Received 26 April 2016; | Revised 6 June 2016; | Accepted 6 June 2016
DOI: xxx/xxxx

RESEARCH PAPER

Branch Coverage Prediction in Automated Testing [†]

Giovanni Grano | Timofey V. Titov | Sebastiano Panichella | Harald C. Gall

Department of Informatics, University of Zurich, Zurich, Switzerland

Correspondence
Giovanni Grano, Binzmhlestrasse 14, Zurich, Switzerland
Email: grano@ifi.uzh.ch

Summary

Software testing is crucial in continuous integration (CI). Ideally, at every commit, all the test cases should be executed and, moreover, new test cases should be generated for the new source code. This is especially true in a Continuous Test Generation (CTG) environment, where the automatic generation of test cases is integrated into the continuous integration pipeline. In this context, developers want to achieve a certain minimum level of coverage for every software build. However, executing all the test cases and, moreover, generating new ones for all the classes at every commit is not feasible. As a consequence, developers have to select which subset of classes has to be tested and/or targeted by test-case generation. We argue that knowing a priori the branch-coverage that can be achieved with test-data generation tools can help developers into taking informed-decision about those issues. In this paper, we investigate the possibility to use source-code metrics to predict the coverage achieved by test-data generation tools. We use four different categories of source-code features and assess the prediction on a large dataset involving more than 3'000 Java classes. We compare different machine learning algorithms and conduct a fine-grained feature analysis aimed at investigating the factors that most impact the prediction accuracy. Moreover, we extend our investigation to four different search-budgets. Our evaluation shows that the best model achieves an average 0.15 and 0.21 MAE on nested cross-validation over the different budgets, respectively on EvoSuite and Randoop. Finally, the discussion of the results demonstrate the relevance of coupling-related features for the prediction accuracy.

KEYWORDS:
Machine Learning, Software Testing, Automated Software Testing, Coverage Prediction

1 | INTRODUCTION

Software testing is widely recognized as a crucial task in any software development process [1], estimated at being at least about half of the entire development cost [2,3]. In the last years, we witnessed a wide adoption of *continuous integration* (CI) practices, where new or changed code is integrated extremely frequently into the main codebase. Testing plays an important role in such a pipeline: in an ideal world, at every single commit, every system's test case should be executed (*regression testing*). Moreover, additional test cases might be automatically generated to test all the new —or modified— code introduced into the main codebase [4]. This is especially true in a Continuous Test Generation (CTG) environment, where the generation of test cases is directly integrated into the continuous integration cycle [4]. However, due to the time constraints between frequent commits, a complete regression testing is not feasible for large projects [5]. Furthermore, even *test suite augmentation* [6], *i.e.*, the automatic generation considering code changes and their effect on the previous codebase, is hardly doable due to the extensive amount of time needed to generate tests for just a single class.

[†] This work is an extension of the conference paper presented at the MaLTeSQuE 2018 workshop.

As developers want to ensure a certain minimum level of branch coverage for every build, these computational constraints cause many challenges. For instance, developers have to select and rank a subset of classes for which run test-date generation tools, or again, allocate a search-budget (*i.e.*, time) to devote to the test generation per each class. Knowing *a priori* the coverage that will be achieved by test-data generation tools with a given search-budget can help developers in taking informed decisions to answer such questions: we give following some practical examples of decisions that can be taken exploiting such a prediction. With the goal to maximize the branch-coverage on the entire system, developers might want to prioritize the test-data generation effort towards the classes for which they know a high branch-coverage can be achieved. On the contrary, they would avoid to spend precious computational time in running test-case generation tools against classes that will never reach a satisfying level of coverage; for these cases, developers will likely manually write more effective tests. Similarly, knowing the achievable coverage given a certain search-budget, developers might be able to allocate such a budget in a more efficient way, with the goal to maximize the achieved coverage and to minimize the time spent for the generation.

To address such questions, we built a *machine learning* (ML) model to predict the branch coverage that will be achieved by two test-data generation tools —EvoSuite[7] and Randoop[8]— on a given *class under test* (CUT). However, being the achievable branch-coverage strongly depending from the search-budget (*i.e.*, the time) allocated for the generations, we run each of the aforementioned tools by experimenting four different search-budgets. It is important to note that the branch-coverage achieved with every budget represents the dependent variable our models try to predict. Therefore, in this study we train and evaluate four different machine learning models for each tool, where every model is specialized into the prediction of the achievable coverage given the allocated search-budget. It is worth to note that this specialization is needed to address particular questions, like the choice of the search-budget for each test-case generation.

To select the features needed to train the aforementioned models, we investigate metrics able to represent —or measure— the complexity of a class under test. This, we select a total of 79 factors coming from four different categories. We focus on source-code metrics for the following reasons: (i) they can be obtained statically, without actually executing the code; (ii) they are easy to compute, and (iii) they usual come for free in a continuous integration (CI) environment, where the code is constantly analyzed by several quality checkers. Amongst the others, we rely on well-established source-code metrics such as the Chidamber and Kemerer (CK)[9] and the Halstead metrics[10]. To detect the best algorithm for the branch-coverage prediction, we experiment with four distinct algorithms covering distinct algorithmic families. At the end, we find the Random Forest Regressor algorithm to be the best performing ones in the context of brach-coverage prediction. Our final model shows an average *Mean Absolute Error* (MAE) of about 0.15 for EvoSuite and of about 0.22 for Randoop, on average over the experimented budgets. Considering the performance of the devised models, we argue that they can be practically useful to predict the coverage that will be achieved by test-data generation tools in a real-case scenario. We believe that this approach can support developers in taking informed decision when it comes to deploy and practical use test-case generation.

Contributions of the Paper

In this paper we define and evaluate machine learning models with the goal to predict the achievable branch coverage by test-data generation tools like EvoSuite[7] and Randoop[8]. The main contributions of the paper are:

- We investigate four different categories of code-quality metrics, *i.e.*, *Package Level*, *CK and OO*, *Java Reserved Keyword* and Halstead metrics[10] as a features for the machine learning models;
- We experiment the performance of four different machine learning algorithms, *i.e.*, Huber Regression, Support Vector Regression, Multi-Layer Perceptron and Random Forest Regressor, for the branch prediction model;
- We perform a large scale study involving seven large open-source projects for a total of 3,105 Java classes;
- We extensively ran EvoSuite and Randoop over all the classes of the study context, experimenting four different budgets and multiple executions. The overall execution was parallelized over several multi-core servers, as the mere generation of the such amount of tests would take months on a single CPU setup.

Novel Contribution of the Extension

This paper is an extension of our seminal work[11], in which we firstly propose to predict the branch coverage that will be achieved by test-data generation tools. Following, we summarize the novel contribution of this paper in respect to the original one:

- We introduce a new set of features, *i.e.*, the Halstead metrics[10]. They aim at determining a quantitative measure of complexity directly from the operators and operands in a class;
- We introduce a Random Forest Regressor —an an ensemble algorithm— for the branch prediction problem;

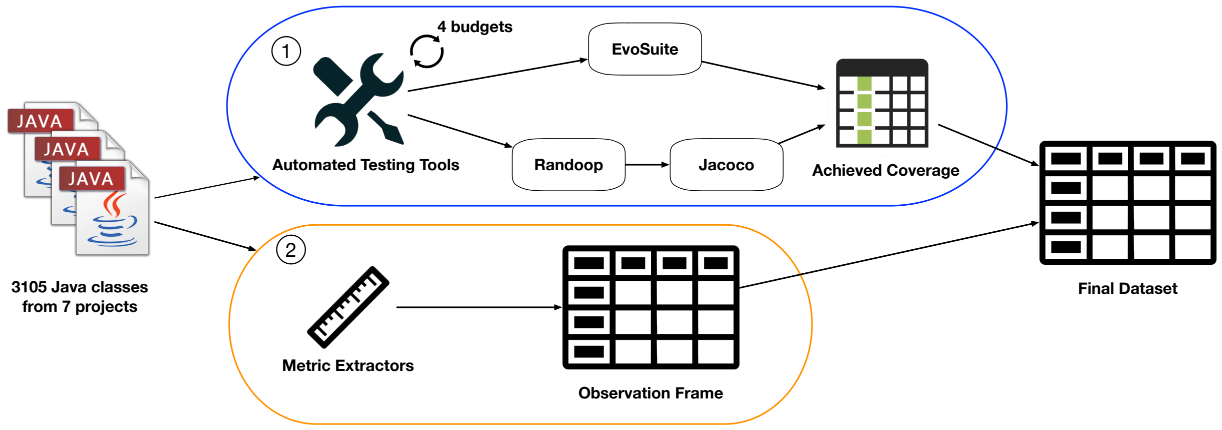


FIGURE 1 Construction process of the training dataset

- We add a fine-grained analysis aiming at understanding the importance of the employed features in the prediction;
- We build the model with three more different budgets than the default one. We generate a test suite multiple times for each class and budget, averaging the results; our preliminary findings were based on a single generation.
- We use a larger dataset to train and evaluate the proposed machine learning models.

Structure of the Paper

Section 2 describes the features we used to train and validate the proposed machine learning models; details about the extraction process are also detailed in this section. Section 3.1 introduces the research questions of the empirical study together with its context: we present there the subjects of the study, the procedure we use to build the training set and the machine learning algorithms employed. Section 4 describes the steps towards the resolution of the proposed research questions, while the achieved results are presented in Section 5; the practical implications of the main findinds are then discussed in Section 6. Section 7 discusses the main threats while related work are presented in Section 8. At the end, Section 9 concludes the paper, drawing the future work.

2 | DATASET AND FEATURES DESCRIPTION

Supervised learning models —like the ones we employ in this study— are used to predict a certain output given a set of inputs, learning from examples of input/output pairs[12]. More formally, we define $x^{(i)}$ as *independent input variables* —also called input *features*— and $y^{(i)}$ as the output —also called *dependent variable*— that we are trying to predict. A pair $(x^{(i)}, y^{(i)})$ is a single training example and the entire dataset used to learn, *i.e.*, the *training set* is a list of m training examples $\{(x^{(i)}, y^{(i)}),\ \forall i = 1, \dots, m\}$. The learning problem consists on learning a function $h = X \rightarrow Y$ with a good accuracy of h in predicting the value of y.

In the context of this study, we build four different datasets, one for each investigated search-budget, consisting of a tuple $(x^{(i)}, y^{(i)}_{b,t})$ for each class i, where:

- $x^{(i)}$ is a vector containing the values for all the factors we described in Section 2.2, for a given class i;
- $y^{(i)}_{b,t}$ is the branch-coverage value —in the range $[0, 1]$— of the test suite generated by a test-data generator tool t, for the class i with the search-budget b. This is the *dependent variable* we want to predict.

It is worth to note that we investigate the prediction performance for two tools; since we experiment four search-budgets, we build a total of eight different training sets. The process we use for the construction of such training sets is depicted in Figure 1. At first, we execute the scripts needed for the extraction of the factors described in Section 2.2 ($\textcircled{2}$ in Figure 1); those values form the $\{x^{(i)},\ \forall i = 1, \dots, m\}$ features vector, one for each subject class i. It is worth to note that the features vector is the same for all the budgets: indeed, all the factors refers to the classes under test; thus, their value is not affected by the used search-budget. Therefore, we compute the dependent variables, as explained in Section 2.1, obtaining the $\{y^{(i)}_{b,t},\ \forall i = 1, \dots, m\}$, where $y^{(i)}_{b,t}$ is the average coverage obtained for the class i by the tool t with a search budget b. At the end of the process, we result with a training dataset for each combination of tool and budget. More formally, for each budget b and tool t, we have a training dataset $\{(x^{(i)}, y^{(i)}_{b,t}),\ \forall i = 1, \ldots, m\}$ where $x^{(i)}$ and $y^{(i)}_{b,t}$ are respectively (i) the features vector for the class i, and (ii) the average coverage achieved over the independent runs by a test-data generator tool t, for a subject class i with a search-budget b. The procedure we use to calculate the dependent variable is reported in Section 2.1.

2.1 | Dependent Variable

As dependent variable, we use the branch coverage achieved by the two experimented automated tools, *i.e.*, EvoSuite and Randoop. We run both the tools with four different budgets: default (*i.e.*, 60 and 90 seconds, respectively for EvoSuite and Randoop), 180 seconds (*i.e.*, 3 minutes), 300 seconds (*i.e.*, 5 minutes) and 600 seconds (*i.e.*, 10 minutes). We select these budgets for the following reasons: 180 and 300 seconds have been the most used budgets exploited in the literature so far [13,14,15,16]; 10 minutes is a longer budget that we select to have an intuition on what can be expected with extra time allowed for the search. We do not experiment longer budgets because (i) the computation-time needed to compute the dependent variable for such a longer budget, and (ii) more importantly, the usage of test-data generation tools with such a long budget would be hardly feasible in practice.

To collect the dependent variable —*i.e.*, the variable we want to predict — we run both the tools for 10 times on the CUTs used in the study, obtaining 10 different test suites (10 for each tool, *i.e.*, 20 in total). We repeat this process for each budget we experiment. It is worth to note that, for the 10 minutes budget, we only generate 3 tests suites. To sum up, for each class in our dataset we run each tool 33 times. Thus, averaging the branch coverage of those suites by class, tool and budget, we obtain the $\{y^{(i)}_{b,t},\ \forall i = 1, \ldots, m\}$, where $y^{(i)}_{b,t}$ is the average coverage obtained for the class i by a tool t with a search budget b. We use the so computed dependent variable to build the different training datasets as described above. We averaged the results of different generations due to the non-deterministic nature of the algorithms underlying test-data generation tools. It is worth to note that such a multiple execution represents an improvement over our previous work [11], where we executed the tools once per class and with the default search-budget only.

To calculate the coverage for each run we proceed as follows (refer to $\textcircled{1}$ box in Figure 1): while EvoSuite automatically reports such information in its CSV report, we have to compile, execute and measure the achieved coverage for the tests generated by Randoop. For the measurement step we rely on Jacoco. [1] It is worth to note that we discard the data-points with branch coverage equals to 0. It is also important to underline how time expensive the described process is: we ran both EvoSuite and Randoop multiple times for the 3,105 classes we use in the study, using four different search-budgets for about 820,000 executions. In a nutshell, we estimate the entire test generation process to take about 250 days on single core machine. To speed up such process, we ran the generation on an OpenStack cluster using three different 16 cores Ubuntu server, with 64GB of RAM memory each. Figure 2 shows the distribution of the achieved branch-coverage for both EvoSuite and Randoop over the four experimented budgets. On one hand, we can observe that EvoSuite consistently reaches higher branch-coverage on the CUTs while the search-budget increases: indeed, the mean of the achieved coverage ranges from 74% with the default budget (*i.e.*, 60 seconds) to the 81% with the 600 seconds (*i.e.*, 10 minutes) budget. On the other hand, the branch-coverage reached by Randoop does not seem particularly influenced by the budget given to the search. It is worth to note that we measure the achieved coverage for Randoop over the regression test suites generated by the tool.

2.2 | Independent Variables

In this study, we consider 79 factors belonging to 4 different categories, that might be correlated with the coverage that will be achieved by automated testing tools of a given target. We train our models on a set of features designed primarily to capture the code complexity of CUTs. The first set of features comes from JDepend [17] and captures information about the outer context layer of a CUT. We then use the Chidamber and Kemerer (CK) [9] metrics —as depth of inheritance tree (DIT) and coupling between objects (CBO)— along with other object-oriented metrics, *e.g.*, number of static invocations (NOSI) and number of public methods (NOPM). These metrics have been computed using an open source tool provided by Aniche [18]. To capture even more fine-grained details, we include the counts for 52 Java keywords, including keywords such as `synchronized`, `import` or `instanceof`. In addition to the ones used in our preliminary study [11], we also include the Halstead metrics [10]. These metrics aim at measuring the complexity of a given program looking at its operators and operands.

Package Level Features

Table 1 summarizes the features computed at package-level calculated with JDepend [18]. Originally, such features have been developed to represent an indication of the quality of a package. For instance, *TotalClasses* is a measure of the extensibility of a package. The features *Ca* and *Ce* respectively are meant to capture the responsibility and independence of the package. In our application, both represent complexity indicators for the purpose

[1] https://www.jacoco.org

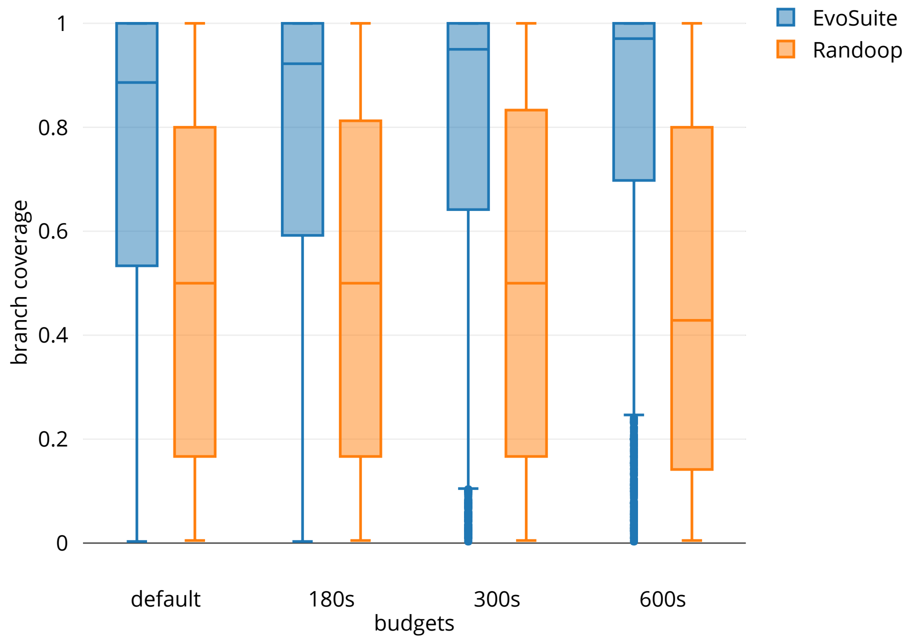


FIGURE 2 Distribution of the branch-coverage achieved by the two experimented tools over the 4 different employed budgets

TABLE 1 Package-level features computed with JDepend

Name	Description
TotalClasses	The number of concrete and abstract classes (and interfaces) in the package
Ca	The number of other packages that depend upon classes within the package
Ce	The number of other packages that the classes in the package depend upon
A	The ratio of the number of abstract classes (and interfaces) to the number of total classes in the analyzed package
I	The ratio of afferent coupling (Ce) to total coupling (Ce+Ca), such that $\mathrm{I} = \mathrm{Ce}/(\mathrm{Ce} + \mathrm{Ca})$
D	The perpendicular distance of a package from the idealized line $\mathrm{A} + \mathrm{I} = 1$

of the coverage prediction. Another particular feature we took into account was the *distance from the main sequence (D)*. It captures the closeness to an optimal package characteristic when the package is *abstract and stable*, *i.e.*, $\mathrm{A} = 1, \mathrm{I} = 0$ or *concrete and unstable*, *i.e.*, $\mathrm{A} = 0, \mathrm{I} = 1$.

Is is worth to specify the way we map such package features to the single classes. In particular, given a feature f_i calculated for a package $\mathcal{P}$, we assign the same value for the feature f_i to all the classes belonging to $\mathcal{P}$.

CK metrics and object-oriented features

This set of features includes the widely adopted Chidamber and Kemerer (CK) metrics, such as WMC, DIT, NOT, CBO, RFC and LCOM[9]. It is worth to note that the ck tool[18] calculates these metrics directly from the source code using a parser. In addition, we included other specific object-oriented features. Such a complete set, with the respective descriptions, can be observed in Table 2.

Java Reserved Keyword Features

In order to capture additional complexity in our model, we include the count of a set of reserved Java keywords (reported in our replication package[19]). Keywords have long been used in Information Retrieval as features[20]. However, to the best of our knowledge, Java reserved keywords have not been used in previous research to capture complexity. Possibly, this is because these features are too fine-grained and do not allow

TABLE 2 CK and object-oriented feature descriptions

Acronym	Name	Description
CBO	Coupling Between Objects	Number of dependencies a class has
DIT	Depth Inheritance Tree	Number of ancestors a class has
NOC	Number of Children	Number of children a class has
NOF	Number of Fields	Number of field a class regardless the modifiers
NOPF	Number of Public Fields	Number of the public fields
NOSF	Number of Static Fields	Number of the static fields
NOM	Number of Methods	Number of methods regardless of modifiers
NOPM	Number of Public Methods	Number the public methods
NOSM	Number of Static Methods	Number the static methods
NOSI	Number of Static Invocations	Number of invocations to static methods
RFC	Response for a Class	Number of unique method invocation in a class
WMC	Weight Method Class	Number of branch instructions in a class
LOC	Lines of Code	Number of lines ignoring the empty lines
LCOM	Lack of Cohesion Methods	Measures how method access disjoint sets of instance variable

the usage of complexity thresholds, like for instance the CK metrics[21]. It is also worth to underline that there is definitively an overlap for these keywords with some of the aforementioned metrics like, to cite an example, for the keywords `abstract` or `static`. However, it is straightforward to think about those keywords (*e.g.*, `synchronized`, `import` and `switch`) as code complexity indicators.

Halstead Metrics

The Halstead complexity metrics have been developed by Maurice Halstead[10] with the goal to quantitatively measure the complexity of a program directly from the source code. Ideated in 1977, they represent one of the earliest attempt to measure code complexity. Halstead metrics are calculated by processing the source code as a token sequence. Therefore, each token is classifier as an operator or an operand. Being n_1 the number of distinct operators, n_2 the number of distinct operands, N_1 the total number of operators, N_2 the total number of operands, 6 different metrics can be determined with the following formulas:

- Program vocabulary: $n = n_1 + n_2$;
- Program length: $N = N_1 + N_2$;
- Calculated program length: $\hat{N} = n_1 \log_2 n_1 + n_2 \log_2 n_2$;
- Volume: $V = N \times \log_2 n$;
- Difficulty Level $D = \frac{n_1}{2} \times \frac{N_2}{n_2}$;
- Effort: $E = D \times V$.

Operators are syntactic elements such as `+, -, <, >`, while operands consists of literal expressions, constants and variables. It is worth to note that, in our definition of Halstead metrics, functions and method calls are handled as operators, while parameters as operand. The main rationale behind the choice of this set of metrics lies indeed in the fact that they use operands and operators as atomic units of measurement. We argue that a class with more operands and operators might tend to present more complex conditional expressions in branching nodes (*e.g.*, in `if` or on `while` nodes). Therefore, such complex expressions might lead to a reduction in the coverage achieved by testing tools: in fact, the problem of reaching the maximum coverage consists of no more than the satisfaction of both the true and the false branch of every conditional statement.

TABLE 3 Halstead metrics descriptions

Acronym	Name	Description
N	Program Vocabulary	The vocabulary size is the sum of unique operators and operands number
n	Program Length	The program length is the sum of the total number of operators and operands in the program
$\hat{N}$	Calculated Program Length	According to Halstead, the length of a well structured program is function of the number of unique operators and operands
V	Volume	The program volume is the information contents of the program measured in mathematical bits
D	Difficulty Level	The difficulty level is estimated proportionally to the number of unique operators and to the ration between the total number of operands and the number of unique operands
E	Effort	Halstead hypothesized that the effort required to code a program is proportional to its size and difficulty level

Detailed explanation about the meaning of these metrics is shown in Table 3. The tool we use to extract the Halstead metrics is publicly available on GitHub. [2]

3 | RESEARCH QUESTIONS AND CONTEXT

The *goal* of this empirical study is to define and evaluate machine learning models able to predict the coverage that will be achieved by test-data generation tools on a given *class under test* (CUT). In the context of this work, we focus in particular on EvoSuite[7] and Randoop[8], two of the most well-known tools currently available. Moreover, we explore four different search-budgets, *i.e.*, the default one (60 and 90 seconds respectively for EvoSuite and Randoop), 180 seconds, 300 seconds and 600 seconds. Formally, in this paper we investigate the following research questions:

RQ1 *Can we leverage on well-established source-code metrics to train machine learning models to predict the branch coverage that will be achieved by test data generation tools?*

The identification of the right features is the first step for any prediction problem. In our study we rely on well-established source-code metrics aiming at representing the complexity of a class we want to generate tests for. Moreover, all of them can be computed statically: thus, our approach does not require code execution. In detail, we select factors coming from four different categories: *Package Level Features*, *CK and OO Features*, *Java Reserved Keyword* and *Halstead Metrics*. Therefore, in our first research question, we aim at investigating whether this kind of features can be successfully exploited to train machine learning models to predict, with a certain degree of accuracy, the branch coverage that test-data generation tools (EvoSuite and Randoop in our case) can achieve on given CUTs with a given search-budget.

We use a nested cross-validation[22] approach to train, tune, and evaluate several different machine-learning algorithms relying on the aforementioned features. This brings us to the second research in our study:

RQ2 *Which is the best performing algorithm for the prediction of the branch coverage achieved by test data generation tools?*

We train and evaluate in total 8 different models: indeed, for each tool (EvoSuite and Randoop, we build four different models, one for each search-budget. Therefore, we investigate eventual differences in the prediction performance of the two tools with different budgets allowed for the generation.

Once the best model for the prediction problem has been selected, we then conduct a fine-grained analysis aimed at investigating which are the most relevant features employed by the devised approach. Thus, we formulate our third research question:

RQ3 *What are the most important factors affecting the accuracy of the prediction?*

In the context of this research question, we further elaborate on the impact in the prediction accuracy given by the set of features introduced in this work, with respect to the previous study[11].

[2]https://github.com/giograno/Halstead-Complexity-Measures

TABLE 4 Projects used to build the ML models

	Guava	Cassandra	Dagger	Ivy	Math	Lang	Time
LOC	78,525	220,573	848	50,430	94.410	27.552	28.771
Java Files	538	1,474	43	464	927	153	166
#classes employed	449	1,278	14	410	668	124	142

3.1 | Context Selection

The context of this study is composed by 7 different open source projects: Apache Cassandra[23], Apache Ivy[24], Google Guava[25], Google Dagger[26], Apache-Commons Lang[27], Apache-Commons Math[28] and Joda-Time[29]. We selected those projects due to their different domain; moreover, the Apache-Commons is quite popular in software evolution and maintenance literature[30]. Apache Cassandra is a distributed database; Apache Ivy a build tool; Google Guava a set of core libraries; Google Dagger a dependency injector; Joda-Time a replacement for the Java date and time class; Commons-Lang provides helper utilities for Java core classes while Commons-Math is a library of mathematics and statistics operators. Table 4 summarizes the Java classes and the LOC used from the above projects to train our ML models. The LOC and the total number of Java files has been calculated with the `cloc`[3] tool. To foster full replicability of the results, we share the aforementioned projects —along with their correspondent versions— in our replication package[19]. For different reasons, not all the classes of a project might be testable. To obtain the list of testable classes, we exploit a feature of EvoSuite: the tool is able to scan a given class-path and return the list of classes that EvoSuite thinks are testable (*e.g.*, public). It is worth to note that, for this reason, the number of classes (reported in the row *#classes employed* in Table 4) we use for our study is lower than the total number of classes available in the selected projects. With this approach, we ended with 3,105 different classes as a context of this study.

3.2 | Machine Learning Algorithms

In this section we present the four algorithms used in our study and the parameters we tune during the training process: *i.e.*, Huber regression[31], Support Vector Regression[32], Multi-layer Perceptron[33] and Random Forest Regressor[34]. With these four ML algorithms we cover four different ML families, *i.e.*, a robust regression, a SVM, a neural network, and an ensemble algorithm. We selected the mentioned algorithms for different reasons: first, they have been previously used in Software Engineering studies, showing to be highly efficient approaches[35,36,37,38,39?]. In particular, we include the Random Forest since (i) it is able to automatically filter out non-relevant features, avoiding problems related to multi-collinearity[40] and (ii) its results are easier to interpret than other classifiers[41]. We use the implementation of the Python's ScikitLearn Library[42], being an open source framework widely used in both research and industry.

3.2.1 | Huber Regression

Huber Regression[31] is a robust linear regression model designed to overcome some limitations of traditional parametric and non-parametric models. In particular, it is specifically tolerant to data containing outliers. Indeed, in case of outliers, least square estimation might be inefficient and biased. On the contrary, Huber Regression applies only linear loss to such observations, therefore softening the impact on the overall fit. The only parameter to optimize in this case is α, a regularization parameter that avoid the rescaling of the epsilon value when the `y` is under or over a certain factor[31]. We investigated the range of 2 to the power of `linspace(-30, 20, num = 15)`. It is worth to specify that `linspace` is a function that returns evenly spaces number over a specified interval. Therefore, in this particular case, we used 2 to the power of 15 linearly spaced values between -30 and 20.

3.2.2 | Support Vector Regression

Support Vector Regression (SVR)[32] is an application of Support Vector Machine algorithms, characterized by the usage of kernels and by the absence of local minima. The SVR implementation in Python's Scikit library we used is based on `libcsv`[32]. Amongst the various kernels, we chose a *radial basis function kernel (rbf)*, which can be formally defined as $\mathtt{exp}(-\gamma||\mathtt{x} - \mathtt{x}'||^2)$, where the parameter γ is equal to $1/2\sigma^2$. This approach basically learns non-linear patterns in the data by forming hyper-dimensional vectors from the data itself. Then, it evaluates how similar new observations are to the the ones seen during the training phase. The free parameters in this model are C and ϵ. C is a penalty parameter of the error term,

[3]http://cloc.sourceforge.net

while ϵ is the size within which no penalty is associated in the training loss function with points predicted within a distance epsilon from the actual value[32]. Regarding C, just like Huber Regression, we used the range of 2 to the power of `linspace(-30, 20, num = 15)`. On the other side, for the parameter ϵ, we considered the following initial parameters: 0.025, 0.05, 0.1, 0.2 and 0.4.

3.2.3 | Multi-Layer Perceptron

Multi-layer Perceptron (MLP)[43] is a particular class of feedforward neural network. Given a set for features $X = x_1, x_2, ..., x_m$ and a target y, it learns a non-linear function $f(\cdot) : R^m \rightarrow R^o$ where m is the dimension of the input and o is the one of the output. It uses backpropagation for training and it differs from a linear perceptron for its multiple layers (at least three layers of nodes) and for the the non-linear activation. We opted for the MLP algorithm since its different nature compared to two approaches mentioned above. Moreover, despite they are harder to tune, neural networks offer usually good performances and are particularly fitted for finding non-linear interactions between features[33]. It is easy to notice how such a characteristic is desirable for the kind of data in our domain. Also in this case we performed a grid search to look for the best hyper-parameters. For the MLP we had to set α (*alpha*), *i.e.*, the regularization term parameter, as well the number of units in a layer and the number of layers in the network. We looked for α again in the range of 2 to the power of `linspace(-30, 20, num = 15)`. About the number of units in a single layer, we investigated range of $0.5x$, $1x$, $2x$ and $3x$ times the total number of features in out model (*i.e.*, 73). About the number of layers, we took into account the values of 1, 3, 5 and 9.

3.2.4 | Random Forest Regressor

Random Forest Regressor (RFR)[34] represents one of the most effective ensemble machine learning method. An ensemble method uses multiple learning algorithms to obtain better predictive performance. Indeed, a random forest fits classifying decision trees on a sub-sample of the dataset and then uses averaging to improved the final prediction and control overfitting. More formally, the final model $g(x)$ can be defined as $g(x) = f_0(x) + f_1(x) + f_2(x) + ... + f_n(x)$ where the various f_i are the simple decision trees f_i. We include RFC in our study given its capability to handle numerical features, to capture non-linear interaction between the features and the target and to automatically filter out non-relevant features, thus avoiding problems related to multi-collinearity[40]. We tune this model looking at four different parameters:

- `n_estimators`, that represents the number of trees in the forest; we experimented the values in the set $\mathcal{S}_{p_1} = \Big\{2 \cdot x | x \in \big[1, ..., 10\big]\Big\}$;
- `n_features`, that is the number of features to consider when looking at the best split; we considered the values in the set $\mathcal{S}_{p_2} = \Big\{(m//10) \cdot x | x \in \big[1, ..., 10\big]\Big\}$, where m is the number for features available;
- `max_depth`, that is the maximum depth of each decisional tree; we experimented the values in the set $\mathcal{S}_{p_3} = \Big\{5 \cdot x | x \in \big[1, ..., 10\big]\Big\}$;
- `min_sample_leaf`, that is the minimum number of sample required to be at a leaf node; we tried the set of values $\mathcal{S}_{p_4} = \Big\{x | x \in \big[1, ..., 5\big]\Big\}$.

4 | EMPIRICAL STUDY DESIGN

The process we use to answer our research questions is composed by the following steps: at first —as detailed in Section 2— we build a different training sets (i) extracting the described features for the classes under test and (ii) computing the coverage achieved by the testing tools on the same classes with a given search-budget. Therefore, we use a nested-cross validation[22] procedure to tune the parameters of the four employed algorithms and, at the same time, selecting and evaluating the performance of the best one. At the end, we rely on such a best model to compute the importance ranking of the various features for the prediction.

4.1 | RQ1/RQ2 Design: Performance of the Prediction with Source-Code Metrics Features

The goal of the first two research questions is twofold: at first, we want to explore the feasibility for source-code metrics to be used as features for machine learning models in order to predict the achievable branch coverage. Therefore, we aim to detect the best algorithm to tackle such a problem given the presented features.

To train and validate the experimented models we use nested cross-validation[44]. This choice is due to advances achieved in Machine Learning research[44,45] showing that nested cross-validation allows to reliably estimate generalization performance of a learning pipeline involving different steps like preprocessing, features and model selection[45]. In fact, model selection without nested cross-validation (*e.g.*, the classical 10-fold cross validation) would rely on the same data to both tune the parameters of the models and to evaluate their performance: this might risk to optimistically

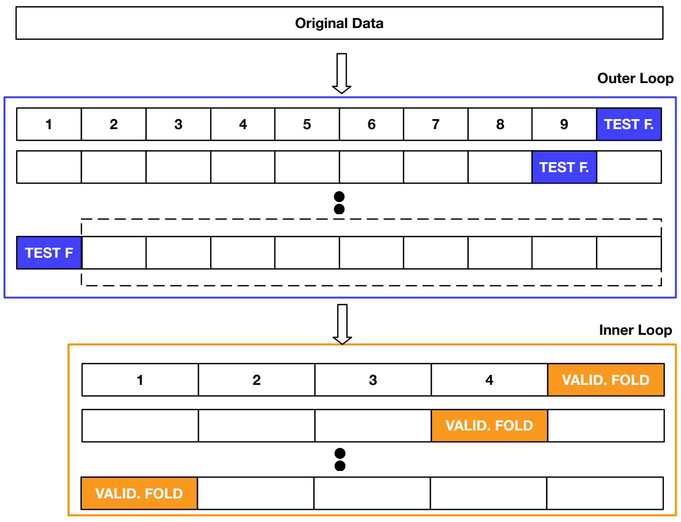


FIGURE 3 Nested Cross-Validation Procedure. The inner fold relies on a 5-fold cross validation while the outer fold on a 10-fold cross validation.

bias the model evaluation, leading to overfit the data[22]. To avoid such a problem, nested cross-validation uses a set of train/validation/test splits in two separate loops. In particular, we adopt a 5-fold cross-validation for the inner loop and a 10-fold cross-validation[46] for the outer loop. The correspondent process is depicted in Figure 3. Nested cross-validation works as follows: at first, an inner cross-validation loop fits a model —one for each algorithm and combination of parameters— for each training set. Thus, the best configuration is selected over the validation set (the orange boxes on Figure 3). Therefore, the outer loop estimates the generalization error by averaging the test score over several test splits (the boxes in blue on Figure 3).

In detail, in our implementation we apply the well-known *Grid Search* method[47], consisting in training different models to explore the parameter space to find the best configuration. To this aim, we rely on the `GridSearchCV` utility[4] provided by `sciknit-learn`. It is worth to note that, for every combination of parameters, we train two different models, one with feature scaling (*a.k.a.*, data normalization) preprocessing[48] and one without. Feature scaling mutates the raw feature vector into a more suitable representation for the downstream estimator: such a normalization is needed to contrast the fact that different independent variables have a pretty different range of values and it is important especially for Support Vector Machine algorithms[47], since they assume the data to be in a standard range. We rely on the StandardScaler implemented in `scikit-learn` that processes the features by removing the mean and scaling to unit variance, thus centering the distribution around 0 with a standard deviation of 1.

For the outer loop, we use the `cross_validate` function provided by the `sklearn.model_selection` module.[5] It is worth to note that, to cope with the randomness arising from using different data splits[49], we repeat the outer cross-validation loop 10 times. We use the Mean Absolute Error (MAE) as test score to evaluate the inner cross-validation. The MAE is formally defined as:

$$MAE = \frac{\sum_{i+1}^{n} |y_i - x_i|}{n}$$

where y is the predicted value, x are the observed values for the class i and n is the entire set of classes used in the training set. We rely on MAE since is easy to interpret, being in the same unit of the target variable, *i.e.*, branch coverage fraction. MAE ranges between 0 and 1, where a MAE of 0 indicates a perfect prediction, and a MAE of 1 refers to the worst possible one (*i.e.*, actual coverage 0%, with the model predicting 100%, and viceversa). Practically speaking, a MAE $= 0.10$ indicate that our model predicts on average the achievable coverage either as 10% higher or lower than the actual value. Moreover, previous research recommends MAE to compare the performance of prediction models, given its unbiased nature towards over and underestimation[50]. In addition to MAE, outer loop relies on different performance indicators, *i.e.*, R2 Score, Mean Squared Error (MSE), Mean Squared Log Error (MSLE) and Median Absolute Error (MedianAE)[51].

4.2 | RQ3 Design: Feature Analysis

In addition to the first two research questions, we conduct a fine-grained analysis to understand which are the most influential factors uses by the trained model to actually predict the output variable, *i.e.*, the achievable branch coverage. This fine-grained analysis aims at answering our **RQ**$_3$.

[4]http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html
[5]http://scikit-learn.org/stable/modules/cross_validation.html

Algorithm 1: Mean Decrease in Accuracy (MDA) algorithm

```
Input: training data X and desired output values vector Y
Result: feature MDA scores F
begin
    for k ∈ k-fold validations do
        $X_{train}, X_{test} \longleftarrow$ train_test_split(train_size=0.7, test_size=0.3);    // split 70/30 into train and test set
        $Y_{train}, Y_{test} \longleftarrow$ train_test_split(train_size=0.7, test_size=0.3)
        model $\longleftarrow$ fit the regressor
        acc $\longleftarrow Y_{test} -$ prediction on $X_{train}$;    // accuracy computed with R2 score
        for f features do
            $X_{new} \longleftarrow$ permute values of f
            $acc_{new} \longleftarrow Y_{test} -$ prediction on $X_{new}$;    // new accuracy with permutation on feature f
            $F[f] \longleftarrow F[f] \cup (acc_{new} - acc/acc)$
        end
    end
    average F[f] over the k-folds
end
```

In order to detect the most relevant features influencing the prediction, we rely on the *Mean Decrease in Accuracy* (MDA) approach[52]. MDA estimates the importance of a certain feature in terms of the reduction it provides to the overall accuracy of the model. Algorithm 1 shows the pseudo-code of the entire process. The approach trains n different models, where n is the number of the features. Each feature f is represented by a vector $f = \langle f_1, f_2, ..., f_m \rangle$ where m is the number of classes and f_i is the value of the feature f for the class i. For each feature $j = 1, ..., n$, the correspondent f_j feature vector is randomly permuted (*i.e.*, the vector is shuffled and thus the features assume wrong values for the classes) and the accuracy is again computed[53] (lines 8-9 of Algorithm 1). The MDA score for the variable j is computed as:

$$MDA(j) = \frac{acc_j - acc}{acc}$$

where acc_j is the accuracy computed with the permuted values of the feature j and acc is the accuracy computed on the original train set (line 10 of Algorithm 1). Clearly, for important variables such a permutation will have significant effects on the accuracy of the model, while for unimportant variables the permutation should have little effects. For the calculation of the prediction accuracy, we rely on the R2 score (R-squared)[51]. In order to have more precise results, we repeat such a process for k-times using a 10-fold cross-validation process, averaging the results over the separate runs (line 13 of Algorithm 1). It is worth to note that sklearn does not expose any implementation for MDA; we share our implementation in the replication package[19].

5 | RESULTS

In the following section we report the results of the four discussed research questions; therefore, we present and discuss the related main findings.

5.1 | RQ_1/RQ_2 - Performance of the Prediction with Source-Code Metrics Features

To answer both **RQ_1** and **RQ_2** we rely nested cross-validation, an approach able to tune, train and evaluated different algorithms with different configurations of parameters, giving in output the best model overall. At the end of such a procedure, the Random Forest Regressor results the best algorithm for our prediction model: indeed, it achieves the best Mean Average Error (MAE) for all the possible models built experimenting the two tools and the four search-budget. Is is worth to note that is the first improvement over our previous work[11]. In fact, in this study we introduce the RFC, comparing it with the three algorithms previously evaluated, *i.e.*, Huber Regression, Support Vector Regression and Multi-Layer Perceptron. Given this result, only we focus and discuss the performance of this model in the remaining of the paper. However, detailed results concerning the remaining algorithms can be found in our replication package[19]. Table 5 shows the performance of the Random Forest Regressor for the five considered evaluation metrics at the end of nested cross-validation, for each investigated configuration. As explained in Section 4, we build 8 different models: four based on the coverage achieved by EvoSuite with the four different search-budgets; four based on the one achieved by Randoop over the same four budgets. Figure 4 shows the Mean Average Error (MAE) achieved over the cross-validation for all the aforementioned eight models: the five metrics we use for the evaluations are reported in the rows, while the columns report their values for each combination of tool/budget. In addition we report in Table 6 the optimal combination of parameters resulting by the employed nested cross-validation procedure following the same structure of Table 6.

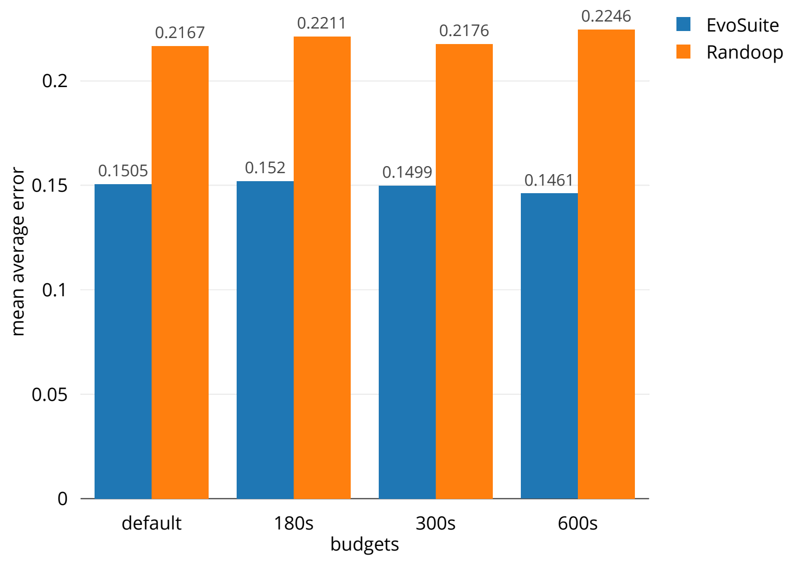


FIGURE 4 Mean Average Error (MAE) over cross-validation for the Random Forest Regressor. The blue bar refers to the model trained on the EvoSuite outcomes, while the orange bar refers to Randoop. The MAEs are grouped according to the search-budget used to train the models.

TABLE 5 Results for the Random Forest Regressor over nested cross-validation. We report the statistics for each built model, *i.e.*, 4 for each tools, according to the 4 experimented search-budgets

	default		**180s**		**300s**		**600s**	
	EvoSuite	Randoop	EvoSuite	Randoop	EvoSuite	Randoop	Evosuite	Randoop
Mean Abs. Error	0.151	0.217	0.152	0.221	0.150	0.218	0.146	0.225
Mean Sqr. Error	0.044	0.071	0.046	0.073	0.046	0.073	0.046	0.077
Mean Sqr. Log Error	0.019	0.033	0.020	0.034	0.020	0.034	0.019	0.037
Mean Abs. Error	0.110	0.192	0.110	0.197	0.104	0.185	0.097	0.195
R2-Score	0.525	0.364	0.479	0.348	0.452	0.358	0.412	0.322

In detail, we observe a MAE of about 0.15 for EvoSuite on average over the different search-budgets. In this case, the performance of the predictions tends to be slightly better when the search-budget increases: indeed, for the 10 minutes budget, we observe MAE $= 0.146$. Despite this improvement is only marginal, we attribute it to the nature of the genetic algorithms (GA) used by EvoSuite. Indeed, more search-budget given to the search, more time for the GA to explore all the possible solutions in the search-space. On the contrary, in case of lower budget, the search is still largely influenced by the initial solutions randomly generated at the beginning of the evolutionary search. Observing the MAEs achieved by the Randoop models, we notice that there is no trend in the performance related to the changes in the employed search-budgets. Randoop relies on random-search: this means that the additional budget given to the search is merely used to generate additional random inputs, without any *guidance* towards better solutions. We argue that the intrinsic randomness of Randoop makes harder the prediction of the achievable branch-coverage, compared to EvoSuite. It is worth to remark that, in our previous work[11], we achieve a MAE of 0.291 and 0.225 respectively for the EvoSuite and Randoop model (only trained with the default-budget). Therefore, in this work we are able to significantly improve the performance for the prediction of the coverage reached by EvoSuite, with about the 48% of MAE reduction for the default budget. On the contrary, the prediction of Randoop with the default budget only shows a very marginal improvement (about 3.6% of MAE reduction). As mentioned before, we believe that the prediction for a totally random algorithm is intrinsically harder. Moreover, that the greater improvement of the EvoSuite model, compared to the Randoop one, might also depend on the stability of the achieved coverage over the 10 runs. Indeed, EvoSuite has higher standard deviation and therefore, averaging the dependent variable different runs might noticeably foster the accuracy of the ML model. It is worth to remember that, in our previous work[11], we run the testing tools only once per each class.

TABLE 6 Optimal parameter combinations for the Random Forest Regressor over nested cross-validation. We set all the other parameters to their default value. We report the data for each combination of tool and search-budget.

	default		**180s**		**300s**		**600s**	
	EvoSuite	Randoop	EvoSuite	Randoop	EvoSuite	Randoop	Evosuite	Randoop
`n_estimators`	16	14	16	16	20	6	8	4
`max_features`	71	55	71	71	79	55	63	79
`max_depth`	45	40	25	35	30	30	45	35
`min_sample_leaf`	1	3	1	1	1	1	1	3

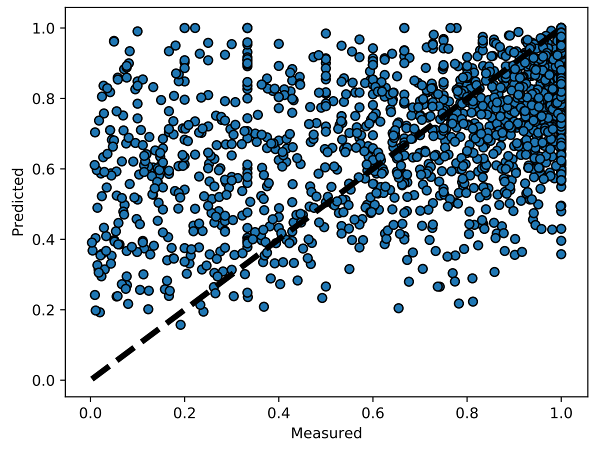
TABLE 7 Analysis of the quartiles for the difference (in absolute value) between the predicted achievable coverage and the actual measured value. We report the data for each combination of tool and search-budget.

	default		**180s**		**300s**		**600s**	
	EvoSuite	Randoop	EvoSuite	Randoop	EvoSuite	Randoop	Evosuite	Randoop
Q1 (0.25)	0.032493	0.089104	0.033553	0.090193	0.030084	0.083736	0.020833	0.085119
Q2 (0.50)	0.108008	0.191158	0.104393	0.199465	0.100278	0.188710	0.098958	0.175827
Q3 (0.75)	0.222223	0.297596	0.219620	0.313606	0.211624	0.323992	0.214683	0.323685

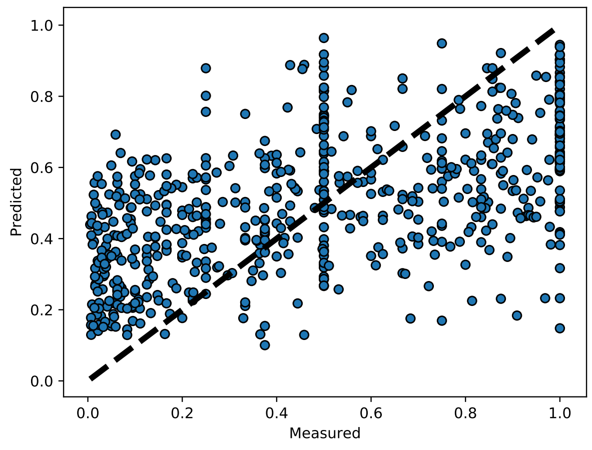
The presented results also strengthen the findings arose in our original paper. In particular, we confirm the possibility to use traditional code-metrics as features in order to train machine learning models able to predict the coverage that will be achieved by automated testing tools. It is also worth to note that the Random Forest Regressor we introduce in this study performs better than the three algorithms investigated in our previous work[11].

To provide a deeper understanding of the prediction performance and complement the evaluation based on the mere MAE metric, we report in Table 7 the quartiles for the differences between the predicted and the measured values of achieved coverage. Moreover, we visualize the results of the predictions relying on scatterplots, showed in Figure 5. In particular, the subfigure 5a refers to the predictions given by the EvoSuite model, while subfigure 5b refers to the prediction given by the Randoop one. It is worth to note that both of them show the predictions of the model trained with the search-budget of 300 seconds. For the sake of space we do not include the plots for all the combination tool/budget in the paper; however, they can be found in our replication package. Nevertheless, the distribution of the predictions and their pattern is similar to the one showed in the paper. Such scatterplots are particularly helpful to visualize the prediction errors of the trained models: the dashed-black line represents a perfect prediction. Each point is represented by a blue dot in term of x and y coordinates, where x and y denote respectively the measured and the predicted value for a given point. Obviously, closer is a blue dot to the black line, more accurate is the prediction. The coverage values on both axes are represented in a scale between 0 and 1 (*i.e.*, from 0% to 100% of coverage over the CUT). It is worth to recall that the mean average error for the two models is respectively 0.150 and 0.218 (see table 5). A $\mathrm{MAE} = 0.15$ simply means that the prediction is, on average, wrong by 15% in absolute value (*i.e.*, the measured coverage is either 15% higher or lower than the predicted one). Interesting findings arise looking at the distribution of the predictions; indeed, the pattern that arises from both the scatterplots shoes that generally the models tend to slightly over-estimate the prediction for lower values of measured coverage. Obviously, on the contrary, when the measured value of the achieved coverage is high, the error in the prediction is mostly by underestimation. As said, a very similar pattern is observed for each other search-budget we experimented.

Despite the average accuracy of the prediction, the plots in Figure 5 also show that in some cases the absolute value of the difference between the predicted and the measured value is quite high: 0.9 and 0.82 are the worst cases respectively for EvoSuite and Randoop, measured for the classes `JarResource` and `Ivy14`, both from the `ivy` project. We plan to explore the utility of clustering in reducing error for the cases where our approach poorly performs[54]. In detail, clustering algorithms could be used to group classes with similar structural properties and therefore, *ad-hoc* models can be trained over the resulting clusters.

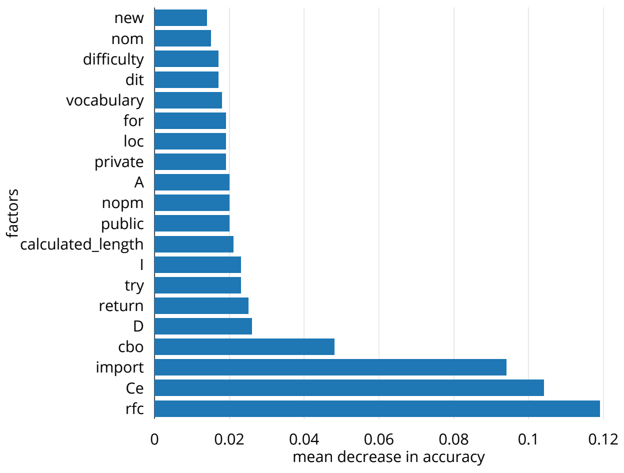


(a) Prediction errors for the EvoSuite model trained with 300 seconds a search-budget. The MAE for such a model is equal to 0.150.

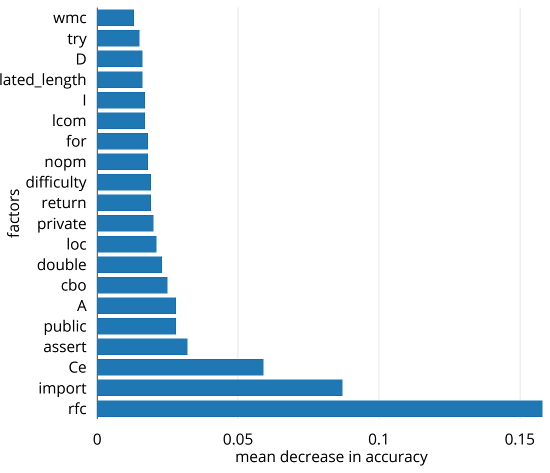


(b) Prediction errors for the Randoop model trained with 300 seconds a search-budget. The MAE for such a model is equal to 0.210.

FIGURE 5 Scatterplots representing the prediction errors of two models over 10-cross validation. Subfigure 5a shows the predictions for the EvoSuite model, while subfigure 6b shows the same data for the Randoop model. Both the subfigures refer to the correspondent models trained with a search-budget of 300 seconds. More accurate the predictions, closer the blue dots to the black-dotted line.



(a) Top 20 features for the EvoSuite model trained with 300 seconds as search-budget



(b) Top 20 features for the Randoop model trained with 300 seconds as search-budget

FIGURE 6 Bar chart representing the most 20 important feature according to their Mean Decrease in Accuracy (MAE) score. Sub-figure 6a reports such scores for the model trained on EvoSuite, while Sub-figure 6b reports the top scores for the Randoop one. Both the models have been trained with a search-budget of 300 seconds.

In Summary. We confirm the feasibility of code-metrics as features for the achievable coverage prediction problem. Random Forest Regressor is the most accurate algorithm amongst the considered ones, scoring an average 0.15 and a 0.21 MAE respectively for the EvoSuite and the Randoop model over the different search-budgets. Comparing these performance to the Support Vector Regressor model proved to the best in our previous work, we report a significant MAE reduction regarding the EvoSuite model; for the Randoop ones, on the contrary, the improvement is only marginal.

5.2 | RQ$_3$ - Feature Analysis

To answer our third research question we rely on the Random Forest Regressor algorithm we found to be the best performing one from the previous research questions. Figure 6 depicts a bar plot showing the most 20 relevant feature, according to their *Mean Decrease in Accuracy*, used by the two models, respectively by EvoSuite (6a) and by Randoop (6b). The MDA varies in a range between 0 and 1 and has the following interpretation: omitting from the training the feature `f`, with a MDA equals to `x`, will approximate results in a loss of prediction accuracy equal to `x`. It is worth to note that both the figures refer to the models trained with 300 seconds as a search-budget. In order to investigate the impact of different budgets in the relevant features, we compute the mean decrease in accuracy for every configuration of tool and budget. Our analysis shows that the most relevant features tend to be same varying the search-budget.

Looking at the most relevant features for the EvoSuite model, we observe that the CK and the Object-Oriented metrics are amongst the most important ones, with 6 out of the top 20 features. In particular, we notice that coupling-related metrics like CBO (Coupling Between Objects) RFC (Response for a Class) and Ce (afferent coupling) are immediately ranked at the top of the MAE ranking. Package-level and keyword features counts respectively for 4 and 7 out of the top 20 features. Looking at the Halstead metrics we introduce in this work, we observe that 3 of them (out of 6), namely the *program vocabulary* (n) the *calculated program length* ($\hat{N}$) and *difficulty level* (D) are ranked amongst the top 20.

Figure 6b shows the top 20 most important features for the model trained on the Randoop tool. We observe a similar distribution of features compared to the EvoSuite model: coupling-related factors like RFC and Ce are still at the top of the ranking. CK and Object-Oriented features take 6 out of the 20 metrics, while package-level and keywords features counts for 4 and 7 out of 20, respectively. Regarding the Halstead metrics, for this model both the *program difficulty* (D) and the *calculated program length* ($\hat{N}$) metrics are present in the top 20 ranking.

To strengthen the analysis performed relying on the *Mean Decrease in Accuracy*, we take advantage of a built-in feature of Random Forest. In particular, a RFR is able to automatically discern the most relevant features influencing the prediction. In doing so, it relies on the *Gini* index, also referred as *Mean Decrease in Impurity* (MDI)[55]. MDI indicates the importance of a given features in respect to the overall entropy of the model. Therefore, to confirm the results provided by the MAEs, we execute 10 different validation runs computing, for each of them, the *Gini* index for each features. It is worth to note that, for every model training, the `sklearn` implementation of the Random Forest automatically computes and stores the information about the *Gini* index in a `feature_importance` vector. The results provided by the MDI analysis (fully reported in our replication package[19]) essentially confirm the most important features discussed so far. Indeed, for the EvoSuite model 18 out of 20 most important features according to the *Gini* index are present in the MDA ranking showed in Figure 6a. On the other hand, 15 out of 20 top features according to the *Gini* index overlap with the ones in the MDA ranking showed in Figure 6b.

In Summary. Coupling-related features like RFC, CBO and Ce are the most important features for the branch coverage prediction. This result is valid for both the EvoSuite and the Randoop model. Some of the Halstead metrics we introduce, like *calculated program length*, *program vocabulary* and *program difficulty*, appear amongst the 20 most important features for both the MDA and the MDI analysis. The most related features are not influenced by the search-budget given to the search.

6 | DISCUSSION AND PRACTICAL USAGE

The prediction model for the achievable branch-coverage by automated testing tools might have different concrete applications in practice. At first, it is important to underline that predictions are intrinsically dependent to the search-budget these tools use for the search. Indeed, the problem we investigate is the prediction of the achievable branch-coverage as dependent variable. Such a value is obviously influenced by the search-budget employed for the search: more the budget, higher the chances to reach higher coverage values. This is the reason why we have to train different models for each experimented search-budget. This is particularly crucial for one of the applications we further discuss, *i.e.*, the problem of allocating a proper budget to the test-case generation.

We list following some possible practical applications of the proposed models.

Generating vs Manually-Writing Tests. Test-case generation has the primarily goal of alleviate developers burden for manually writing test cases. One of the possible applications of a prediction model might help developers to decide for which classes spend some manual effort in writing tests and, on the other side, for which others exploit testing tools to automate the generations of tests. To take such an *informed decision*, we envision the two following scenarios: In the first case, our model predicts *a-priori* that the coverage reached by an automated tool will be low; thus, developers would manually write tests for the these classes, saving at all the cost (*i.e.*, time) deriving from running a test-case generation tool. In the opposite case, the model predicts high coverage for a given set of classes. Then, developers might decide to rely on the tests generated by the tool, only checking and/or adding assertions to the generated tests.

Search-Budget Allocation. As said, generally more the search-budget given for the search, higher the achieved branch coverage. While this relations is not always true for Randoop, it absolutely holds for EvoSuite and other similar tools that rely on guided-search algorithms. However, in some cases —especially for a very trivial class— a testing tool achieves a high-level of coverage in a short amount of time, sometimes even the 100% of coverage with the default search-budget. The same might happen also for classes that are generally trivial to cover, except for a couple of very hard-branches to hit. In this cases, the coverage is never maximized: it immediately reaches high level and then does not substantially increase even with more search-budget allocated for the generation. We report some related examples from our context of study: for the class `Equivalence` of the `guava` project, EvoSuite reaches about 60% of branch-coverage with all the experimented search-budgets (*i.e.*, 1, 3, 5 and 10 minutes). The same happens for the class `TypeSizes` of the *cassandra* project: here the branch-coverage reached with all the budgets by EvoSuite remains the same, *i.e.*, about 72%. In cases like the aforementioned ones, choosing a smaller search-budget would allow to reach the same coverage with a noticeable saving in computational time. In this study, we build several models trained on different search-budgets. We argue that the prediction for the achievable coverage under different budgets might help developers in properly allocating the time to spent for test-case generation for different CUTs.

Delta Generation. The execution of test cases takes place in different phases of the software development pipeline. In continuous integration, tests usually run every time new code is committed. Often, integrated automation servers like Jenkins[6] rely on plugins (*e.g.*, Cobertura[7]) to generate trend reports of the coverage. These reports are stored and developers rely on them to diagnose the health status of the test suite. We envision our prediction model to be integrated in such a process with the aim to increase the overall quality —expressed in term of covered code— of the exiting test suites. New plugins can be integrated in automation servers like Jenkins: they would rely both on the coverage reports automatically generated at every commit and on the predictions given by the model with the goal to identify che classes for which new tests might be generated. For instance, assume a test case $\mathtt{t}$ in the test suite that has a coverage value $\mathtt{c}$ for a given production class. At the same time, the coverage prediction for the same class is equal to $\mathtt{c} + \mathtt{x}$, where $\mathtt{x}$ is the delta between the actual coverage and the predicted one. A potential plugin might automatically trigger the generation of new tests for all the classes according to a given $\mathtt{x}$ delta. Moreover, such a threshold might be set by the developers according to their personal needs.

7 | THREATS TO VALIDITY

In this section we analyze all the threats that might have an impact on the validity of the results.

Threats to Construct Validity.

To have a wide overview of the extend to which a machine learning model might predict the branch coverage achieved test data generation tools, we initially experimented 4 different algorithms, *i.e.*, Huber Regression[31], Support Vector Regression[32], Random Forest[34] and Vector Space Model[33]. Future effort will go in the direction of enlarging the number of algorithms employed.

In our study we rely on two different test data generation tools: EvoSuite, based on genetic algorithms, and Randoop, which implements a random testing approach. Despite we rely on the most widely used tools in practice, we cannot ensure the applicability or our findings to different generation approaches such as AVM[56] or symbolic execution[57].

The prediction of the achievable branch-coverage is dependent by the search-budgets given to the tools for the generation. In the context of this work, we investigate four different budgets: the default one; 3 minutes; 5 minutes; and 10 minutes. We select these budgets for the following reasons: the first three have been previously exploited in research[13,14,15,16] and represent a great balance between the time spent for the generation and the coverage that can be achieved. 10 minutes was selected to investigate the performance of the prediction given a longer budget. We argue that relying on even longer budgets would make test-case generation hardly doable in practice, especially in continuous integration scenario. Different models built with different budgets might help developers to assign the proper budget to test-case generation. We argue that the range of budgets we investigate is broad enough to cover most of the concrete applications. However, new models can be easily built with an *ad-hoc* search-budgets, following the same steps we describe in this paper.

Threats to Conclusion Validity.

To select and validate the best model amongst the four experimented algorithms, we used a nested cross-validation procedure with 5-fold inner and a 10-fold outer cross-validation. Therefore, we configured the parameters of each model with the aim of relying on the most effective configuration.

[6]https://jenkins.io
[7]http://cobertura.github.io/cobertura/

Moreover, to reduce the biases in the interpretation of the results and to deal with the randomness arising from using different data splits we repeated the validation 10 times. To properly interpret the prediction results, we exploited a number of evaluation metrics, with the aim of providing a wider overview of the performance of the devised model. Finally, when evaluating the most relevant features adopted by the RFR prediction model, we relied on both *Mean Decrease in Accuracy*[52] and on *Gini* index[55].

Threats to External Validity.

In this category, the main discussion point regards the generalizability of the results. We conducted our study taking into account 79 factors related to four different categories, *i.e.*, package level features, CK and OO features, Halstead's metrics and Java reserved keywords. To calculate them we rely on the ck tool[18]. Thus, metrics calculated with this tool may presents small variations compared to different tools. However, we argue that a possible small variation for each metric/tool would not affect the results[58]. Of course, there might be other additional factors the achieved coverage that we did not consider. About the threats to the generalizability of our findings, we train our models with a dataset of 7 different open source projects, having different size and scope. However, 4 out of these 7 projects belong to the Apache Software Foundation; this might introduce some bias in the generalizability of the results. Hence, we plan to enlarge the dataset by including projects of different kind and domain.

8 | RELATED WORK

The closer work to what we present in this paper is the one of Ferrer *et al.*[59]. They proposed the Branch Coverage Expectation (BCE) metric as the difficulty for a computer to generate test cases. The definition of such a metric is based on a Markov model of the program. They relied on this model also to estimate the number of test cases needed to reach a certain coverage. Differently for our work, they showed traditional metrics to be not effective in estimating the coverage obtained with test-data generation tools. Phogat and Kumar[60] started from the study of the literature to list all the existing object-oriented metrics related to testability. Shaheen and du Bousquet investigated the correlation between the Depth of Inheritance Tree (DIT) and the cost of testing[61]. Analyzing 25 different applications, they showed that the $\mathtt{DIT}_\mathtt{A}$, *i.e.*, the depth of inheritance tree of a class without considering its JDK's ancestors, is too abstract to be a good predictor. Gu *et al.*[62] investigated the testability of object-oriented classes. They observed that the most effective test cases consist of a tuple $(\mathtt{s}, \omega)$ where s is the class state and ω is a sequence of operations applicable to that state. Kout *et al.*[63] adapted the Metric Based Testability Model for Object-Oriented Design (MTMOOD) proposed by Khan *et al.*[64] to assess the testability of classes at the code level. Khanna[65] used an Analytic Hierarchy Process (AHP) method to detect the most used metrics for testability. It results that CK metrics and NOH metric, *i.e.*, number of class hierarchy in the design, have the higher priority. In her work, du Bousquet[66] followed a diametric approach: instead of assessing the testability thought metrics, she first verified the good practices in testing; therefore, she checked whether such practices are implemented in either models or code.

Different approaches have been proposed to transform and adapt programs in order to facilitate evolutionary testing[67]. McMinn *et al.*conducted a study transforming nested `if` such that the second predicate can be evaluated regardless the first one has been satisfied or not[68]. They showed that the evolutionary algorithm was way more efficient in finding test data for the transformed versions of the program. Similarly, Baresel *et al.*applied a testability transformation approach to solve the problem of programs with loop-assigned flags[69]. Their empirical studies demonstrated that existing genetic techniques were more efficiently working of the modified version of the program.

Test-Case Generation Tools.

In this study, we rely on EvoSuite[7] and Randoop[8]. In last years the automated generation of test cases has caught growing interest, by both researches and practitioners[7,15,13]. Search-based approaches have been fruitfully exploited for such goal[70]. Indeed, current tools have been shown to generate test cases with a high branch coverage and helpful to successfully detect bugs in real systems[71], even if in some cases these test cases are difficult to understand[72,73] or maintain[74]. Proposed approaches can be categorized into two formulations: *single-target* and *multi-target*. In the former, evolutionary algorithms aim to optimize one single coverage target (*e.g.*,, branch) at one time[70,15]. Fraser and Arcuri were the first to propose a multi-target approach, which optimizes all coverage targets simultaneously[75]. They propose the *Whole-suite* (WS) generation. With this technique, GAs evolve entire test suites rather than single test cases. The search is guided by a suite-level fitness function that sums up all the branch distances) in the CUT. It is worth to note that WS is the default approach used by EvoSuite. Following the idea of targeting all branches at once, Panichella *et al.*[13] addressed the test-case generation problem in a many-objective fashion proposing a many-objective genetic algorithm called MOSA. Also this algorithm has been implemented in EvoSuite.

Some approaches try to estimate the difficulty to cover a target during the test-data automatic generation process. Xu *et al.*[76] designed an adaptive fitness function that is based on the branch hardness. They rely on the expected number of visit to compute such a metric for each branch. Their experimental results indicates that the newly proposed fitness function is more effective for some numerical programs.

As well as search-based approaches, also symbolic executions has been fruitfully exploited with the aim to automatically generate test cases[77]. The key idea behind symbolic execution is to use symbolic values, instead of concrete values, to represent the values of program variables as symbolic expressions over the symbolic input values[78]. Modern symbolic executions techniques can be discerned in *concolic testing* and *execution-generated testing* (EGT)[78]. In the former, symbolic execution is performed dynamically, with the program executed on concrete input variable: DART[79] (directed automated random testing) and CUTE[80] are two of the most important tools proposed in the past years falling in this category. The latter maintain a distinction between the concrete and the symbolic state in a program by dynamically checking whether the values involved in a symbolic execution are all concrete. Notable tools belonging to this category are EXE[81] and KLEE[82].

9 | CONCLUSIONS & FUTURE WORK

In a continuous integration environment, knowing *a priori* the coverage that will be achieved by test-data generation tools would allow them to take *informed decisions*. For instance, they would be able to: (i) proper allocate the budget for the generation, reaching the optimal balance between branch-coverage and time saving, and (ii) select the subset of classes for which run the test-case generation. In this paper, we extend our previous work taking the first steps toward the prediction of achievable the branch coverage by test-data generator tools, using machine learning approaches and source-code features experimenting four different search-budget. Due to the non-deterministic nature of the algorithms employed for this purpose, such prediction remains a troublesome task. We selected four different categories of metrics designed to capture the complexity of a given class. Therefore, we performed a large-scale study aiming at comparing four different algorithms trained of such features. This study improves the results we achieved in our previous work: we get a MAE reduction achieved using a Random Forest Regressor and the new set of metrics. Moreover, our fine-grained features analysis proves the importance of coupling-related feature in ensuring a good performance for the prediction. Future efforts will involve both the horizontal and the vertical extension of this work: with the former, we plan to still enlarge the training dataset; with the latter we aim at detecting even more sophisticated features aimed at improving the precision of the model.

ACKNOWLEDGEMENTS

We acknowledge the Swiss National Science Foundation (SNSF) under project named “SURF-MobileAppsData” (Project no. 200021-166275) and the Swiss Group for Software Engineering (CHOOSE).

References

1. Bertolino Antonia. Software Testing Research: Achievements, Challenges, Dreams. In: FOSE ’07:85–103IEEE Computer Society; 2007; Washington, DC, USA.

2. Beizer Boris. *Software Testing Techniques (2Nd Ed.)*. New York, NY, USA: Van Nostrand Reinhold Co.; 1990.

3. Hailpern B., Santhanam P.. Software Debugging, Testing, and Verification. *IBM Syst. J.*. 2002;41(1):4–12.

4. Campos José, Arcuri Andrea, Fraser Gordon, Abreu Rui. Continuous Test Generation: Enhancing Continuous Integration with Automated Test Generation. In: ASE ’14:55–66ACM; 2014; New York, NY, USA.

5. Yoo S., Harman M.. Regression Testing Minimization, Selection and Prioritization: A Survey. *Softw. Test. Verif. Reliab.*. 2012;22(2):67–120.

6. Xu Zhihong. Directed Test Suite Augmentation. In: ICSE ’11:1110–1113ACM; 2011; New York, NY, USA.

7. Fraser Gordon, Arcuri Andrea. EvoSuite: Automatic Test Suite Generation for Object-oriented Software. In: ESEC/FSE ’11:416–419ACM; 2011; New York, NY, USA.

8. Pacheco Carlos, Ernst Michael D.. Randoop: Feedback-directed Random Testing for Java. In: OOPSLA ’07:815–816ACM; 2007; New York, NY, USA.

9. Chidamber S. R., Kemerer C. F.. A Metrics Suite for Object Oriented Design. *IEEE Trans. Softw. Eng.*. 1994;20(6):476–493.

10. Halstead Maurice H, others . *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., New York, NY; 1977.

11. Grano Giovanni, Titov Timofey V., Panichella Sebastiano, Gall Harald C.. How high will it be? Using machine learning models to predict branch coverage in automated testing. In: Fontana Francesca Arcelli, Walter Bartosz, Ampatzoglou Apostolos, Palomba Fabio, eds. *2018 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation, MaLTeSQuE@SANER 2018, Campobasso, Italy, March 20, 2018*, :19–24IEEE Computer Society; 2018.

12. Goldberg David E, Holland John H. Genetic algorithms and machine learning. *Machine learning*. 1988;3(2):95–99.

13. Panichella Annibale, Kifetew Fitsum Meshesha, Tonella Paolo. Reformulating Branch Coverage as a Many-Objective Optimization Problem. In: ICST '15:1–10; 2015.

14. Panichella Annibale, Kifetew Fitsum Meshesha, Tonella Paolo. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering*. 2018;44(2):122–158.

15. Scalabrino Simone, Grano Giovanni, Di Nucci Dario, Oliveto Rocco, De Lucia Andrea. Search-Based Testing of Procedural Programs: Iterative Single-Target or Multi-target Approach?. In: Sarro Federica, Deb Kalyanmoy, eds. *Search Based Software Engineering - 8th International Symposium, SSBSE 2016, Raleigh, NC, USA, October 8-10, 2016, Proceedings*, Lecture Notes in Computer Science, vol. 9962: :64–79; 2016.

16. Palomba Fabio, Panichella Annibale, Zaidman Andy, Oliveto Rocco, De Lucia Andrea. Automatic Test Case Generation: What if Test Code Quality Matters?. In: ISSTA 2016:130–141ACM; 2016; New York, NY, USA.

17. Clark Mike. jdepend: A Java package dependency analyzer that generates design quality metrics https://github.com/clarkware/jdepend.

18. Aniche Mauricio. ck: extracts code metrics from Java code by means of static analysis https://github.com/mauricioaniche/ck.

19. Grano Giovanni, Titov Timofey V., Panichella Sebastiano, Gall Harald C.. *Replication Package - Branch Coverage Prediction in Automated Testing*. https://doi.org/10.5281/zenodo.2548323; doi 10.5281/zenodo.254832; 2019.

20. Sanderson Mark, Croft W. Bruce. The History of Information Retrieval Research. *Proceedings of the IEEE*. 2012;100(Centennial-Issue):1444–1451.

21. Benlarbi Saïda, Emam Khaled El, Goel Nishith, Rai Shesh N.. Thresholds for object-oriented measures. In: Software Reliability Engineering, Proceedings. 11th International Symposium on:24–38IEEE; 2000.

22. Cawley Gavin C, Talbot Nicola LC. On over-fitting in model selection and subsequent selection bias in performance evaluation. *Journal of Machine Learning Research*. 2010;11(Jul):2079–2107.

23. Foundation Apache Software. Apache Cassandra http://cassandra.apache.org.

24. Ant Apache. Apache Ivy http://ant.apache.org/ivy/.

25. Google . Guava https://github.com/google/guava.

26. Google . Guava https://github.com/google/dagger.

27. Commons Apache. Apache-Commons Lang https://commons.apache.org/lang.

28. Commons Apache. Apache-Commons Math https://commons.apache.org/math.

29. Joda-Time . Time http://www.joda.org/joda-time/.

30. Bavota Gabriele, Canfora Gerardo, Penta Massimiliano Di, Oliveto Rocco, Panichella Sebastiano. The Evolution of Project Inter-dependencies in a Software Ecosystem: The Case of Apache. In: ICSM '13:280–289IEEE Computer Society; 2013; Washington, DC, USA.

31. Hampel F.R., Ronchetti E.M., Rousseeuw P.J., Stahel W.A.. *Robust Statistics: The Approach Based on Influence Functions*. Wiley Series in Probability and StatisticsWiley; 2011.

32. Chang Chih-Chung, Lin Chih-Jen. LIBSVM: A Library for Support Vector Machines. *ACM Trans. Intell. Syst. Technol.*. 2011;2(3):27:1–27:27.

33. Nassif Ali Bou, Ho Danny, Capretz Luiz Fernando. Towards an Early Software Estimation Using Log-linear Regression and a Multilayer Perceptron Model. *J. Syst. Softw.*. 2013;86(1):144–160.

34. Random Decision Forests. In: Encyclopedia of Machine Learning and Data Mining. Springer 2017 (pp. 1054).

35. Gayathri M, Sudha A. Software defect prediction system using multilayer perceptron neural network with data mining. *International Journal of Recent Technology and Engineering.* 2014;3(2):54–59.

36. Nassif Ali Bou, Ho Danny, Capretz Luiz Fernando. Towards an Early Software Estimation Using Log-linear Regression and a Multilayer Perceptron Model. *J. Syst. Softw..* 2013;86(1):144–160.

37. Svetnik Vladimir, Liaw Andy, Tong Christopher, Culberson J Christopher, Sheridan Robert P, Feuston Bradley P. Random forest: a classification and regression tool for compound classification and QSAR modeling. *Journal of chemical information and computer sciences.* 2003;43(6):1947–1958.

38. Khoshgoftaar Taghi M, Golawala Moiz, Van Hulse Jason. An empirical study of learning from imbalanced data using random forest. *Tools with Artificial Intelligence, 2007. ICTAI 2007. 19th IEEE international conference on.* 2007;2:310–317.

39. Gray David, Bowes David, Davey Neil, Sun Yi, Christianson Bruce. Using the Support Vector Machine as a Classification Method for Software Defect Prediction with Static Code Metrics. In: Palmer-Brown Dominic, Draganova Chrisina, Pimenidis Elias, Mouratidis Haris, eds. *Engineering Applications of Neural Networks*, :223–234Springer Berlin Heidelberg; 2009; Berlin, Heidelberg.

40. O'brien Robert M. A caution regarding rules of thumb for variance inflation factors. *Quality & quantity.* 2007;41(5):673–690.

41. Riley Richard D, Higgins Julian PT, Deeks Jonathan J. Interpretation of random effects meta-analyses. *Bmj.* 2011;342:d549.

42. Pedregosa F., Varoquaux G., Gramfort A., et al. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research.* 2011;12:2825–2830.

43. Rumelhart D. E., Hinton G. E., Williams R. J.. Learning Internal Representations by Error Propagation. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition.* 1986;1:318–362.

44. Stone Mervyn. Cross-validatory choice and assessment of statistical predictions. *Journal of the royal statistical society. Series B (Methodological).* 1974;36:111–147.

45. Krstajic Damjan, Buturovic Ljubomir J, Leahy David E, Thomas Simon. Cross-validation pitfalls when selecting and assessing regression and classification models. *Journal of cheminformatics.* 2014;6(1):10.

46. Efron Bradley. Estimating the error rate of a prediction rule: improvement on cross-validation. *Journal of the American statistical association.* 1983;78(382):316–331.

47. Hsu Chih-Wei, Chang Chih-Chung, Lin Chih-Jen, others . A practical guide to support vector classification. 2003;.

48. Hall Mark A.. *Correlation-based Feature Selection for Discrete and Numeric Class Machine Learning.* 2000.

49. Pinto Leandro Sales, Sinha Saurabh, Orso Alessandro. Understanding myths and realities of test-suite evolution. *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012.* 2012;:33.

50. Sarro Federica, Petrozziello Alessio, Harman Mark. Multi-objective Software Effort Estimation. In: ICSE '16:619–630ACM; 2016; New York, NY, USA.

51. Baeza-Yates Ricardo, Ribeiro-Neto Berthier, others . *Modern information retrieval.* ACM press New York; 1999.

52. Guyon Isabelle, Elisseeff André. An introduction to variable and feature selection. *Journal of machine learning research.* 2003;3(Mar):1157–1182.

53. Friedman Jerome, Hastie Trevor, Tibshirani Robert. *The elements of statistical learning.* Springer series in statistics New York, NY, USA; 2001.

54. Trivedi Shubhendu, Pardos Zachary A, Heffernan Neil T. The utility of clustering in prediction tasks. *arXiv preprint arXiv:1509.06163.* 2015;.

55. Grabmeier Johannes L, Lambe Larry A. Decision trees for binary classification variables grow equally with the Gini impurity measure and Pearson's chi-square test. *International Journal of Business Intelligence and Data Mining.* 2007;2(2):213–226.

56. Lakhotia Kiran, Harman Mark, Gross Hamilton. AUSTIN: An open source tool for search based software testing of C programs. *Information and Software Technology.* 2013;55(1):112–125.

57. Albert Elvira, Arenas Puri, Gómez-Zamalloa Miguel, Rojas Jose Miguel. Test Case Generation by Symbolic Execution: Basic Concepts, a CLP-Based Instance, and Actor-Based Concurrency. In: Advanced Lectures of the 14th International School on Formal Methods for Executable Software Models - Volume 8483:263–309; 2014; New York, NY, USA.

58. Aniche Maurício, Gerosa Marco Aurélio, Treude Christoph. Developers' Perceptions on Object-Oriented Design and Architectural Roles. In: SBES '16:63–72ACM; 2016; New York, NY, USA.

59. Ferrer Javier, Chicano Francisco, Alba Enrique. Estimating Software Testing Complexity. *Inf. Softw. Technol.*. 2013;55(12):2125–2139.

60. Phogat Manu, Kumar Dharmender, Murthal DCRUST. Testability of Software System. *IJCEM International Journal of Computational Engineering & Management*. 2011;14:10.

61. Shaheen Muhammad Rabee, Du Bousquet Lydie. Is Depth of Inheritance Tree a Good Cost Prediction for Branch Coverage Testing?. In: Advances in System Testing and Validation Lifecycle, 2009. VALID'09. First International Conference on:42–47IEEE; 2009.

62. Gu Dechang, Zhong Yin, Ali Sarwar. On testing of classes in object-oriented programs. *Proceedings of the 1994 conference of the Centre for Advanced Studies on Collaborative research*. 1994;:22.

63. Kout Aymen, Toure Fadel, Badri Mourad. An empirical analysis of a testability model for object-oriented programs. *ACM SIGSOFT Software Engineering Notes*. 2011;36(4):1–5.

64. Khan Raees A, Mustafa Khurram. Metric based testability model for object oriented design (MTMOOD). *ACM SIGSOFT Software Engineering Notes*. 2009;34(2):1–6.

65. Khanna Priyanksha. Testability of object-oriented systems: An AHP-based approach for prioritization of metrics. *Contemporary Computing and Informatics (IC3I), 2014 International Conference on*. 2014;:273–281.

66. Bousquet Lydie. A New Approach for Software Testability. In: Bottaci Leonardo, Fraser Gordon, eds. *Testing – Practice and Research Techniques*, :207–210Springer Berlin Heidelberg; 2010; Berlin, Heidelberg.

67. Harman Mark, Baresel André, Binkley David, et al. Testability Transformation - Program Transformation to Improve Testability. In: Hierons Robert M., Bowen Jonathan P., Harman Mark, eds. *Formal Methods and Testing*, Lecture Notes in Computer Science, vol. 4949: :320–344Springer; 2008.

68. McMinn Phil, Binkley David, Harman Mark. Empirical Evaluation of a Nesting Testability Transformation for Evolutionary Testing. *ACM Trans. Softw. Eng. Methodol.*. 2009;18(3):11:1–11:27.

69. Baresel André, Binkley David, Harman Mark, Korel Bogdan. Evolutionary Testing in the Presence of Loop-assigned Flags: A Testability Transformation Approach. In: ISSTA '04:108–118ACM; 2004; New York, NY, USA.

70. McMinn Phil. Search-based software test data generation: a survey. *Software Testing, Verification and Reliability*. 2004;14(2):105–156.

71. Shamshiri Sina, Just Rene, Rojas Jose Miguel, Fraser Gordon, McMinn Phil, Arcuri Andrea. Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges (T). In: ASE '15:201–211IEEE Computer Society; 2015; Washington, DC, USA.

72. Panichella Sebastiano, Panichella Annibale, Beller Moritz, Zaidman Andy, Gall Harald C.. The Impact of Test Case Summaries on Bug Fixing Performance: An Empirical Investigation. In: ICSE '16:547–558ACM; 2016; New York, NY, USA.

73. Grano Giovanni, Scalabrino Simone, Gall Harald C., Oliveto Rocco. An empirical investigation on the readability of manual and generated test cases. In: Khomh Foutse, Roy Chanchal K., Siegmund Janet, eds. *Proceedings of the 26th Conference on Program Comprehension, ICPC 2018, Gothenburg, Sweden, May 27-28, 2018*, :348–351ACM; 2018.

74. Fraser Gordon, Staats Matt, McMinn Phil, Arcuri Andrea, Padberg Frank. Does Automated Unit Test Generation Really Help Software Testers? A Controlled Empirical Study. *ACM Trans. Softw. Eng. Methodol.*. 2015;24(4):23:1–23:49.

75. Fraser Gordon, Arcuri Andrea. Whole test suite generation. *IEEE Transactions on Software Engineering*. 2013;39(2):276–291.

76. Xu Xiong, Zhu Ziming, Jiao Li. An Adaptive Fitness Function Based on Branch Hardness for Search Based Testing. In: GECCO '17:1335–1342ACM; 2017; New York, NY, USA.

77. Sen Koushik, Marinov Darko, Agha Gul. CUTE: A Concolic Unit Testing Engine for C. *SIGSOFT Softw. Eng. Notes*. 2005;30(5):263–272.

78. Cadar Cristian, Sen Koushik. Symbolic execution for software testing: three decades later. *Communications of the ACM*. 2013;56(2):82–90.

79. Godefroid Patrice, Klarlund Nils, Sen Koushik. DART: directed automated random testing. *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*. 2005;40(6):213–223.

80. Sen Koushik, Marinov Darko, Agha Gul. CUTE: a concolic unit testing engine for C. *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*. 2005;30(5):263–272.

81. Cadar Cristian, Ganesh Vijay, Pawlowski Peter M, Dill David L, Engler Dawson R. EXE: automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)*. 2008;12(2):10.

82. Cadar Cristian, Dunbar Daniel, Engler Dawson. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In: OSDI'08:209–224USENIX Association; 2008; Berkeley, CA, USA.

How to cite this article: Grano G., Titov T.V., Panichella S., and Gall H.C. (2018), Branch Coverage Prediction in Automated Testing, *J. Softw Evol Proc., 2018*