The CSI Journal on
Computer Science and Engineering
Vol. 19, No. 1, 2025
Pages 43-53
doi.org/10.22034/jcse.2025.181984
Regular Paper

Learning to Predict Software Testability Axiomatically

Morteza Zakeri[1] Saeed Parsa[2]
[1]School of Computer Engineering, Amirkabir University of Technology (Tehran Polytechnic), Tehran, Iran.
[2]School of Computer Engineering, Iran University of Science and Technology, Tehran, Iran.

Abstract

Software testability is the propensity of code to reveal its existing faults, particularly during automated testing. Testing success highly depends on the testability of the program under test. On the other hand, testing success relies on the test coverage provided by a given test data generation algorithm. However, little empirical evidence has been proposed to clarify whether and how software testability affects test coverage. This article proposes an approach to measure testability based on an axiomatic test quality metric, i.e., branch coverage. The main difficulty is determining the branch coverage before performing the actual testing to reduce the cost and time imposed by the program execution. To address this problem, we leverage machine learning classification techniques to predict the extent to which a class under test could be covered based on static source code metrics. Automatic test data generation is applied to compute the branch coverage, a concrete proxy to quantify source code testability, for a large corpus of 110 Java project with 23,000 classes. An extensive set of software metrics represents each class to be used in the classification process. Our experiments show an acceptable accuracy of 81.94% in predicting software testability.

Keywords: Software testability, software analytics, code coverage, software metrics, machine learning.

1. Introduction

Software testability metrics mainly evaluate the degree to which a software artifact (i.e., a software component, system, or requirements) supports its testing in a test context [1]. If the software artifact's testability is high, then finding faults in the system (if it has any) through testing is more comfortable. Since software testing is an undecidable problem [2], it is time-consuming and expensive to achieve fault-free programs. Testability gives an outlook on how much software testing will help find defects before any testing is performed. However, measuring testability is challenging due to various subjective definitions [3].

According to a recent study, there is no clear guidance on measuring testability and how to handle relevant issues such as quantifying [3]. Many formulations and metrics for measuring testability have been proposed regardless of the actual software testing conditions and outputs [4], [6]. Most of them are devoted to measuring implementation (source code) testability [5]. Despite many studies on software testability, we observed that the relationship between testability and test adequacy criteria had not been studied when automated software testing tools are being applied. Measuring and predicting testability helps us explorer the current automated testing obstacles and issues. Before a program could be tested thoroughly, it should be prepared for testing. Testability measurement could be applied to determine to what extent the test adequacy criterion could be satisfied. The low testability means that the program is not ready for testing and should be refactored to improve testability. Non-testable software artifact will increase the cost and time of testing while decreasing the possibility of finding defects [7].

The fact that testability must directly point to the ease of software testing pushes us to propose a practical approach to measure and predict testability by utilizing the testing's outcomes. The starting point to determine the relationship between testability and software metrics is testing effort (TE). In 1972, Edsger W. Dijkstra made the seminal quote: *"Program testing can be used to show the presence of bugs, but never to show their absence!"* [8]. As a result, test adequacy criteria, such as code coverage, have emerged to measure testing quality [2]. An adequate test suite is one that provides enough test data to ensure the correctness of the software under test (SUT) by satisfying a given adequacy criterion. Considering the test adequacy, the testing effort can be defined as the effort (budget) required to find and execute a test suite satisfying a given adequacy criterion. Hence testability of a component, X, is approximately equaled to the

required testing effort, $RTE(X)$:

$$T(X) \propto RTE(X) \quad (1)$$

The required effort, including the test data generation and test execution, is directly related to the percentage of the adequacy criterion, C(X), satisfied by that effort, i.e.:

$$RTE(X) \propto C(X) \quad (2)$$

Since no test suite may exist to satisfy a given criterion on a particular program, the required effort may lead to infinite, and it is unmeasurable. For instance, the path coverage criterion may not be fully satisfied with programs containing loop constructs due to the infinite number of paths or existing infeasible paths. The testing budget restricts the effort of testing in the case of an unsatisfiable adequacy criterion; therefore, only the portion of the adequacy criterion will be satisfied under these conditions. The transitive property for relations (1) and (2) results:

$$T(x) \propto C(X) \quad (3)$$

Software testability can be defined as the extent to which a test adequacy criterion could be satisfied after testing. Connecting runtime information, e.g., the percentage of satisfied test adequacy criterion, to the static properties of the program, e.g., source code metrics, is a key point to measure testability. To this aim, a machine learning approach is used on a large dataset of experimental data. Our methodology works in two steps: First, the code coverage of real-world software projects is obtained by performing automated testing. Second, a set of metrics belong to each software map to the corresponding coverage information through a machine learning approach. The resultant model is then used to predict the testability of the Software Under Test (SUT) before applying automatic test data generation tools and helps the developer know about accessible code coverage. In summary, the following contributions are proposed in this article:

1. To introduce a new definition for software testability based on the test adequacy criteria, which can use to estimate the testability in automated testing.
2. To present a machine learning approach for measuring and predicting software testability based on our new definition. The learned model classifies the testability of SUT into five different categories: very low, low, mean, high, and very high.
3. To automatically designate the important source code metrics affecting testability using machine learning feature importance analysis.
4. To perform an extensive study on software testability at the source code level and provide a large experimental dataset relevant to testability.

We evaluated our approach on a large corpus of Java open-source projects containing more than 23,000 Java classes using five different machine learning models and 280 software metrics as features. Our result demonstrates the feasibility of learning testability based on software metrics with an accuracy of 81.94%. Our dataset and replication package are publicly available on https://m-zakeri.github.io/ADAFEST/.

The rest of this paper is organized as follows. In Section 2, related works are reviewed and discussed. Section 3 describes the proposed methodology. Experimental evaluations and results are explained in Section 4. Threats to the validity of our proposed approach and experiments are discussed in Section 5. Finally, the conclusion and feature works are discussed in Section 6.

2. Related Works

Many definitions for software testability have been offered in the literature [3]. Several definitions have been proposed by IEEE and ISO [9], [10]. Our work is mainly related to the definition by ISO standard 25010:2011 [10]: "degree of effectiveness and efficiency with which test criteria can be established for a system, product or component and tests can be performed to determine whether those criteria have been met." However, definitions do not imply how to measure testability. Hence many formulations have been devised to measure testability. Most of them are based on software metrics.

Shaheen and Bousquet [5] have classified testability metrics into *scope* and *complexity* metrics. Scope metrics predict how given testing strategies require many test cases. Complexity metrics predict how difficult it will be to generate test cases. It is challenging to distinguish the boundary between these two categories. For instance, the cyclomatic complexity can fall into both categories. Meanwhile, they have concluded that testability metrics have been rarely validated, neither formally nor empirically.

Bruntink [11] has proposed a metric-based approach to predict the size of the test suite required for a Class. Experiments have been performed on two software systems and have considered testability from the perspective of the unit testing at the class level. The results show that none of the metrics is significantly more correlated to the test suite size than the relatively simple lines of code per class (LOCC) metric.

Badri et al. [12] have tried to measure the effect of clone refactoring on the size of unit test cases in object-oriented software. They used different software metrics to quantify the considered source code attributes and the size of unit test cases. Linear regression analysis and five machine learning algorithms have been used to develop predictive and explanatory models. They do not consider the metric impacts on test adequacy criteria and program runtime information. An empirical study has been performed only on two open-source Java software systems.

Khan et al. [6] have proposed a metrics-based model for object-oriented design testability (MTMOOD). They have established a relationship between three design metrics (encapsulation, inheritance, and coupling) and testability through the linear regression based on investigating three medium-size projects manually. Equation 4 summarizes their computational formula for testability, where encapsulation's metric count of all the methods defined in a class, inheritance's metric count of the number of class hierarchies in the design, and coupling count of the different number of classes that a class directly depends on.

$$Testability = -0.08 \times Encapsulation + 1.12 \times Inheritance + 0.97 \times Coupling \quad (4)$$

Controllability and observability concepts have been explored broadly for measuring software testability [3], [13]. COTT [14] provides a framework that helps the developer to instrument object-oriented software to build the required controllability and observability. COTS needs the SUT to be instrumented and executed to gather the controllability and observability information, which is exhausting for large source code. Runtime testability is defined as the maximum coverage that can be achieved by executing tests. In [15], a formulation based on the component interaction graph [16] has been proposed based on a set of operations that can be covered from the input. No solid empirical evidence exists for the presented runtime testability metrics.

Despite a large number of researches on software testability, we observed that the relationship between testability and test adequacy criteria had not been studied when automated software testing tools are being applied. Testability has severe impacts on the result of automated testing. [17], [18] have been studied the effect of code visibility on code coverage in both manual and automatic testing. Results demonstrate that developer-written tests are insensitive to code visibility; however, automatically generated tests yield lower code coverage on less visible code. Only the access level of methods is investigated in [17]. Establishing a relationship between other metrics rather than visibility for measuring and predicting testability helps us explore the current automated testing obstacles and issues. Machine learning techniques are suitable for this purpose.

Machine learning is about developing the required software that automatically analyses data for making predictions, categorizations, and recommendations. Recently, machine learning has been applied to test data generation [19], and code smells detection [20], [21]. This paper uses machine learning to predict the source code coverage as an axiomatic proxy to measure testability.

3. Methodology

3.1 Framework

First, we introduce our testability learning and prediction framework. The proposed framework consists of five steps. Figure 1 shows the workflow of the testability measurement. We explain each step-in detail:

1. **Gathering software repositories:** At the first step, we need a large set of source codes to examine their testability. Open-source software repository holders such as *GitHub* and *SourceForge* can be used to access the various software source codes. In our experiments, we used SF110 corpus [22], a collection of more than 23K classes from 110 different Java open-source projects on SourceForge.
2. **Computing code coverage:** At the second step, we measure the code coverage of different codes with different programming styles to understand the runtime behavior of a specific code snip precisely. Our goal is to understand the testability of the software for testing with automatic test data generation tools, such as EvoSuite [22] and JDART [23]. In our experiments, we run EvoSuite on the SF110 to generate and execute test data.
3. **Computing software metrics:** This step computes the software metrics by static analysis of the source code of each project within the repository. The aim is to understand the impact of different code metrics on testability. To this aim, we provided an extended list of software metrics, including those discussed in the literature and those developed in this paper. Table 1 shows each metric and its definition. The value of these metrics for each entity and the runtime information, i.e., code coverage, form a dataset.
4. **Training:** In this step, a relation between metrics as static properties of each entity and code coverage as a measure for testability is extracted by training a machine learning model on our dataset. The model is then used to predict testability in the term of code coverage. More coverage means that the class is more testable when using an automatic test data generation tool.
5. **Testing:** After building the model, it should be tested on a set of previously unseen data to ensure its correctness and effectiveness. The test set consists of a portion of the dataset that has not participated in the training process.

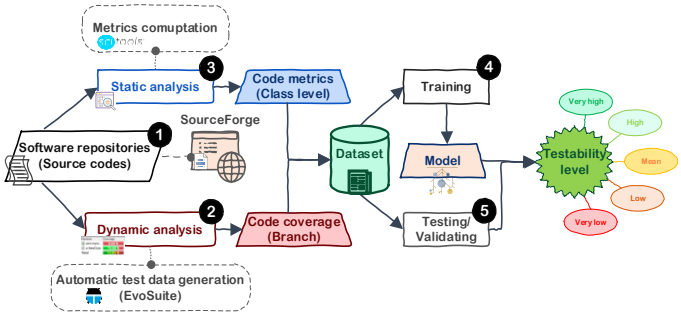

Figure 1. Testability measurement workflow

3.2 Code Coverage and Testability Levels

Code coverage is the main test adequacy criterion when the artifact of testing is source code [2]. It is defined at the level of statements, branches, and path. Statement coverage criterion enforces that all statements of the software under test (SUT) must be executed at least once. The branch coverage criterion requires that all branches of the SUT are executed at least once, and the path coverage criterion requires that each of the possible paths through the program is executed. Path coverage subsumes branch coverage and statement coverage, i.e., path coverage satisfaction guarantees the satisfaction of the other two criteria [2]. However, it cannot apply to large systems due to the infinite number of paths. Branch coverage is the most practical test adequacy criteria to be used in practice. None of the coverage criteria may fully satisfy during testing with automated testing tools. The percentage of the

covered criterion is reported as the main factor to measure the efficiency of testing.

As we described in Section 1, testability is propositional to the percentage of test adequacy criteria. To propose a framework for testability measurement based on this definition and to evaluate it empirically, we consider the branch coverage of the SUT to be used as the runtime metric. To this aim, the value of branch coverage is discretized to five bins (intervals with equal lengths) to predict the testability. Figure 1 shows the testability levels based on the percentage of the branch coverage. The reason behind discretization is that it is more developer-friendly to report testability in qualitative terms. Another technical reason is that predicting the exact value of testability with machine learning models is too tricky since no two software entities have exactly equal testability. Discretization mitigates the sparsity of testability values.

EvoSuite [24], an automatic test-suite generation tool, is applied to compute branch coverage. We chose EvoSuite because it is a state-of-the-art tool in testing Java classes. EvoSuite tries to maximize code coverage with an evolutionary algorithm. Due to the random nature of evolutionary algorithms, the result of different executions may not be the same. To ensure the reliability of the result, we repeated the test data generation process five times and took the average coverage obtained in all executions as a target for learning.

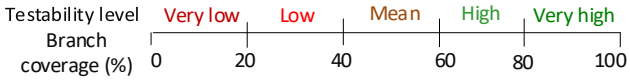

Figure 2. Discretizing branch coverage to testability bins.

Table 1. Source code metrics used for testability prediction

Quality attribute	Metric	Full name	Class (CS)	Package (PK)
Size / Count	LOC	Line of code	X	X
	NOST	Number of statements	X	X
	NOSM	Number of static methods	X	X
	NOSA	Number of static attributes	X	X
	NOIM	Number of instance methods	X	X
	NOIA	Number of instance attributes	X	X
	NOM	Number of methods	X	X
	NOMNAMM	Number of not accessor or mutator methods	X	X
	NOCON	Number of constructors	X	X
	NOP	Number of parameters	X	—
	NOCS	Number of classes	—	X
	NOFL	Number of files	—	X
Complexity	CC	Cyclomatic complexity	X	X
	CCS	Cyclomatic complexity strict	X	X
	CCE	Cyclomatic complexity essential	X	X
	CCM	Cyclomatic complexity modified	X	X
	NESTING	Nesting level of control constructs	X	X
	PATH	Number of unique paths	X	—
	KNOTS	Measure of overlapping jumps	X	—
Cohesion and Coupling	LOCM	Lack of cohesion in methods	X	—
	CBO	Coupling between objects	X	—
	RFC	Response set for a class	X	—
	FANIN	Number of incoming invocations	X	—
	FANOUT	Number of outgoing invocations	X	—
	DEPENDS	All dependencies of the class	X	—
	DEPENDSBY	Entities depended on the class	X	—
	ATFD	Access to foreign data	X	—
	CFNAMM	Called foreign not accessor or mutator methods	X	—
	DAC	Data abstraction coupling	X	—
	NOMCALL	Number of method calls	X	—
Visibility	NODM	Number of default methods	X	X
	NOPM	Number of private methods	X	X
	NOPRM	Number of protected methods	X	X
	NOPLM	Number of public methods	X	X
	NOAMM	Number of accessor methods	X	X
Inheritance	DIT	Depth of inheritance tree	X	—
	NOC	Number of children	X	—
	NOP	Number of parents	X	—
	NIM	Number of inherited methods	X	—
	NMO	Number of methods overridden	X	—
	NOII	Number of implemented interfaces	X	—
	NOI	Number of interfaces	—	X
	NOAC	Number of abstract classes	—	X

3.3 Source Code Metrics

Software metrics are used to quantize different subjective aspects of software during the software development lifecycle (SLDC).

Source code is the most crucial software artifact, and source code metrics are a kind of software product metrics focused on measuring the quality of the software. We collected an extensive set of source code metrics, including C&K metrics [25], [26], HS metrics [27], QMOOD metrics [28], and MTMOOD metrics [6], and computed them for each project in the SF110 corpus [22]. A complete list of source code metrics has been shown in Table 1.

For better understanding, metrics were categorized based on two aspects: *Granularity* level and *quality attribute.* The granularity level of each metric represents the position of the entity for which the metric is defined in a hierarchical structure. This structure of object-oriented applications includes method, class, package, and project. The granularity of source code entities respectively increases from fine-grain entities, i.e., methods, to a coarse-grain entity, i.e., project. The *minimum (MIN)*, *maximum (MAX)*, *mean (AVG)*, *standard deviation (SD)*, and *logarithm (LOG)* of method-level metrics are used as the class and package-level metrics. The quality attribute outlines the relevant quality aspect of a metric. We considered six object-oriented well-known quality attributes, including size, complexity, cohesion, coupling, visibility, and inheritance, to put each metric on a more abstraction level.

In addition to metrics in Table 1, we defined and computed a new set of metrics, called *lexicon metrics*, which capture the lexical properties of each class source code, e.g., the number of identifiers and the number of operators used in the class. The goal is to study the impact of program lexical properties in automated unit testing. Lexicon metrics are listed in Table 2. For the purpose of our study, we computed lexicon metrics only for classes in a program while they can be computed for other entities, i.e., method, package, and project.

We considered every source code metric related to testability and discarded metrics, which could not affect testability when using automatic testing frameworks. Excluded metrics are contained the number of blank lines, number of comment lines, and comment to code ratio. In total, we extracted 280 source code metrics from each Java class in our repository.

Table 2. Lexicon metrics and their definitions

Metric abbreviation	Metric name and description
CSNOTK	Class number of tokens
CSNOTKU	Class number of unique tokens
CSNOID	Class number of identifies
CSNOIDU	Class number of unique identifiers
CSNOKW	Class number of keywords
CSNOKWU	Class number of unique keywords
CSNOASS	Class number of assignments
CSNOOP	Class number of operators without assignments
CSNOOPU	Class number of unique operators
CSNOSC	Class number of semicolons
CSNODOT	Class number of dots
CSNOREPR	Class number of return and print statements

3.4 Data Representation

When both metrics and code coverages are computed, a dataset for the machine learning task can be constructed. For each class in each project, source code metrics and code coverage are extracted. Each row of the dataset represents a class instance, and each column represents a metric as an attribute of that class. The last column is a nominal label that expresses the testability level of the corresponding class.

For each instance, the metrics of the respective containers are included in the features. The containment relation defines that a method is contained in a class, a class is contained in a package, and a package is contained in a project. The inclusion of the metrics of containers allows exploiting the interaction (if existing) among the features of the classified element and the ones of its containers [20]. Evaluating an entity in isolation may lead to a false misunderstanding of the structure and behavior of that entity. To overcome this issue, we consider a vector for each entity consisted of an entity's containers. For a method, the context vector contains metrics of its enclosing project, package, and class. For a class, it contains metrics of its enclosing project and package. For the scope of this work, we only considered one label and one type of instance, i.e., classes. Future work can be focused on method testability and other test adequacy criteria.

3.5 Data Preparation

Data preprocessing should be applied before building any machine learning model to ensure the applicability of sample instances. Two main proposes of preprocessing are addressing imbalanced data and removing outlier data, which avoids a good training process. The process of the data preparation has been illustrated in Figure 3. Our data processing consists of data balancing and data cleaning, as described in the following sections.

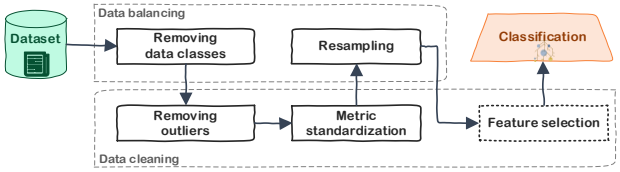

Figure 3. Preprocessing workflow

3.5.1 Data Balancing

We observed that the number of classes with very high branch coverage is more than the number of classes with all remaining levels. It reveals that most classes have sufficient testability such that EvoSuite can almost cover all of their branches. However, the low frequency of untestable classes results in an *imbalanced dataset*, a common problem in machine learning, which makes it hard to achieve precise predictive models for testability.

Manual investigation of classes with full coverage illustrates that most of these classes contain trivial codes with low complexity. Such classes are prevalent even in large-scale and complex software projects. For example, class `AddUserFieldsBeanInfo` in WEKA [29], a leading data mining software, has only five lines of code with cyclomatic complexity one, as shown in Figure 4.

To mitigate the problem of imbalanced data, we removed all instances with trivial codes, including data classes, those that only have attributes and accessor methods [20], and fully covered class with at most one branch. We also applied *SMOTEENN* algorithm [30] to oversample classes with few samples in our trainset after data cleaning.

```
1  public class AddUserFieldsBeanInfo extends SimpleBeanInfo {
2
3    /**
4     * Get the bean descriptor for this bean
5     *
6     * @return a <code>BeanDescriptor</code> value
7     */
8    @Override
9    public BeanDescriptor getBeanDescriptor() {
10     return new BeanDescriptor(
11       weka.filters.unsupervised.attribute.AddUserFields.class,
12       weka.gui.filters.AddUserFieldsCustomizer.class);
13   }
14 }
```

Figure 4. Example of Java class in a large software project, WEKA, with trivial code which is fully covered by EvoSuite tests.

3.5.2 Data Cleaning and Scaling

Besides addressing the imbalanced data issues, we performed typical preprocessing activities, including outlier elimination, feature set standardization, and feature selection, to prepare our dataset for use in the testability learning task.

To remove outliers, we dropped instance class C, where $\exists f \in F\ s.t. |Z_{score}\ f(C;\ S)| \geq 3$. F is a set of all features in the dataset, i.e., set of all metrics, $f(C)$ denotes the corresponding value of metric f for class C, and S is the set of all samples in the dataset. Z-score is a statistical concept that expresses how many standard deviations away a data point is from the mean. Z_{score} for each instance, x, of a statistical sample, S, is computed as:

$$Z_{score}(x;\ S) = \frac{x-\bar{x}}{s_x} \qquad (5)$$

where $\bar{x}$ and s_x are the mean and standard deviation of the sample, respectively. In a normal distribution, a standard cut-off value for finding outliers is Z-scores of ± 3.

To standardized features, we scaled each feature's values in the $[0, 1]$ range via *MinMax* scaling. Each value x of feature f is transformed into x_{std} by the following formula:

$$x_{std} = \frac{x-x_{min}}{x_{max}-x_{min}} \qquad (6)$$

where $x_{max}\ and\ x_{min}$ are the maximum and minimum values of feature f, respectively. Source code metrics typically have very different ranges, and some classification algorithms, such as neural networks (MPLs), are highly sensitive to the range of independent variables. To address this issue, metrics' values were scaled into the same range by the above equation.

Finally, we used the *CfsSubsetEval* algorithm [29] for feature selection. It evaluates the worth of a subset of attributes by considering the individual predictive ability of each feature along with the degree of redundancy between them. In our experiments, we considered both possible datasets: a dataset with all features and a dataset with the selected feature and reported the evaluation metrics for them to discover the effect of automatic metric selection on predicting testability.

3.6 Machine Learning Algorithms

There are many machine learning algorithms, and none of them the best for all tasks. Therefore, a set of machine learning algorithms was selected from two different well-known families, tree-based and functional-based techniques, to establish the predicting model for testability:

- **Decision Tree (DT)** [31] works based on selecting the most informative features to split instances with different classes from each other. C4.5 is the most widely used version of the decision tree in practice. It utilizes the concept of information entropy to select the root nodes for each subtree. Decision trees are human interpretable and easy to understand classifiers. However, it is unstable, i.e., a small change in the data may lead to a significant change in the structure of the optimal decision tree.
- **Random Forest (RF)** [32] is an ensemble learning method that constructs a multitude of decision trees on bootstrapped data. Each tree is built on a subset of original data, and features result in a more robust classifier than the Decision Tree. In testing time, each tree votes or selects the class, and the class receiving the most votes by a simple majority is the predicted class. Random Forest often performs well out-of-the-box; however, it needs more time and resources at both the training and testing phases.
- **Multilayer Perceptron (MLP)** [33] is a class of feed-forward neural networks with a deep architecture that consists of multiple layers of computational units interconnected in a directed acyclic graph (DAG). MLP builds a function for mapping input to output and works well in regression tasks. Features are fed to the first layer of the network, and the result is generated in output, where the actual output is expected to generate. The weight of each connection in DAG is learned during the training.

We provided a boosted version of the decision tree and random forest classifiers by applying the *AdaBoost* technique as a meta-learning algorithm [29]. AdaBoost increases the weight of incorrectly classified instances by a base learner in a sequential manner such that subsequent classifiers focus more on severe cases. The final classification result depends on the weights assigned to the input instances.

The grid-search strategy with cross-validation is employed to find the optimal hyperparameters of the model. Using this way, we ensure to select the best model with optimum hyperparameters. The model is then used to predict testability before testing could be done. If the model predicts the low testability, testing can be postponed until the SUT is prepared for testing. Our proposed method reduces the time and cost of inappropriate testing by preventing it from being performed.

4. Evaluations

In this section, we report the result of our experiments with the proposed methodology on the provided datasets. We define and discuss two research questions about the machine learning testability measurement:

RQ1: What is the effectiveness of the learned model in predicting testability, and what is the best machine learning algorithm?

RQ2: Which aspects of software (in the term of lexical and source code metrics) have a more significant impact on the testability of automated testing?

RQ3: Which software quality attributes mostly impact the testability of automated testing?

4.1 Experimental Setup

All experiments were performed on Windows 10 (x64) machine with 2.6 GHz Intel® Core™ i7 6700HQ CPU and 16GB of RAM. The creation of the primary dataset on this machine took about 30 hours. To study the impact of data preprocessing on the efficiency of predictive models, we constructed three additional datasets (DS2 to DS4) with different schemes to use in our experiments. Table 3 summarizes prepared datasets. DS1 is the original dataset without any preprocessing. In DS2, all metrics have been standardized by applying the *MinMaxScaler* in the Scikit learn [34]. DS3 contains original data, while irrelevant and redundant metrics have been eliminated by applying the *CfsSubsetEval* algorithm [29]. DS4 is the result of performing both the standardization and feature selection preprocessing.

Table 3. Datasets used in our experiments

Dataset	Preprocessing	Number of metrics	Range
DS1	Default	280	[0, +∞)
DS2	Standardization	280	[0, 1]
DS3	Feature selection	20	[0, +∞)
DS4	Feature selection + standardization	20	[0, 1]

4.2 Evaluation Metrics

For each classifier, we construct the confusion matrix and compute the following metrics to report and compare the efficiency of testability predictive models:

Accuracy. The accuracy for each testability level is calculated as the following equation:

$$ACC = \frac{TP + TN}{TP+FP+TN+FN} \quad (7)$$

Accuracy is not sufficient for evaluating a model. It may give misleading information, mainly when the dataset consists of imbalanced classes. In this case, a model that has been biased to the majority class takes better accuracy, while it is not desirable. Therefore, other metrics must be used to evaluate a model correctly.

F-measure (F_1 Score). The F-measure is the harmonic mean of the precision and recall and calculated as:

$$F_1 = 2 \times \frac{PPV \times TPR}{PPV + TPR} \quad (8)$$

where PPV is the positive predictive rate known as precision that determines the fraction of relevant instances among all retrieved instances, and TPR is the true positive rate known as recall or sensitivity that determines the fraction of all relevant instances that were retrieved by the model. Precision and recall are given by the following equations:

$$PPV = \frac{TP}{TP+FP} \quad (9)$$

$$TPR = \frac{TP}{TP+FN} \quad (10)$$

F-measure is typically used for evaluating a binary classifier. We used macro-averaging as arithmetic means of class-wise F-scores to obtain the final score for the model. In macro-averaging, the F-measure of each class is computed, and then the mean of all classes' scores is reported as F-measure.

Area Under ROC (AUC). The receiver operating characteristic (ROC) curve is obtained by plotting the TPR against the false positive rate (FPR) at various discrimination threshold settings used for selecting and reporting the final class. The goal of plotting this curve is to maximize true positives while minimizing false positives and is utilized to select the most suitable threshold. The top left corner of the plot is the ideal point in which FPR is zero and TPR is one. A larger area under this curve, shown by AUC, means the better discrimination power of a classifier. AUC for a good classifier is close to value one, while in the worst case, e.g., random classification, the AUC value is approximately 0.5. Similar to F-measure, AUC originally is computed for a binary classifier. Like for F-measure, the macro-averaging could be applied to compute the AUC for the multi-label classification model by computing the mean of AUC of all classes.

4.3 Learning Effectiveness

We measured the effectiveness of each learning algorithm in terms of evaluation metrics on each dataset. Each model was trained in its best configuration obtained by the grid search optimization through cross-validation, as discussed in Section 3.6, and then tested on 20% of remained data uses as the testing set.

The hyperparameters which were considered to be optimized for each model along with the result of the grid search optimization process, i.e., best hyperparameters founded during the cross-validation, are shown in Table 4. We used *balanced accuracy* provided by scikit-learn as a metric for scoring and ranking the performance of each model. The model selection was separately performed on all datasets in Table 3.

Table 5 shows the accuracy, F-measure, and AUC of predictive models. Boosted versions of RF and DT have been prefixed with a capital B letter. The best value for each evaluation metric has been bolded. Random Forest with AdaBoost on DS4 has the best performance with an accuracy of 81.94% and F-measure 82.49%. The maximum difference in accuracy for all models on all datasets is 30%. Compared with other machine learning tasks in software engineering, such accuracy means that learning to predict testability is a difficult problem and needs more data or more specific techniques. For example, machine learning accuracy for code smell detection is about 98.0% [20]. Tree-based algorithms, i.e., DT and RF, have better accuracy in comparison to MLP on most datasets. Feature selection (DS4) results in better accuracy in almost all models.

Table 4. Models' configurable parameters and results of hyperparameter tuning

Classifier	Parameters and searching values	Best value found in experiments
DT	max_depth: *range (1, 100, 2)*	*73, 71, 95, 47*
	class_weight: *{None, balanced}*	*None, None, None, balanced*
	criterion: *{gini, entropy}*	*gini, entropy, gini, gini*
	min_samples_split: *range (2, 100, 2)*	*2, 2, 2, 2*

RF	n_estimators: range (100, 2000, 10)	*400, 1600, 1600, 600*
MLP	hidden_layer_sizes: {(1024, 512, 256, 128), (512, 256, 128, 10), (512, 512, 256), (256, 256, 128), (512, 128, 100), (944, 472, 236), (472, 236, 59)}	*(512, 512, 256), (256, 256, 128), (256, 256, 128), (944, 472, 236)*
	activation: *{logistic, tanh, relu}*	*tanh, tanh, tanh, tanh*
	solver: *{adam, sgd}*	*adam, adam, adam, adam*
	learning-rate: *{constant, adaptive}*	*constant, constant, constant, constant*
	epoch: *range (100, 1500, 100)*	*100, 300, 200, 400*
AdaBoost	n_estimators: *range (100, 1000, 100)*	Different for each model and each dataset

Table 5. The performance of testability predictive models

Model	Measure	DS1	DS2	DS3	DS4
MLP	ACC (%)	27.21	81.90	76.89	80.80
	F1 (%)	30.47	82.18	77.48	81.13
	AUC	0.5827	0.8891	0.8680	0.8836
DT	ACC (%)	67.76	66.45	65.08	71.53
	F1 (%)	68.11	66.67	65.28	71.26
	AUC	0.7610	0.7527	0.7442	0.7846
RF	ACC (%)	81.43	80.22	77.76	81.57
	F1 (%)	82.02	80.89	78.71	82.19
	AUC	0.8935	0.8926	0.8827	0.8917
B-DT	ACC (%)	69.12	69.42	65.68	71.65
	F1 (%)	69.08	69.33	66.28	71.35
	AUC	0.7695	0.7714	0.7480	0.7853
B-RF	ACC (%)	81.30	80.92	77.64	**81.94**
	F1 (%)	81.93	81.47	78.65	**82.49**
	AUC	0.8935	0.8924	0.8831	**0.8938**

Figure 5 shows the confusion matrix for the best classification model, i.e., B-RF on DS4. It can be observed that classes with *very low* coverage have been classified as *low* coverage in 18 samples, which means that distinguishing between low and very low coverage levels in the current dataset is more challenging than other levels. Increasing the number of samples with very low coverage, i.e., finding non-testable codes, can mitigate this error and improve accuracy. Overall, the B-RF model has performed very effective on DS4 and predict the coverage of almost all instance correctly.

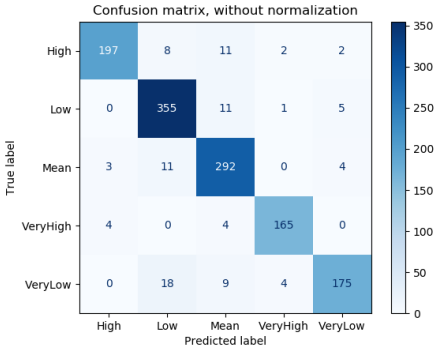

Figure 5. Confusion matrix for Boosted Random Forest (B-RF) model on the DS4 testing set.

Answer to RQ1: *The learning effectiveness of predicting models is good enough to be used in practice for measuring testability. The best machine learning model is Boosted Random Forest (B-RF), with 81.94% accuracy.*

4.4 Important Testability Metrics

One important application of testability learning is to automatically designate important software metrics affecting testability. Previous works have specified various software metrics that may affect testability manually without solid empirical evidence. We use the concept of feature importance analysis in machine learning to answer RQ2 in a more reliable scheme. To this aim, we compute the importance of each metric for the best model, B-RF, on the best dataset, DS4.

We employ two well-known feature importance analysis techniques. The first one is tree importance or mean decrease in impurity (MDI), which counts the times a feature is used to split a node, weighted by the number of samples it splits. It is calculable for tree-based learning models such as random-forest based on both the Gini index or Entropy. The second technique is the permutation importance technique [32]. In this technique, the values of a single feature are shuffled, then a learned model is asked to make predictions using the resulting dataset. Using these predictions and the actual target values determine how much the model suffered from shuffling. That performance deterioration denotes the importance of the shuffled feature. The permutation process for each feature was repeated 30 times to alleviate the effects caused by the randomness nature of permutation.

Figure 6 shows the metric importance based on tree-based feature importance and permutation importance [34]. The importance score for each metric specifies its impact on the accuracy of the model. For instance, the permutation importance in Figure 6.b indicates that permuting a metric

drops the model accuracy by at most 0.6%. We can conclude that these software metrics are correlated together even after applying feature selection. Therefore, we perform a more in-depth analysis to determine the most important metrics affecting testability.

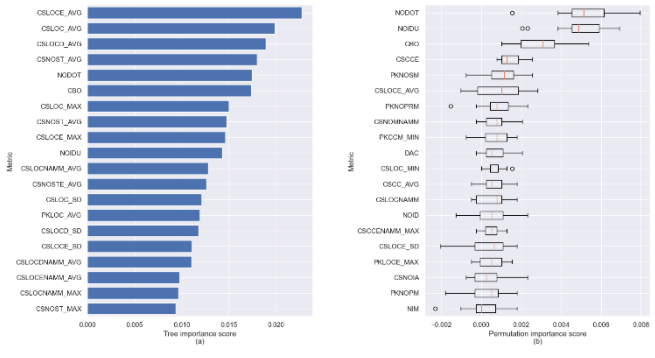

Figure 6. Essential software metrics for testability prediction. (a) tree-based feature importance, and (b) permutation importance.

Figure 7.a illustrates a correlation matrix of the top 20 important software metrics used for testability prediction. As shown in the figure, the correlation between most pairs of metrics is too high (close to 1). Figure 7.b shows the dendrogram of agglomerative clustering of code metrics based on their correlation. Using such a hierarchical clustering helps us identify important software metrics which are less correlated together.

We can obtain five clusters of software metrics by picking the clustering threshold of two for agglomerative clustering, which is appropriate to determine the most independent important metrics. For each cluster, the metric with the highest permutation importance score is selected as the representative member of that cluster. The top software metrics affecting testability are Number of Dots (NODOT), Package Number of Static Methods (PKNOSM), Minimum Package Cyclomatic Complexity Modified (PKCCM_MIN), Data Abstraction Coupling (DAC), and Minimum Class Line of Code (CSLOC_MIN).

We computed the Pearson correlation coefficient between these metrics and branch coverage of classes under test in our dataset. We observed that all of these metrics negatively impact testability. It means that increasing the value of each metric decreases testability and vice versa when applying automated test data generation algorithms.

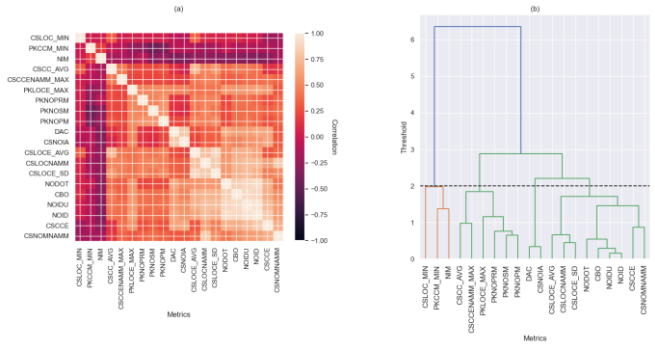

Figure 7. Relationship between the top 20 essential testability metrics. (a) correlation matrix, and (b) source code metrics clusters.

Answer to RQ2: *Software metrics, including the number of dots, number of static methods, cyclomatic complexity, data abstraction coupling, and lines of code, significantly impact the testability of class in automated testing.*

4.4 Important Quality Attributes

To answer RQ3, we determined the quality subject of each important metric in Figure 6 based on quality attributes mentioned in Table 1. Figure 8 illustrates the radar diagram of quality attributes participants in determining testability. Each dimension of the radar diagram indicates a quality attribute. Each point the number of software metrics that belong to the corresponding quality attributes.

It is observed that size and complexity are the most crucial quality attributes which affect testability. 12 out of 20 important metrics belong to the size, and 4 out of 20 metrics belong to the complexity. Cohesion and coupling are less important than size and complexity. However, they are more important than visibility and inheritance.

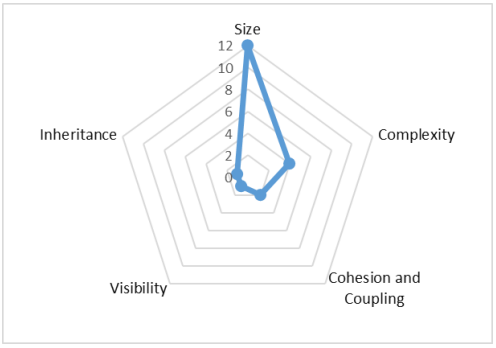

Figure 8. Important quality attributes affecting testability

Answer to RQ3: *Size and complexity are the most important quality attributes affecting the testability of the class under test in automated testing. On the other hand, visibility and inheritance are less related to testability.*

5. Threats to Validity

We explain threats to the validity of our methodology and experiments and the efforts we made to reduce them. These threats are related to construction validity, internal validity, and external validity of the proposed approach.

Threats to construct validity are about how we define our model of testability. We use the branch coverage of class under test as an axiomatic metric to measure software testability. In a recent study, Salahirad et al. [35] have shown that branch coverage is the most influential criterion which can be used as a fitness function in search-based testing. Meanwhile, the other coverage metrics must be investigated

for testability measurement.

The most significant threats to internal validity are the stochastic nature of both the search-based test data generation and the machine learning algorithms used in testability prediction. We repeated the test suite generation process five times for each class in our dataset, using default parameters with different random seeds to ensure the reliability outputs of EvoSuite. For the machine learning models, we select the best hyperparameters using grid search with cross-validation to ensure the reliability of our models when they are used in production. Other machine learning models can be train and test in future works.

The main threat to external validity is related to the generalization capability of our approach in predicting the testability of other programs. We use a large corpus of 110 Java open-source projects with more than 23,000 classes to ensure the generalization of our results. However, it still is required to evaluate this approach in other programming languages since testability is an inherent feature of the software. It mainly depended on the programming language of the software under test.

6. Conclusion

The proposed approach in this paper aims at making a relationship between static properties of the source code and runtime coverage information obtained by the executing test to predict software testability. We leverage the machine learning classification technique to build testability prediction models for Java programs. The main research question in our study is how the source code metrics affect the degree of code coverage before the software actually is tested. Our experiments with a wide range of machine learning techniques, including tree-based models and functional-based models, indicate a tight relationship between software metrics and branch coverage. We confirm this hypothesis by accessing the accuracy of 81.94% in predicting testability with machine learning models on a set of 110 Java open-source projects.

Testability prediction helps early identification of software components which are required more testing effort. The next step is connecting testability to software smells and discover the relationship between these two concepts in software engineering. There is no such study on this topic in the literature to the best of our knowledge, and we are planning to perform it as future work.

References

[1] J. M. Voas and K. W. Miller, “Software testability: the new verification,” *IEEE Software*, vol. 12, no. 3, pp. 17–28, May 1995.

[2] P. Ammann and J. Offutt, *Introduction to Software Testing*. Cambridge, UK: Cambridge University Press, 2016.

[3] V. Garousi, M. Felderer, and F. N. Kılıçaslan, “A survey on software testability,” *Information and Software Technology*, vol. 108, pp. 35–64, Apr. 2019.

[4] P. R. Suri and H. Singhani, “Object-oriented software testability (OOSTe) metrics analysis,” *International Journal of Computer Applications Technology and Research*, vol. 4, no. 5, pp. 359–367, 2015.

[5] L. D. B. M. R. Shaheen, “Survey of source code metrics for evaluating testability of object-oriented systems,” Inria France, 2014.

[6] R. A. Khan and K. Mustafa, “Metric based testability model for object-oriented design (MTMOOD),” *ACM SIGSOFT Software Engineering Notes*, vol. 34, no. 2, p. 1, Feb. 2009.

[7] F. Toure, M. Badri, and L. Lamontagne, “Predicting different levels of the unit testing effort of classes using source code metrics: a multiple case study on open-source software,” *Innovations in Systems and Software Engineering*, vol. 14, no. 1, pp. 15–46, Mar. 2018.

[8] E. W. Dijkstra, “Notes on structured programming,” Apr. 1970.

[9] ISO and IEEE, “Systems and software engineering — Vocabulary,” ISO, p. 511, 2017.

[10] ISO and IEC, “ISO/IEC 25010:2011 systems and software engineering — systems and software quality requirements and evaluation (SQuaRE) — system and software quality models,” ISO, p. 34, 2011.

[11] M. Bruntink, “Testability of object-oriented systems: a metrics-based approach,” University van Amsterdam, 2003.

[12] M. Badri, L. Badri, O. Hachemane, and A. Ouellet, “Measuring the effect of clone refactoring on the size of unit test cases in object-oriented software: an empirical study,” *Innovations in Systems and Software Engineering*, vol. 15, no. 2, pp. 117–137, Jun. 2019.

[13] R. Sharma and A. Saha, “A systematic review of software testability measurement techniques,” in *2018 International Conference on Computing, Power and Communication Technologies (GUCON)*, 2018, pp. 299–303.

[14] A. Goel, S. C. Gupta, and S. K. Wasan, “COTT – a testability framework for object-oriented software testing,” *World Academy of Science, Engineering and Technology International Journal of Computer, Electrical, Automation, Control and Information Engineering*, vol. 2, pp. 4224–4231, 2008.

[15] A. Gonzalez-Sanchez, E. Piel, H.-G. Gross, and A. J. C. van Gemund, “Minimising the preparation cost of runtime testing based on testability metrics,” in *2010 IEEE 34th Annual Computer Software and Applications Conference*, 2010, pp. 419–424.

[16] Y. Wu, D. Pan, and M.-H. Chen, “Techniques for testing component-based software,” in *Proceedings Seventh IEEE International Conference on Engineering of Complex Computer Systems*, pp. 222–232.

[17] L. Ma, C. Zhang, B. Yu, and H. Sato, “An empirical study on effects of code visibility on code coverage of software testing,” in *2015 IEEE/ACM 10th International Workshop on Automation of Software Test*, 2015, pp. 80–84.

[18] L. Ma, C. Zhang, B. Yu, and H. Sato, “An empirical study on the effects of code visibility on program testability,” *Software Quality Journal*, vol. 25, no. 3, pp. 951–978, Sep. 2017.

[19] M. Zakeri Nasrabadi, S. Parsa, and A. Kalaee, "Format-aware learn&fuzz: deep test data generation for efficient fuzzing," *Neural Computing and Applications*, Jun. 2020.

[20] F. A. Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," *Empirical Software Engineering*, vol. 21, no. 3, pp. 1143–1191, Jun. 2016.

[21] F. A. Fontana and M. Zanoni, "Code smell severity classification using machine learning techniques," *Knowledge-Based Systems*, vol. 128, pp. 43–58, Jul. 2017.

[22] G. Fraser and A. Arcuri, "A large-scale evaluation of automated unit test generation using EvoSuite," *ACM Transactions on Software Engineering and Methodology*, vol. 24, no. 2, pp. 1–42, Dec. 2014.

[23] K. Luckow *et al.*, "JDart: A Dynamic Symbolic Analysis Framework," 2016, pp. 442–459.

[24] A. Arcuri, J. Campos, and G. Fraser, "Unit test generation during software development: EvoSuite plugins for Maven, IntelliJ and Jenkins," in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2016, pp. 401–408.

[25] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object-oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, Jun. 1994.[26] S. R. Chidamber, S. S. M. C. for, and C. F. Kemerer, *A Metrics Suite for Object Oriented Design*. Creative Media Partners, LLC, 2018.

[27] B. Henderson-Sellers, *Object-oriented metrics: measures of complexity*. Prentice-Hall, Inc., 1995.

[28] J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE Transactions on Software Engineering*, vol. 28, no. 1, pp. 4–17, 2002.

[29] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The WEKA data mining software," *ACM SIGKDD Explorations Newsletter*, vol. 11, no. 1, p. 10, Nov. 2009.

[30] G. E. A. P. A. Batista, R. C. Prati, and M. C. Monard, "A study of the behavior of several methods for balancing machine learning training data," *ACM SIGKDD Explorations Newsletter*, vol. 6, no. 1, pp. 20–29, Jun. 2004.

[31] T. M. Mitchell, *Machine Learning*, 1st ed. New York, NY, USA: McGraw-Hill, Inc., 1997.

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

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

[34] F. Pedregosa *et al.*, "Scikit-learn: machine learning in python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[35] A. Salahirad, H. Almulla, and G. Gay, "Choosing the fitness function for the job: Automated generation of test suites that detect real faults," *Software Testing, Verification & Reliability*, vol. 29, no. 4–5, Jun. 2019.

Morteza Zakeri received his Ph.D. in Software Engineering from the Iran University of Science and Technology (IUST) in 2023, following his M.Sc. degree in Software Engineering from the same institution in 2018. He is currently an Assistant Professor at the School of Computer Engineering, Amirkabir University of Technology (Tehran Polytechnic). His research focuses on software engineering and machine learning, with particular emphasis on leveraging machine learning techniques for software testing and refactoring.
Email: *zakeri@aut.ac.ir*

Saeed Parsa received his B.Sc. in Mathematics and Computer Science from the Sharif University of Technology, Iran, his M.Sc. degree in Computer Science from the University of Salford in England, and his Ph.D. in Computer Science from the University of Salford, England. He is an associate professor of Computer Science at the Iran University of Science and Technology. His research interests include software engineering, software testing, and debugging, and algorithms.
Email: parsa@iust.ac.ir (Author in Correspondence)