more important than the other features. This explains why PMT works in one aspect because the trained classifiers put more attention on dynamic features and mutator type (top-5) which are important in most subjects for distinguishing the killed mutants as shown in Fig. 6.

We also observe that there exist 82 subjects with zero importance sum for top-5 features. We dive into these subjects

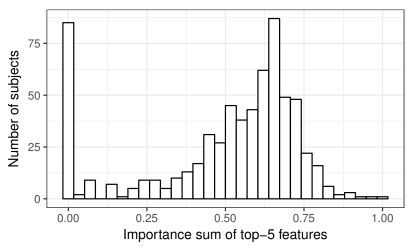


Fig. 7: Distribution of importance sum for top-5 features.

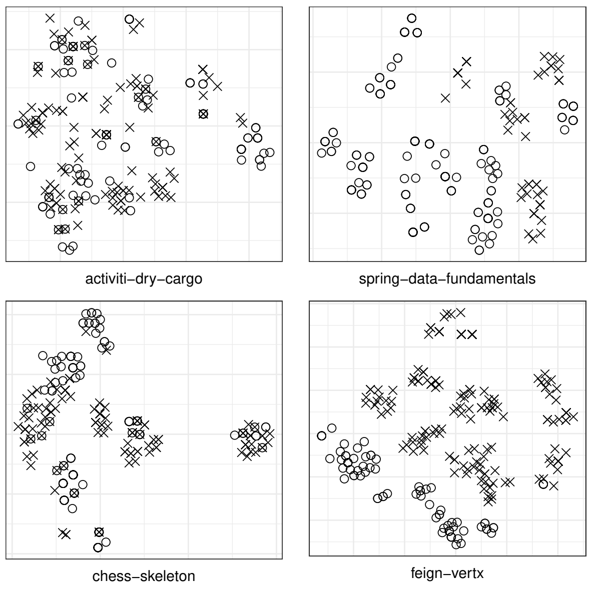


Fig. 8: t-SNE visualization (2-d) of subjects' datasets (subtitles are subject names) with low (left two) and high (right two) prediction accuracies, (×) denotes killed mutants, (○) denotes alive mutants. We sampled 250 mutant records in each dataset to present.

and find that all of them are extremely imbalanced, i.e., all mutants are killed or alive. The mean prediction accuracy in these subjects is only 0.7068, clearly it is one reason that make the difference between the mean and median of the statistics as shown in TABLE VI.

In addition, we use *t-SNE* [81] to perform dimension reduction (mapping the original feature set into a two-dimensional surface) and visualize each subject's dataset. Fig. 8 shows a selected of four visualizations from subjects with good or bad prediction accuracy. Compared with the right two, clearly the left ones show more overlapping between killed (×) and alive (○) mutant records. It means that the features are not powerful enough to separate killed and alive mutants, so more powerful features should be designed to strengthen PMT for those subjects.

In summary, to further improve the performance of PMT, we have the following suggestions: 1) add more dynamic features as they dominate the prediction; 2) add more high-level (e.g., package-level) software metrics as they are shown to be more important than the detailed class and method level metrics; 3) consider adding other type of features like semantic features [63], data dependency and AST features [82] to encode more program characteristics into the feature list.

E. Threats to Validity

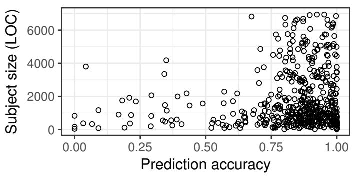


Fig. 9: Subject size vs prediction accuracy (use Random Forest (12) as classifier), subjects with more than 7k LOCs are filtered for better demonstration.

The threat to *internal validity* lies in the implementations, we reduce this threat by using widely used tools to do mutation testing, feature extraction, training and testing in our PMT implementation. The threat to *external validity* lies in the subject set, we will consider trying to characterize which types of subjects benefit more from PMT. As an initial attempt, we calculated the Pearson correlation between the subject size (LOCs) and prediction accuracies (results from Random Forest (12)), and found that the coefficient is only 0.08, indicating that project size itself is not a good indicator for effectiveness (as seen in fig. 9) and we should consider more useful features in the future. Also, as our implementation only support Java project now, we will consider extending it to support cross-language setting. The threat to *construct validity* lies in the measurements used to evaluate PMT, in this study we use different measures and analyze the performance of our PMT under them to reduce this threat. However, we may still lack effective measurements, e.g., more detailed analysis about whether fault revealing mutants belong to the reported error margin or not [83].

V. Related Work

Mutation testing is a powerful methodology to evaluate the quality of test suite and is gaining more and more attention in both academia and industry [5], [9]–[11]. One of the main limitations of mutation testing lies in its efficiency as it is extremely time consuming to execute the whole generated mutant set against the test suite [5]. Therefore, many researchers have proposed various techniques to reduce the cost (discussed as follows). Please refer to a recent survey for more details about the limitations and optimizations for mutation testing [30].

Selective mutation testing focuses on reducing the number of mutants. Wong and Mathur [33] found that several mutation operators contribute most to the quality of the generated mutants. Offutt et al. [35] Showed that selective mutation testing performs comparable to non-selective mutation testing and proposed a technique to reduce the number of mutants by an order of magnitude. Following these works, many research have focused on deciding an enough subset of mutation operators for selective mutation testing [34], [36], [84]. Beside the operator-based selection, random selection is also shown to be a good strategy [4].

Weak mutation testing speeds up mutation testing by partially executing mutants. The concept of weak mutation testing is proposed by Howden [37]. The idea is that weak mutation testing uses a weaker definition of mutant killing that judge a mutant as killed if there are differences in program states between mutant and original program executions. Weak mutation testing is shown to be as effective as mutation testing with significant computational savings [85].

There are also other ways proposed to improve the efficiency of mutation testing. DeMillo et al. [31] proposed compiler-integrated method to support program mutation via information generated at compile-time. Untch et al. [32] proposed a method that uses program schemata to encode all mutants for a program into one meta-program to speed mutant execution. Some research [10], [86] also investigated using parallel method to speed up the mutation execution process. To speed up mutation testing for evolving system, researchers have also proposed analyze program differences to incrementally collect mutation testing results [40], [87].

Recently *predictive mutation testing* is proposed to predict the mutant execution results using pre-trained classifier [42]. PMT characterizes each mutant to a list of features and then train a classifier using a set of mutant execution records to predict whether a new mutant will be kill or not by the test suite. Because the mutant execution phase in mutation testing is fully replaced by a classifier inference, PMT achieves a significant speedup. This work further presents an extensive study of the PMT work in the cross-project setting by considering more features, projects, and classifier settings, and also investigating the importance of each feature to reveal important guidance to the future work on this line.

VI. Conclusion

In this paper, we present an extensive study to evaluate the performance of *predictive mutation testing*, a methodology that aims to predict mutation testing results using machine learning approach. We start our design by including more static software metrics as features and considering powerful deep learning algorithms. In summary, a total of 95 features and 11 classification algorithms are evaluated in this study. All classifiers are carefully tuned to select their preferred data and parameter settings, then a costly 5-fold cross validation are used on the datasets generated from 654 real-world projects with 4M mutant records to evaluate the performance of our PMT implementation. Our results show that PMT is both efficient and effectiveness when applied to most subjects. We also studied the importance of each feature and the results show that dynamic features are dominant in prediction, which provide an important guidance for refining PMT design. In the future, we will extend our study by designing more powerful features to improve the prediction quality, and also try to evaluate PMT in subjects with other languages.

VII. Acknowledgements

This work was partially supported by National Science Foundation grants CCF-1566589 and CCF-1763906.

References

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

[2] R. G. Hamlet, "Testing programs with the aid of a compiler," *TSE*, no. 4, pp. 279–290, 1977.

[3] D. Schuler and A. Zeller, "Javalanche: Efficient mutation testing for java," in *FSE*, 2009, pp. 297–298.

[4] L. Zhang, S. Hou, J. Hu, T. Xie, and H. Mei, "Is operator-based mutant selection superior to random mutant selection?" in *ICSE*, vol. 1, 2010, pp. 435–444.

[5] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *TSE*, vol. 37, no. 5, pp. 649–678, 2011.

[6] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, "Evaluating and improving fault localization," in *ICSE*, 2017, pp. 609–620.

[7] A. Sullivan, K. Wang, R. N. Zaeem, and S. Khurshid, "Automated test generation and mutation testing for alloy," *ICST*, pp. 264–275, 2017.

[8] S. Mirshokraie, A. Mesbah, and K. Pattabiraman, "Guided mutation testing for javascript web applications," *TSE*, vol. 41, no. 5, pp. 429–444, 2015.

[9] A. Groce, I. Ahmed, C. Jensen, and P. E. McKenney, "How verified is my code? falsification-driven verification (t)," in *ASE*, 2015, pp. 737–748.

[10] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "Pit: A practical mutation testing tool for java (demo)," in *ISSTA*, 2016, pp. 449–452.

[11] I. Ahmed, C. Jensen, A. Groce, and P. E. McKenney, "Applying mutation analysis on kernel test suites: An experience report," in *ICSTW*, 2017, pp. 110–115.

[12] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *ICSE*, 2005, pp. 402–411.

[13] 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 *FSE*, 2014, pp. 654–665.

[14] Y. Lou, D. Hao, and L. Zhang, "Mutation-based test-case prioritization in software evolution," in *ISSRE*, 2015, pp. 46–57.

[15] M. Papadakis and Y. L. Traon, "Using mutants to locate "unknown" faults," in *ICST*, 2012, pp. 691–700.

[16] L. Zhang, L. Zhang, and S. Khurshid, "Injecting mechanical faults to localize developer faults for evolving software," in *OOPSLA*, 2013, pp. 765–784.

[17] S. Moon, Y. Kim, M. Kim, and S. Yoo, "Ask the mutants: Mutating faulty programs for fault localization," in *ICST*, 2014, pp. 153–162.

[18] X. Li and L. Zhang, "Transforming programs and tests in tandem for fault localization," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, pp. 92:1–92:30, Oct. 2017.

[19] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," *TSE*, vol. 38, no. 2, pp. 278–292, 2012.

[20] M. Papadakis and N. Malevris, "Automatic mutation test case generation via dynamic symbolic execution," in *ISSRE*, 2010, pp. 121–130.

[21] L. Zhang, T. Xie, L. Zhang, N. Tillmann, J. de Halleux, and H. Mei, "Test generation via dynamic symbolic execution for mutation testing," in *ICSM*, 2010, pp. 1–10.

[22] G. Fraser and A. Arcuri, "Achieving scalable mutation-based generation of whole test suites," *Empirical Softw. Engg.*, vol. 20, no. 3, pp. 783–812, 2015.

[23] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *ISSTA*, 2015, pp. 24–36.

[24] B.-C. Rothenberg and O. Grumberg, "Sound and complete mutation-based program repair," in *Formal Methods*. Springer, 2016, pp. 593–611.

[25] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *ICSE*, 2012, pp. 3–13.

[26] V. Debroy and W. E. Wong, "Using mutation to automatically suggest fixes for faulty programs," in *ICST*, 2010, pp. 65–74.

[27] M. Martinez and M. Monperrus, "Astor: A program repair library for java," in *ISSTA*, 2016, pp. 441–444.

[28] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "Context-aware patch generation for better automated program repair," in *ICSE*, 2018.

[29] A. Ghanbari and L. Zhang, "Practical program repair via bytecode mutation," *arXiv preprint arXiv:1807.03512*, 2018.

[30] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, and M. Harman, "Mutation testing advances: An analysis and survey," in *Advances in Computers*, 2018.

[31] R. A. DeMillo, E. W. Krauser, and A. P. Mathur, "Compiler-integrated program mutation," in *CMPSAC*, 1991, pp. 351–356.

[32] R. H. Untch, A. J. Offutt, and M. J. Harrold, "Mutation analysis using mutant schemata," in *ISSTA*, 1993, pp. 139–148.

[33] W. E. Wong and A. P. Mathur, "Reducing the cost of mutation testing: An empirical study," *J. Syst. Softw.*, vol. 31, no. 3, pp. 185–196, 1995.

[34] A. J. Offutt, G. Rothermel, and C. Zapf, "An experimental evaluation of selective mutation," in *ICSE*, 1993, pp. 100–107.

[35] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, "An experimental determination of sufficient mutant operators," *ACM Trans. Softw. Eng. Methodol.*, vol. 5, no. 2, pp. 99–118, 1996.

[36] M. Gligoric, L. Zhang, C. Pereira, and G. Pokam, "Selective mutation testing for concurrent code," in *ISSTA*, 2013, pp. 224–234.

[37] W. E. Howden, "Weak mutation testing and completeness of test sets," *TSE*, vol. SE-8, no. 4, pp. 371–379, 1982.

[38] M. Harman, Y. Jia, and W. B. Langdon, "Strong higher order mutation-based test data generation," in *FSE*, 2011, pp. 212–222.

[39] L. Zhang, D. Marinov, and S. Khurshid, "Faster mutation testing inspired by test prioritization and reduction," in *ISSTA*, 2013, pp. 235–245.

[40] L. Zhang, D. Marinov, L. Zhang, and S. Khurshid, "Regression mutation testing," in *ISSTA*, 2012, pp. 331–341.

[41] J. Zhang, Z. Wang, L. Zhang, D. Hao, L. Zang, S. Cheng, and L. Zhang, "Predictive mutation testing," in *ISSTA*, 2016, pp. 342–353.

[42] J. Zhang, L. Zhang, M. Harman, D. Hao, Y. Jia, and L. Zhang, "Predictive mutation testing," *TSE*, pp. 1–1, 2018.

[43] A. Zheng and A. Casari, *Feature Engineering for Machine Learning: Principles and Techniques for Data Scientists*. O'Reilly Media, 2018.

[44] J. M. Voas, "Pie: a dynamic failure-based technique," *TSE*, vol. 18, no. 8, pp. 717–727, 1992.

[45] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[46] J. Guo, J. Cheng, and J. Cleland-Huang, "Semantically enhanced software traceability using deep learning techniques," in *ICSE*, 2017, pp. 3–14.

[47] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *ASE*, 2016, pp. 87–98.

[48] S.-R. Lee, M.-J. Heo, C.-G. Lee, M. Kim, and G. Jeong, "Applying deep learning based automatic bug triager to industrial projects," in *FSE*, 2017, pp. 926–931.

[49] C. Cummins, P. Petoumenos, A. Murray, and H. Leather, "Compiler fuzzing through deep learning," in *ISSTA*, 2018, pp. 95–105.

[50] Y. Tian, K. Pei, S. Jana, and B. Ray, "Deeptest: Automated testing of deep-neural-network-driven autonomous cars," in *ICSE*, 2018, pp. 303–314.

[51] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

[52] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *NIPS*, 2012, pp. 1097–1105.

[53] Z.-H. Zhou and J. Feng, "Deep forest: Towards an alternative to deep neural networks," in *IJCAI*, 2017, pp. 3553–3559.

[54] T. Honglei, S. Wei, and Z. Yanan, "The research on software metrics and software complexity metrics," in *IFCSTA*, 2009, pp. 131–136.

[55] D. Coleman, D. Ash, B. Lowther, and P. Oman, "Using metrics to evaluate software system maintainability," *Computer*, vol. 27, no. 8, pp. 44–49, 1994.

[56] S. H. Kan, *Metrics and Models in Software Quality Engineering*, 2nd ed. Addison-Wesley, 2002.

[57] K. Gao, T. M. Khoshgoftaar, H. Wang, and N. Seliya, "Choosing software metrics for defect prediction: An investigation on feature selection techniques," *Softw. Pract. Exper.*, vol. 41, no. 5, pp. 579–606, 2011.

[58] F. Zhang, Q. Zheng, Y. Zou, and A. E. Hassan, "Cross-project defect prediction using a connectivity-based unsupervised classifier," in *ICSE*, 2016, pp. 309–320.

[59] A. Estero-Botaro, F. Palomo-Lozano, I. Medina-Bulo, J. J. Domínguez-Jiménez, and A. García-Domínguez, "Quality metrics for mutation testing with applications to ws-bpel compositions," *Softw. Test. Verif. Reliab.*, vol. 25, no. 5-7, pp. 536–571, 2015.

[60] A. Murgia, M. Marchesi, G. Concas, R. Tonelli, and S. Counsell, "Parameter-based refactoring and the relationship with fan-in/fan-out coupling," in *ICSTW*, 2011, pp. 430–436.
[61] S. Counsell, X. Liu, S. Eldh, R. Tonelli, M. Marchesi, G. Concas, and A. Murgia, "Re-visiting the 'maintainability index' metric from an object-oriented perspective," in *SEAA*, 2015, pp. 84–87.
[62] A. Mubarak, S. Counsell, and R. M. Hierons, "An evolutionary study of fan-in and fan-out metrics in oss," in *RCIS*, 2010, pp. 473–482.
[63] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *ICSE*, 2016, pp. 297–308.
[64] T.-D. B. Le, D. Lo, C. Le Goues, and L. Grunske, "A learning-to-rank based fault localization approach using likely invariants," in *ISSTA*, 2016, pp. 177–188.
[65] P. Geurts, D. Ernst, and L. Wehenkel, "Extremely randomized trees," *Machine Learning*, vol. 63, no. 1, pp. 3–42, 2006.
[66] W. I. Ai and P. Langley, "Induction of one-level decision trees," in *ICML*, 1992, pp. 233–240.
[67] J. Quinlan, *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.
[68] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.
[69] C. Cortes and V. Vapnik, "Support-vector networks," *Mach. Learn.*, vol. 20, no. 3, pp. 273–297, 1995.
[70] B. Dasarathy, *Nearest Neighbor (NN) Norms: Nn Pattern Classification Techniques*. IEEE Computer Society Press, 1991.
[71] X. Wu, V. Kumar, J. Ross Quinlan, J. Ghosh, Q. Yang, H. Motoda, G. J. McLachlan, A. Ng, B. Liu, P. S. Yu, Z.-H. Zhou, M. Steinbach, D. J. Hand, and D. Steinberg, "Top 10 algorithms in data mining," *Knowl. Inf. Syst.*, vol. 14, no. 1, pp. 1–37, 2007.
[72] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.
[73] M. Defferrard, X. Bresson, and P. Vandergheynst, "Convolutional neural networks on graphs with fast localized spectral filtering," in *NIPS*, 2016, pp. 3844–3852.
[74] M. Niepert, M. Ahmed, and K. Kutzkov, "Learning convolutional neural networks for graphs," in *ICML*, 2016, pp. 2014–2023.
[75] A. Shi, T. Yung, A. Gyori, and D. Marinov, "Comparing and combining test-suite reduction and regression test selection," in *FSE*, 2015, pp. 237–247.
[76] A. Shi, A. Gyori, M. Gligoric, A. Zaytsev, and D. Marinov, "Balancing trade-offs in test-suite reduction," in *FSE*, 2014, pp. 246–256.
[77] M. Delahaye and L. du Bousquet, "A comparison of mutation analysis tools for java," in *QSIC*, 2013, pp. 187–195.
[78] A. P. Bradley, "The use of the area under the roc curve in the evaluation of machine learning algorithms," *Pattern Recogn.*, vol. 30, no. 7, pp. 1145–1159, Jul. 1997.
[79] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *JMLR*, vol. 12, pp. 2121–2159, 2011.
[80] M. D. Zeiler, "ADADELTA: an adaptive learning rate method," *CoRR*, vol. abs/1212.5701, 2012.
[81] L. van der Maaten and G. Hinton, "Visualizing data using t-SNE," *JMLR*, vol. 9, pp. 2579–2605, 2008.
[82] T. T. Chekam, M. Papadakis, T. F. Bissyandé, Y. L. Traon, and K. Sen, "Speeding up mutation testing via regression test selection: An extensive study," *CoRR*, vol. abs/1803.07901, 2018.
[83] M. Papadakis, D. Shin, S. Yoo, and D.-H. Bae, "Are mutation scores correlated with real fault detection? a large scale empirical study on the relationship between mutants and real faults," in *ICSE*, 2018.
[84] A. S. Namin, J. Andrews, and D. Murdoch, "Sufficient mutation operators for measuring test effectiveness," in *ICSE*, 2008, pp. 351–360.
[85] A. J. Offutt and S. D. Lee, "An empirical evaluation of weak mutation," *TSE*, vol. 20, no. 5, pp. 337–344, 1994.
[86] A. J. Offutt, R. P. Pargas, S. V. Fichter, and P. K. Khambekar, "Mutation testing of software using a mimd computer," in *ICPP*, 1992, pp. 257–266.
[87] L. Chen and L. Zhang, "Speeding up mutation testing via regression test selection: An extensive study," in *ICST*, 2018, pp. 58–69.