Received 20 July 2018; | Revised 19 December 2018; | Accepted 25 January 2019

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, Binzmühlestrasse 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 all 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 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. We argue that knowing a priori the branch coverage that can be achieved with test-data generation tools can help developers prioritize testing tasks. 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. Our best model shows a 0.2 MAE on nested cross-validation —improving the performance of the model presented in our previous work by the 23%— and it demonstrates the relevance of coupling-related features.

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 should 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.

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 to test, or again, allocate a budget (*i.e.*, time) to devote to the test

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

generation per each class. Knowing *a priori* the coverage that can be achieved by test-data generation tools can help in answering such questions and in better allocating the correspondent resources. As an example, to maximize the branch coverage on the entire system, we might want to prioritize the testing on the classes for which we can achieve a high branch coverage. Similarly, knowing that a critical component has a low predicted branch coverage, we might want to spend more budget on it with the aim to generate better (*i.e.*, higher coverage) tests.

We built a *machine learning* (ML) model to predict the branch coverage that will be achieved by test-data generation tools —such as EvoSuite[7] or Randoop[8]— on a given *class under test* (CUT). With this goal, we select a set of features, representing the complexity of a CUT, that we use to train the machine learning model. Amongst 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. Our final model shows an average *Mean Absolute Error* (MAE) of about 0.2. Considering the performance of the devised model, we argue that it can be practically useful to predict the coverage achieved by test-data generation tools in a real-case scenario. We believe that this approach can support developers in selecting and prioritizing the classes they want to automatically generate tests for.

Contributions of the paper

This paper is an extension of our previous work[11], in which we investigated the possibility to predict the branch coverage achieved by test-data generation tools relying on machine learning. In our extension, we strengthen and enrich the analysis carried out in the previous work. The novel contributions can be summarized as follows:

- We introduce a nested cross validation[12] approach to tune the hyper-parameters of the different machine learning algorithms and at the same time, evaluate and compare their performance. Therefore, we use the same projects to train, validate and test our approach, while in our previous paper[11] we relied on two sets of distinct projects, one for the validation and one for the testing. We believe that such a choice strengthens and reduces the bias of our results.

- We rely on a larger dataset that we use to train and evaluate our machine learning models. We analyze seven large open-source projects for a total of 3'105 Java classes while in our previous work (i) the number of subjects (*i.e.*, classes) was smaller and (ii) we used different projects for the validation and for the test set, threatening the validity of our finding.

- Supervised learning is used to predict a certain outcome from a given input, learning by examples of input/output pairs. In our work, such output is the branch coverage achieved by test-data generator tools on a given class. Due to the non-deterministic nature of the employed algorithms, the branch-coverage value in output of such tools might vary from execution to execution. Therefore, in this extension, we generate a test suite 10 times for each class, averaging the achieved branch coverage in order to have more reliable and stable data. This is an improvement over our previous work[11], where the tools were executed only once per each class.

- In our preliminary work[11] we experimented with three different machine learning algorithms, *i.e.*, Huber Regression, Support Vector Regressor, and Multi-Layer Perceptron. In this new study, we further experiment with the fitting of a Random Forest Regressor as an ensemble algorithm for the branch prediction problem. In summary, we experiment with a total of four different algorithms coming from distinct ML families; moreover, the newly added ensemble algorithm resulted in the best performance of all four.

- In our previous work[11], we used three different categories of features, *i.e.*, *Package Level*, *CK and OO* and *Java Reserved Keyword* features. In this paper, we introduce and experiment with a new subset of metrics, *i.e.*, the Halstead metrics[10]. They aim at determining a quantitative measure of complexity directly from the operators and operands in a class. Moreover, we introduce a fine-grained analysis aiming at understanding the importance of the employed features in the prediction. Our results show that Halstead metrics play an important role in improving the accuracy of the model.

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. Section 6 discusses the main threat while related work are presented in Section 7. At the end, Section 8 concludes the paper, drawing the future work.

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) 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 $\mathrm{A} + \mathrm{I} = 1$

2 | FEATURES USED FOR THE BRANCH COVERAGE PREDICTION

In this study, we consider 79 factors belonging to 4 different categories, that might be correlated with the coverage 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[13] 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[14]. 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 package-level features computed with JDepend[14]. 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 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.*, $\mathrm{A} = 0, \mathrm{I} = 1$.

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[14] 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[15]). Keywords have long been used in Information Retrieval as features[16]. 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 the usage of complexity thresholds, like for instance the CK metrics[17]. 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

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

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 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. 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. 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. [1]

3 | RESEARCH QUESTIONS AND CONTEXT

The *goal* of this empirical study is to define and evaluate machine learning models able to predict the coverage 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. Formally, in this paper we investigate the following research questions:

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

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

RQ1 *Can we leverage on well-established source-code metrics to train machine learning models to predict the branch coverage 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 that 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.

We use a nested cross-validation [12] approach to train, tune, and evaluate four 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 trained and validated two different models, one for EvoSuite and one for Randoop, to investigate eventual differences in the prediction performance of the two.

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].

3.1 | Context Selection

The context of this study is composed by 7 different open source projects: Apache Cassandra [18], Apache Ivy [19], Google Guava [20], Google Dagger [21], Apache-Commons Lang [22], Apache-Commons Math [23] and Joda-Time [24]. We selected those projects due to their different domain; moreover, the Apache-Commons is quite popular in software evolution and maintenance literature [25]. 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. At the end, we use 3,105 different classes as a context of this study.

3.2 | Dataset Construction

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 [26]. More formally, we define $x^{(i)}$ as input variables —also called input *features*— and $y^{(i)}$ as the output —also called *target*

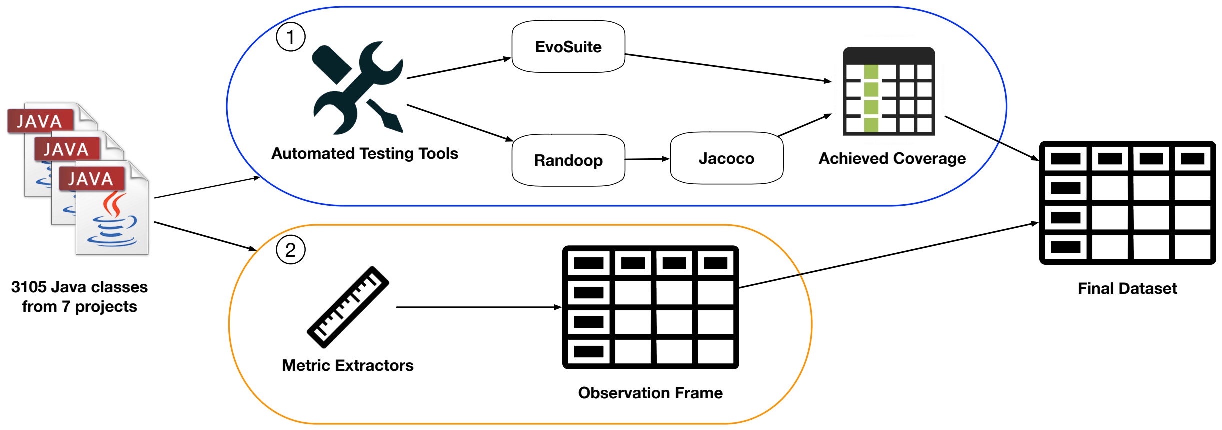

FIGURE 1 Construction process of the training dataset

variable— that we are trying to predict. A pair $(x^{(i)}, y^{(i)})$ is a single training example and the entire dataset used for to learn, *i.e.*, the *training set* is a list of m training examples $\{(x^{(i)}, y^{(i)}),\ \forall i = 1, ..., m\}$. The learning problem consists on learning a function $h = X \to Y$ with a good accuracy of h in predicting the value of y.

In the context of this study, we need to build a dataset consisting of a tuple $(x^{(i)}, y^{(i)})$ for each class i, where:

- $x^{(i)}$ is a vector containing the values for all the factors we described in Section 2, for a given class i;
- $y^{(i)}$ is the branch coverage value —in the range $[0, 1]$— of the test suite generated by a test-data generator tool, for the class i. This is the output variable we want to predict.

The process we use for the construction of the training set is depicted in Figure 1. At first, we execute the scripts needed for the extraction of the factors described in Section 2 (② in Figure 1); those values form the $\{x^{(i)},\ \forall i = 1, ..., m\}$ feature vectors, one for each subject class i. Therefore, to collect the output variable —*i.e.*, the variable we want to predict — we run both EvoSuite and Randoop for 10 times on the CUTs used in the study, obtaining 10 different test suites. Averaging the branch coverage of those suites, we obtain the $\{y^{(i)},\ \forall i = 1, ..., m\}$, where $y^{(i)}$ is the average coverage obtained for the class i. We adopt this approach 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 tool only once per class. At the end of the process, we result with a training dataset $\{(x^{(i)}, y^{(i)}),\ \forall i = 1, ..., m\}$ where $x^{(i)}$ and $y^{(i)}$ are respectively the feature vector and the average coverage achieved over 10 independent runs by test-data generator tools, for a subject class i.

To calculate the coverage for each run we proceed as follows (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. [2] It is worth to note that we discard the points with branch coverage equals to 0. It is also important to underline how time expensive the described process is: we run both EvoSuite and Randoop 10 times for the 3,105 classes we use in the study. Given that we use the default time budget (*i.e.*, 60 and 90 seconds, respectively for EvoSuite and Randoop) the test generation would take about 53 days (21 and 32, respectively for EvoSuite and Randoop) in total with a single processor machine. To speed up such process, we ran the generation on three different 16 cores Ubuntu server, with 64GB of RAM memory each.

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

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

TABLE 5 Statistics about the coverage achieved over the 10 runs of EVOSUITE and RANDOOP. We report (i) minimum coverage, (ii) maximum coverage, (iii) average coverage, and (iv) the average of the standard deviations calculated over the 10 distinct runs for each class

	Min. Coverage	Max. Coverage	μ	Mean σ
EVOSUITE	0.003	1.0	0.744	0.022
RANDOOP	0.005	1.0	0.487	<0.001

3.3 | 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[27], Support Vector Regression[28], Multi-layer Perceptron[29] and Random Forest Regressor[30]. 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[31,32,33,34,35,36]. 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[37] and (ii) its results are easier to interpret than other classifiers[38]. We use the implementation of the Python's ScikitLearn Library[39], being an open source framework widely used in both research and industry.

3.3.1 | Huber Regression

Huber Regression[27] 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[27]. 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.

3.3.2 | Support Vector Regression

Support Vector Regression (SVR)[28] 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`[28]. Amongst the various kernels, we chose a *radial basis function kernel (rbf)*, which can be formally defined as $\exp(-\gamma||x-x'||^2)$, 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, 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[28]. 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.

3.3.3 | Multi-Layer Perceptron

Multi-layer Perceptron (MLP)[40] 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[29]. 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.

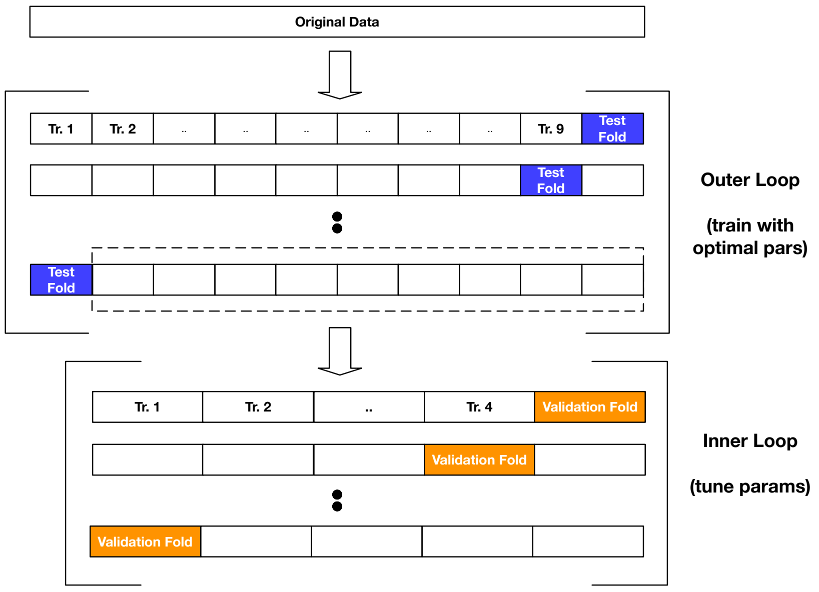

FIGURE 2 Nested Cross-Validation Procedure. The inner fold relies on a 5-fold cross validation while the outer fold on a 10-fold cross validation.

3.3.4 | Random Forest Regressor

Random Forest Regressor (RFR)[30] 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 $g(x) = f_0(x) + f_1(x) + f_2(x) + \ldots + f_n(x)$ 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[37]. We tune this model looking at four different parameters: `n_estimators` represents the number of trees in the forest; `n_features` is the number of features to consider when looking at the best split; `max_depth` is the maximum depth of each decisional tree; `min_sample_leaf` is the minimum number of sample required to be at a leaf node.

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 3.2— we build a training set (i) extracting the described features for the classes under test and (ii) computing the coverage achieved by the testing tools on the same classes. Therefore, we use a nested-cross validation[12] 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[41]. This choice is due to advances achieved in Machine Learning research[41,42] showing that nested cross-validation allows to reliably estimate generalization performance of a learning pipeline involving different steps like preprocessing, features and model selection[42]. 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 bias the model evaluation, leading to overfit the data[12]. 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[43] for the outer loop. The correspondent process is depicted in Figure 2. 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 2). Therefore, the outer loop estimates the generalization error by averaging the test score over several test splits (the boxes in blue on Figure 2).

In detail, in our implementation we apply the well-known *Grid Search* method[44], consisting in training different models to explore the parameter space to find the best configuration. To this aim, we rely on the `GridSearchCV` utility[3] 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[45] 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[44], 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.[4] It is worth to note that, to cope with the randomness arising from using different data splits[46], we repeat the outer cross-validation loop 10 times. We rely on 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. This value is easy to interpret since it is in the same unit of the target variable, *i.e.*, branch coverage fraction. 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)[47].

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 achieved branch coverage. This fine-grained analysis aims at answering our **RQ**$_3$.

In order to detect the most relevant features influencing the prediction, we rely on the *Mean Decrease in Accuracy* (MDA) approach[48]. 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. For $j \in [1, ..., n]$, the jth features is randomly permuted and the accuracy is again computed[49] (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)[47]. 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[15].

5 | RESULTS AND DISCUSSIONS

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, this 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, we focus and discuss the performance of this model in the remaining of the paper. Table 6 shows the performance of the RANDOM

[3]http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html
[4]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 $\in$ 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
```

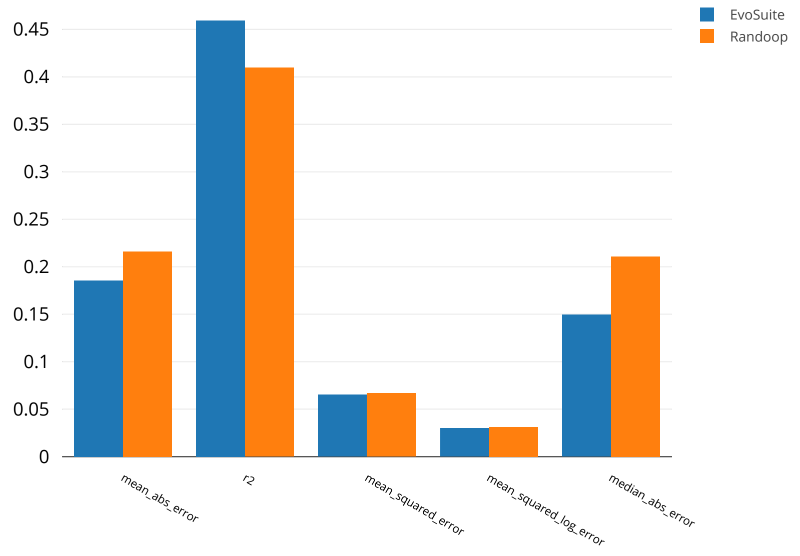

FIGURE 3 Cross-validation metrics for the Random Forest Regressor, reporting respectively Mean Absolute Error, R2 score, Mean Squared Error, Mean Squared Logarithmic Error and Median Absolute Error. The blue bar refers to the model trained on the EvoSuite outcomes, while the orange bar refers to Randoop

Forest Regressor for the five considered evaluation metrics at the end of nested cross-validation. As explained in Section 4, we build two different models, one based on the coverage achieved by EvoSuite and one based on the one achieved by Randoop. Figure 3 reports a grouped box plot showing the same data. In addition we report in Table 7 the optimal combination of parameters resulting by the employed nested cross-validation procedure for both the EvoSuite and Randoop model.

In detail, we observe a MAE of 0.191 and 0.205, respectively for the EvoSuite and Randoop model. Moreover, we report similar performances looking a the other metrics. 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. Therefore, the results we present in this work show a 35% and a 9% MAE reduction for the two models and a 23% MAE reduction on average: thus, we show an improvement in the precision of the model. We believe that the greater improvement of the EvoSuite model, compared to the Randoop one, might depend on the achieved coverage stability over the 10 runs. Indeed, as we show on table 5, EvoSuite has higher standard deviation and therefore, averaging the dependent variable different runs might 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.

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 achieved by automated testing tools. It is also worth

TABLE 6 Results for the RANDOM FOREST REGRESSOR over nested cross-validation. The Mean Absolute Error column also reports the improvement over the original work

	Mean Abs. Error	R2-score	Mean Sqr.Error	Mean Sqr.Log Error	Median Abs.Error
EVOSUITE	0.191 (-35%)	0.473	0.066	0.030	0.152
RANDOOP	0.205 (-9%)	0.411	0.063	0.063	0.189
Total	0.198 (-23%)	0.442	0.065	0.093	0.171

TABLE 7 Optimal parameter combination for the RANDOM FOREST REGRESSOR over nested cross-validation. We set all the other parameters to their default value

model	`n_estimators`	`n_features`	`max_depth`	`min_sample_leaf`
EVOSUITE	20	79	50	2
RANDOOP	4	71	20	1

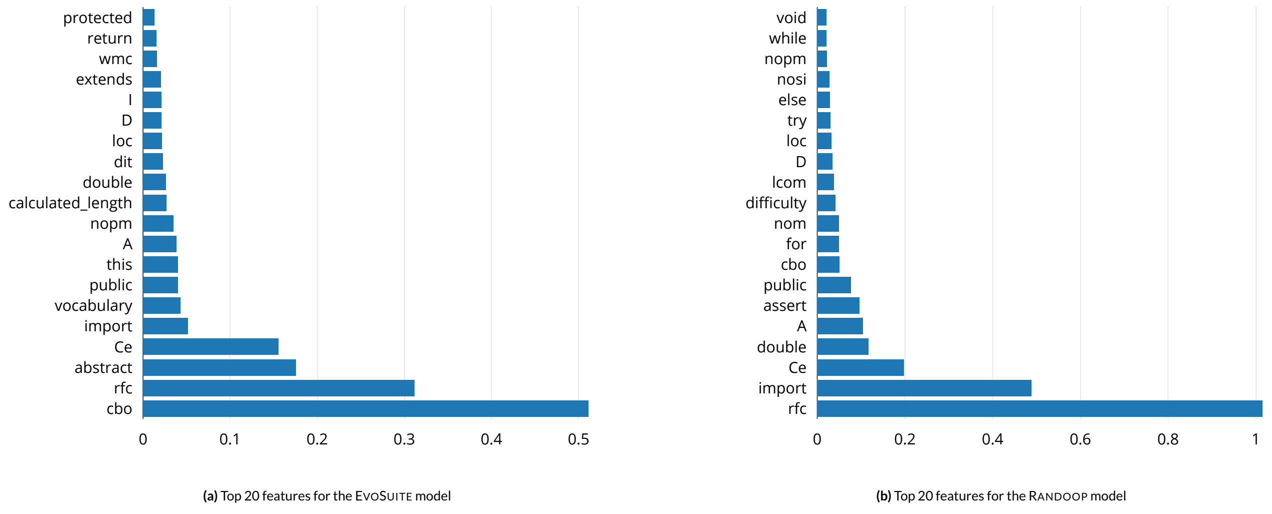

FIGURE 4 Box plot representing the most 20 important feature according to their Mean Decrease in Accuracy (MAE) score. Sub-figure 4a reports such scores for the model trained on EVOSUITE, while Sub-figure 4b reports the stop scores for the RANDOOP model

to note that the RANDOM FOREST REGRESSOR we introduce in this study performs better than the three algorithms investigated in our previous work [11].

In Summary. We confirm the feasibility of code-metrics as features for the achieved coverage prediction problem. RANDOM FOREST REGRESSOR is the most accurate algorithm amongst the considered ones, scoring a 0.191 and a 0.205 MAE respectively for the EVOSUITE and the RANDOOP model. Comparing these performance to the SUPPORT VECTOR REGRESSOR model proved to the best in our previous work, we report an average 23% MAE reduction.

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 4 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 (4a) and by Randoop (4b). 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.

Observing 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 8 and 4 out of the top 20 features. Looking at the Halstead metrics we introduce in this work, we observe that 2 of them (out of 6), namely the *program vocabulary* (n) and the *calculated program length* ($\hat{N}$), are ranked amongst the top 20.

Figure 4b 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 7 out of the 20 metrics, while package-level and keywords features counts for 9 and 3 out of 20, respectively. Regarding the Halstead metrics, for this model only the *program difficulty* (D) metrics is 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)[50]. 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[15]) 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 4a. 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 4b. In this case, another Halstead metric, *i.e.*, *calculated program length* ($\hat{N}$), appears amongst the most important features for the classifier.

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.

6 | 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[27], Support Vector Regression[28], Random Forest[30] and Vector Space Model[29]. 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[51] or symbolic execution[52].

Threats to Conclusion Validity.

To select and validate the best model amongst the 4 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. 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*[48] and on *Gini* index[50].

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 4 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[14]. 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[53]. 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, a larger training set might improve the generalizability of our results.

7 | RELATED WORK

The closer work to what we present in this paper is the one of Ferrer *et al.*[54]. 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[55] 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[56]. 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.*[57] investigated the testability of object-oriented classes. They observed that the most effective test cases consist of a tuple $(\mathtt{s}, \omega)$ where $\mathtt{s}$ is the class state and ω is a sequence of operations applicable to that state. Kout *et al.*[58] adapted the Metric Based Testability Model for Object-Oriented Design (MTMOOD) proposed by Khan *et al.*[59] to assess the testability of classes at the code level. Khanna[60] 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[61] 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[62]. 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[63]. 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[64]. Their empirical studies demonstrated that existing genetic techniques were more efficiently working of the modified version of the program.

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[7,65,66]. Search-based approaches have been fruitfully exploited for such goal[67]. Indeed, current tools have been shown to generate test cases with an high branch coverage and helpful to successfully detect bugs in real systems[68]. Some approaches try to estimate the difficulty to cover a target during the test-data automatic generation process. Xu *et al.*[69] 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.

8 | CONCLUSIONS & FUTURE WORK

In a continuous integration environment, knowing *a priori* the coverage achieved by test-data generation tools might ease important decisions, like the subset (and the order) of classes to test, or the budget to allocate for the every of them. In this paper, we extend our previous work taking the first steps toward the prediction of the branch coverage achieved by test-data generator tools, using machine learning approaches and source-code features. 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 to a 23% average MAE reduction achieved using a Random Forest Regressor. 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: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering: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: Gyimóthy Tibor, Zeller Andreas, eds. *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, :416–419ACM; 2011.

8. Pacheco Carlos, Ernst D. Michael. Randoop: feedback-directed random testing for Java. In: Gabriel Richard P., Bacon David F., Lopes Cristina Videira, Jr. Guy L. Steele, eds. *Companion to the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, :815–816ACM; 2007.

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. 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.

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

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

15. GitHub - Replication Package. https://github.com/sealuzh/branch-prediction.

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

17. 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.

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

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

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

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

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

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

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

25. Bavota Gabriele, Canfora Gerardo, Di Penta Massimiliano, Oliveto Rocco, Panichella Sebastiano. The Evolution of Project Inter-dependencies in a Software Ecosystem: The Case of Apache. In: 2013 IEEE International Conference on Software Maintenance, Eindhoven, The Netherlands, September 22-28, 2013:280–289IEEE Computer Society; 2013.

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

27. 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.

28. 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.

29. 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.

30. Random Decision Forests. In: Springer 2017 (pp. 1054).

31. 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.

32. 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.

33. 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.

34. 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.

35. 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.

36. Jørgensen Magne. Regression Models of Software Development Effort Estimation Accuracy and Bias. *Empirical Software Engineering.* 2004;9(4):297–314.

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

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

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

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

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

42. 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.

43. 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.

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

45. Hall Mark A.. Correlation-based Feature Selection for Discrete and Numeric Class Machine Learning. In: :359–366Morgan Kaufmann; 2000.

46. Pinto Leandro Sales, Sinha Saurabh, Orso Alessandro. Understanding myths and realities of test-suite evolution. In: SIGSOFT FSE:33ACM; 2012.

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

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

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

50. 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.

51. 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.

52. 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.

53. Aniche Mauricio Finavaro, Gerosa Marco Aurélio, Treude Christoph. Developers' Perceptions on Object-Oriented Design and Architectural Roles. In: Almeida Eduardo Santana, ed. *Proceedings of the 30th Brazilian Symposium on Software Engineering, SBES 2016, Maringá, Brazil, September 19 - 23, 2016*, :63–72ACM; 2016.

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

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

56. 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.

57. 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.

58. 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.

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

60. Khanna Priyanksha. Testability of object-oriented systems: An AHP-based approach for prioritization of metrics. In: :273–281IEEE; 2014.

61. 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.

62. 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.

63. 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.

64. 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.

65. 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.

66. Panichella Annibale, Kifetew Fitsum Meshesha, Tonella Paolo. Reformulating Branch Coverage as a Many-Objective Optimization Problem. In: 8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015:1–10IEEE Computer Society; 2015.

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

68. 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.

69. 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.

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*