Lightweight Assessment of Test-Case Effectiveness using Source-Code-Quality Indicators

Giovanni Grano, Fabio Palomba, and Harald C. Gall

Abstract—Test cases are crucial to help developers preventing the introduction of software faults. Unfortunately, not all the tests are properly designed or can effectively capture faults in production code. Some measures have been defined to assess test-case effectiveness: the most relevant one is the mutation score, which highlights the quality of a test by generating the so-called *mutants*, *i.e.*, variations of the production code that make it faulty and that the test is supposed to identify. However, previous studies revealed that mutation analysis is extremely costly and hard to use in practice. The approaches proposed by researchers so far have not been able to provide practical gains in terms of mutation testing efficiency. This leaves the problem of efficiently assessing test-case effectiveness as still open. In this paper, we investigate a novel, orthogonal, and lightweight methodology to assess test-case effectiveness: in particular, we study the feasibility to exploit production and test-code-quality indicators to *estimate* the mutation score of a test case. We firstly select a set of 67 factors and study their relation with test-case effectiveness. Then, we devise a mutation score prediction model exploiting such factors and investigate its performance as well as its most relevant features. The key results of the study reveal that our prediction model only based on static features has 86% of both F-Measure and AUC-ROC. This means that we can estimate the test-case effectiveness, using source-code-quality indicators, with high accuracy and without executing the tests. As a consequence, we can provide a practical approach that is beyond the typical limitations of current mutation testing techniques.

Index Terms—Automated Software Testing, Mutation Testing, Software Quality

1 Introduction

Software testing is a crucial part in the process of evolving and delivering high quality software, especially when catching regression faults [1]. Development teams rely on test case results and code reviews to decide on whether to merge a pull request [2] or to deploy a system [3]; moreover, their productivity is partly dependent on the quality of tests [4]. Thus, being able to assess the reliability of a test case is of a paramount importance for a number of software maintenance and evolution activities.

In recent years, the research community heavily investigated novel approaches for automatically evaluating the quality of tests [5]. Amongst the others, mutation testing is widely recognized as the *high-end* test coverage criteria [5]: the basic concept of mutation testing is the creation of artificially modified versions of the source code, called *mutants* [6]. Changes in the production code are introduced by *mutation operators* with the aim of mimicing real faults [7]; at the end, the test suite is executed against such mutants and evaluated according to the resulting *mutation score*, *i.e.*, the ratio of *detected* (*i.e.*, killed) mutants over the total number of generated ones. Previous studies showed that mutation testing provides developers with a better and trustworthy test-case effectiveness measure with respect to other code coverage criteria (*e.g.*, branch or block coverage) [7], [8].

Despite being so powerful, mutation testing still has one major limitation: it is an extremely expensive activity since it requires (i) the generation of the mutants, (ii) their compilation, and (iii) the execution of the test suites against each of them. Given its nature, this process becomes harder and harder as the size of a system increases during its evolution or when the frequency of commits is high [9].

To address the scalability limitation of mutation testing, researchers investigated three types of approaches [10]: (i) the *"do fewer"* ones, where the goal is to select a subset of mutants to evaluate; (ii) the *"do smarter"* ones, that exploit run-time information to avoid unnecessary test executions; and (iii) the *"do faster"* ones, that aims at reducing the execution time for each single mutant [11]. While these approaches provided some promising results, Gopinath *et al.* [12] showed that most of them do not provide enough practical gain in terms of mutation testing efficiency if compared with a random selection of mutants. As a consequence, *the problem of automatically assessing test-case effectiveness in a timely and efficient manner is still far from being solved.*

In this paper, we present a novel methodology to assess test-case effectiveness, which is orthogonal to existing approaches. Rather than working on the quality or quantity of mutants to generate, we investigate to what extent we can *estimate test-case effectiveness*—as indicated by mutation analysis—by using source-code-quality indicators computed on both production and test code (*e.g.*, quality metrics [13] or code/test smells [14], [15]). It is important to immediately point out that the use of Machine Learning techniques in the context of mutation testing has been initially explored by Zhang *et al.* [16], who proposed a classification model aiming at *selecting* the most powerful mutants by predicting whether a mutant will be killed or

• *G. Grano, F. Palomba and H.C. Gall are with the University of Zurich, Switzerland*
E-mail: {grano, palomba, gall}@ifi.uzh.ch

not. Their technique, therefore, can be seen as an assistant tool for other mutation testing assessment techniques, which can use the model to speed up their performance. The goal of this paper is diametrically different, as we aim at studying the possibility to devise a prediction model able to directly estimate the mutation score of test classes relying on source-code-quality metrics. We argue that such a predictive model might be exploited (i) to limit the use of mutation analysis, *e.g.*, by focusing expensive computations only on some specific tests, and (ii) to support developers in understanding what are the characteristics (*i.e.*, the features used by the prediction model) of both production and test code that limit/boost test-case effectiveness.

With this aim, we firstly conduct an exploratory study to understand the relation between 67 different factors and test-case effectiveness. On the basis of this preliminary study, we then propose and evaluate a Machine Learning model to discern *effective* tests from *non-effective* ones.

The key results of our study reveal significant differences between *effective* and *non-effective* test code with respect to a number of test and production-code factors, which can be thus further explored. A test-code effectiveness prediction model exploiting static code-quality metrics can achieve about 86% of both F-Measure and AUC-ROC; moreover, when compared with a model that also includes *statement coverage* as predictor (*i.e.*, the coverage criterion that is more related to test-code effectiveness according to recent findings [16], [17]), we observe that the use of dynamic information can only provide partial improvements, thus not being extremely needed for obtaining better performance. We argue that the devised static model can be practically useful to assess test-case effectiveness in a real-case scenario, since it does not require the execution of the test cases.

Replication package To enable *full replicability* of this study, we publish all the data extraction and analysis scripts in our replication package [18].

2 Related Work

Mutation testing is an expensive activity and, thus, research has been conducted in last years to reduce its computational cost. In this section, we present an overview of the various approaches presented to achieve such a goal. For the sake of space limitation, we do not discuss previous work having as goal the definition of techniques for improving mutation testing. However, a complete overview on these approaches is available in the survey conducted by Jia and Harman [5].

Offutt and Untch [10] grouped techniques for speeding-up mutation analysis into three distinct categories, *i.e.*, *do fewer*, *do smarter*, and *do faster*. The first category has been the most investigated one: It aims at reducing the number of mutants to be evaluated. Kurtz *et al.* [19] relied on symbolic execution to build static subsumption graphs, where a mutant subsumes another if tests that kill the first also kill the second one. Such graphs are then used to reduce the number of mutants to consider for the mutation analysis. Strug and Strug [20] used machine learning classification algorithm to detect and discard similar mutants. The proposed approach relies on a hierarchical graph representation of mutants, representing a graph kernel using to compute the similarity. Just *et al.* [21] exploited run-time information in order to reduce the number of mutant executions. When two mutants lead to the same state, only one execution is needed, while the other can be inferred. Recently, Gopinath *et al.* [12] empirically showed that common mutation reduction techniques do not give advantage over random sampling, given the tiny effectiveness improvements and the considerable introduced overhead. Zhu *et al.* [11] showed how to improve the efficiency of mutation testing adopting Formal Concept Analysis to cluster mutants and test cases based on reachability (code coverage) and necessity (weak mutation) conditions. Their results show that the approach can speed up mutation analysis up to 94 times, maintaining an accuracy $> 90\%$. The papers discussed above share the goal of reducing the computational time required to apply mutation testing and assess test-case effectiveness. Our work has the same underlying objective: nonetheless, we propose a drastically different alternative, namely a lightweight prediction model exploiting static source-code-quality attributes as opposed to more expensive compression or dynamic approaches.

The closest work to the proposed one is the study of Zhang *et al.* [16]. They devised a classification model relying on 12 static and dynamic features to estimate whether a mutant will be killed or not, showing that code coverage can be a powerful indicator for assessing the quality of single mutants. Therefore, the approach aims at *selecting* the most powerful mutants in order to reduce the overall cost of mutation analysis. While Zhang *et al.* [16] initially showed the suitability of Machine Learning techniques in the context of mutation testing, they limit the approach to the evaluation of the quality of single mutants, as opposed to test cases. Moreover, they need to collect a series of dynamic information about code coverage and execute mutation testing, which still remains computationally costly besides requiring the exploitation of several tools for gathering the features to be used in the model and possibly hampering its applicability in a real-case scenario. Conversely, our work has a different and more comprehensive goal, namely the one of exploiting Machine Learning models to (i) estimate the overall effectiveness of test cases *without* performing any mutation analysis and (ii) support developers in the understanding of the key factors to take into account while developing test cases.

Furthermore, other Machine Learning applications have been experimented for software testing. Daka *et al.* [22] adopted the readability prediction model originally proposed by Buse and Weimer [23] in the context of automatic test case generation with the goal of improving the comprehensibility of the generated tests, while Grano *et al.* [24] preliminarily assessed the feasibility of branch coverage prediction models, showing promising results. Our work can be seen as complementary with respect to these papers, as it aims at predicting test-case effectiveness as measured by mutation score.

Finally, it is worth to remark that the proposed model could be helpful to filter out non-effective test cases. This potentially makes it suitable for improving existing test-case selection, minimization, and prioritization approaches [25]. As an example, the output of our model could be employed within search-based solutions (*e.g.*, [26], [27]) as an additional fitness factor.

3 Empirical Study Variables

The first step of our analysis is the selection of dependent and independent variables.

3.1 Dependent Variable

As dependent variable we use the mutation score, *i.e.*, the percentage of killed mutants over the total of number of generated mutants [5]. The choice is driven by previous research in the field of software testing, which reports mutation score to be the most important code coverage criterion [5] as well as one the most relevant indicators for developers [7], [8]. To compute the mutation score, we rely on PiTest[1] (or PIT). This choice is due to the fact that PIT was found to be the most mature publicly available mutation testing tool [28] and has been shown to limit the generation of equivalent mutants [29]. Moreover, it has been employed by most studies concerning mutation testing in the last years [4], [30], [31]. PIT generates mutants via bytecode manipulation and provides a wide set of built-in mutators. It provides a total of 13 mutation operators: 7 are active by default, *i.e.*, *Conditional Boundary*, *Increments*, *Invert Negatives*, *Math*, *Negate Conditional*, *Return Values* and *Void Method Calls Mutator*; 6 are by default deactivated, *i.e.*, *Constructor Calls*, *Inline Constant*, *Non Void Method Calls*, *Remove Conditionals*, *Member Variable* and *Switch Mutator*. In the context of this work, we apply all the 13 mutators provided by the tool: in this way, we can have a representative set of mutants. Indeed, we consider both line-related operators (*e.g.*, *Invert Negatives*) and class-related ones (*e.g.*, *Member Variable Mutator*), thus covering a wider range of of operators that better simulate the presence of real faults in production code. For the sake of space, we do not report a complete description of the operators in the paper; however, this can be find —along with code examples— in the PIT website.[2] For each production class being mutated, we only execute the corresponding test case—retrieved according to the strategy explained in Section 4.1—rather than executing every time all the test cases of the considered projects (including those that are not related to the mutated production class).

3.2 Independent Variables

In the context of this study, we consider a total of 67 factors along 5 dimensions *i.e.*, *Code Coverage*, *Test Smells*, *Code Metrics*, *Code Smells* and *Readability*. On the one hand, we consider all the code quality dimensions that include metrics computable statically: test smells, production and test-code metrics, code smells, and readability. Such metrics allow us to test whether the test effectiveness can be actually related to source-code quality. On the other hand, we select statement coverage with the aim of evaluating whether it is actually needed to assess test effectiveness (more details later in Section 4). Our final goal is to define a lightweight prediction model only relying on static code-quality features that can be quickly computed on the current version of test classes. Therefore, we exclude from our analysis the so-called process metrics (*e.g.*, code churn or historical metrics about the pass/fail results of the tests). In the following sections, we briefly discuss the selection of the factors and their extraction. For the sake of space, we report the detailed definition of the metrics, as well as the exact versions and parameters used by the data extraction tools, in the wiki page of our replication package [18].

3.2.1 Code Coverage

Code coverage describes the degree to which the production code is executed when a particular test case runs, and has been largely used in software engineering to decide on the quality of a test suite [32]. We compute the *statement* coverage, *i.e.*, the number of production code statements executed by a test case, rather than other types of code coverage (*e.g.*, branch or block coverage) because of several reasons: (i) Gopinath *et al.* [17] showed that this type of coverage is the most related to test-case effectiveness, (ii) it is fast computable by PIT and (iii) it has a direct relation with mutation operators that act at line-level [17].

3.2.2 Test Smells

Test smells represent sub-optimal design or implementation choices applied by developers when defining test cases [15], [33], [34]. On the one hand, previous research showed that the presence of test smells can lead the test code to be less maintainable [35]–[37]. On the other hand, recent work demonstrated that test smells can be related to problems like test flakiness or fault-proneness of test and production code [37], [38]. Thus, test smells may negatively influence the overall ability of a test case to find faults in production code. We investigate 8 different test smell types originating from the catalog by van Deursen *et al.* [15] and that, together with the others included in the catalog, have been shown to be either related to maintainability or effectiveness issues [35], [37], [38], *i.e.*, *Assertion Roulette*, *Eager Test*, *Lazy Test*, *Mystery Guest*, *Sensitive Equality*, *Resource Optimism*, *For Testers Only*, and *Indirect Testing*. A description of these factors is available in the wiki page of our replication package [18].

To detect these smells, we employ the detection tool proposed by Bavota *et al.* [35], which has been employed in several previous works in the area [34], [35], [38]. Unlike other existing detection tools (*e.g.*, [39], [40], or [41]), this tool can identify all the test smells considered in this study with a high precision and recall (88% and 100%, respectively). To ensure the validity of the tool in the context of our study, we re-evaluate the precision of the detector[3] on a statistically significant sample of 330 test smell instances it identified over the considered systems (more details on them in Section 4). Such a set is a 95% statistically significant stratified sample with a 5% confidence interval of the 2,323 total smell instances detected by the tool. The validation has been independently manually conducted by two authors of this paper, who verified each test method and confirmed/refused the recommendation given by the detector. We evaluate the resulting validation agreement using the Krippendorff's alpha Kr_α [42], a test that is generally more reliable than other existing ones (*e.g.*, Cohen's k) [42]. The agreement was equal to 0.94, considerably higher than the 0.80 standard reference score for Kr_α [42]. The remaining

1. http://pitest.org
2. http://pitest.org/quickstart/mutators/

3. The recall cannot be evaluated because of the lack of an oracle of test smells for the considered projects.

instances were discussed until an agreement was reached. As a result, the precision of the approach on our dataset is 85%, thus sufficiently accurate for performing our study.

3.2.3 Production and Test Code Metrics

This set is composed of 21 factors measuring both size and complexity of the code in various ways. We compute those metrics separately for both production and test code. Most of them belong to the object-oriented (OO) metric suite proposed by Chidamber and Kemerer [13], while others capture complementary aspects (*e.g.*, the McCabe cyclomatic complexity [43]) or are an evolution of the original OO metrics (*e.g.*, the LCOM5 defined by Henderson-Sellers [44]). The rationale for using these metrics is twofold. Firstly, larger and more complex production classes might be harder to test, and, as a consequence, writing effective test cases for such classes might be harder [45]. Secondly, large and complex test cases might deeper exercise the *code under test* (CUT), leading to a better fault revelation capability.

3.2.4 Code Smells

Code smells indicate symptoms of the presence of poor design and implementation choices [14]. Previous research demonstrated that they contribute to the technical debt of a system, possible affecting its maintainability [46], [47]. For this reason, we include code smells into the considered factors; our hypothesis is that writing tests for smelly code is harder, and therefore, tests tend to be less effective. In the context of this work, we consider a total of 8 code smells, *i.e.*, *Class Data Should Be Private*, *Complex Class*, *Blob*, *Spaghetti Code*, *Message Chain*, *Long Method*, *Feature Envy*, *Functional Decomposition*. Again, a detailed description of these smells is available in the shared wiki page. The choice of selecting this wide range of code smells is driven by the willingness of reaching a high degree of representativeness with respect to the entire set of code smells available in literature. Indeed, we consider design flaws that affect most of the suboptimal aspects of object-oriented design: from methods and classes having poor cohesion and/or high coupling and complexity to methods and classes presenting symptoms of poor encapsulation or, again, developed using a different programming paradigm. Furthermore, these smells have different levels of granularity and have been shown to be highly harmful for both maintainability [47] and comprehensibility [48], [49].

We employ DECOR [50] to identify instances of the considered code smells. While the authors of DECOR already showed its accuracy (F-measure=≈80%), we also re-evaluated its precision in the context of our work to ensure that this is the right tool to use. We follow a similar process as the one described for test smells: We manually validate a sample composed of 322 code smell instances output by DECOR. Also in this case, the stratified sample is deemed to be a 95% statistically significant (confidence interval=5%) of the 1,967 code smell instances detected. The Kr_α agreement between the two authors was 0.96. In this case, the precision reached 75%: While this value can be considered pretty high, we are aware of the existence of other detection tools that might perform better (*e.g.*, [51], [52]). Nevertheless, we still preferred DECOR because it is lightweight and fast, as opposed to other approaches (*e.g.*, the ones that analyze the entire change history of systems [51]).

3.2.5 Readability

The final dimension investigates the readability of both test cases and production code. Besides being a desirable property to have while performing maintenance and evolution tasks [53], readability-based metrics have been related in the past to the fault-proneness of source code [54]. Thus, it is reasonable to think that might be easier to write effective unit tests for readable production code [55]. At the same time, test cases with poor readability can be harder to be evolved and maintained [55], becoming less effective overtime. To compute the readability scores on both tests and CUTs, we rely on a state-of-the-art model defined by Scalabrino *et al.* [56]. This model improves the seminal work by Buse and Weimer [23] by (i) adding textual-based features, being able to be more precise in the assessment of readability, and (ii) training the model on both production and test code, allowing its usability in both the contexts. The approach computes the continuous readability level $r \in [0, 1]$ as the probability for a given class to be readable. It is worth noting that we employed the original tool made available by Scalabrino *et al.* [56] with the aim of avoiding biases due to re-implementation.

4 RESEARCH QUESTIONS AND CONTEXT

The *goal* of the empirical study is to gain a deeper understanding about the factors that might affect the effectiveness of test cases, *i.e.*, the ability to reveal faults, with the *purpose* of devising an automated approach able to support developers when assessing the goodness of test cases. The *perspective* is of both researchers and practitioners: The former are interested in evaluating the extent to which lightweight code quality indicators can be exploited as an alternative to standard mutation analysis to assess test-case effectiveness; The latter are instead interested in more scalable solutions to be adopted in a real use-case scenario.

To achieve our goal, we formulated three research questions (**RQ**s). The first one represents a *preliminary* analysis of the relation between the 67 independent variables selected and discussed in Section 3 and test-case effectiveness—as indicated by the mutation score. In particular, we aim at understanding whether and to what extent the distribution of the independent variables values differ for test cases having high or low mutation scores. If so, this might possibly indicate a dependence between independent and dependent variables considered that can be further explored:

RQ$_1$. *Is there a relationship between the selected code-quality factors and test-case effectiveness?*

Once established the value of code quality metrics in the context of test-case effectiveness assessment, we move toward the definition of an automated technique, based on Machine Learning approaches [57], able to estimate whether a test is effective or not, based on its mutation score. This leads to our second research question:

RQ$_2$. *To what extent can we estimate the effectiveness of test cases?*

Besides evaluating the mutation score prediction model as a whole, we then conduct a *fine-grained* analysis aimed at investigating which are the most relevant features employed by the devised approach. This can provide further information for developers with respect to the source code aspects to keep under control to make a test case effective. Thus, we formulate our third research question:

RQ$_3$. *What are the important code-quality factors that can indicate that a test case is effective?*

The following sections examine the methodological choices applied to address our three research questions.

4.1 Context Selection

The context of the study is composed of 18 different Java open source projects whose characteristics are reported in Table 1. Specifically, the column *"build"* reports the build tool (either `Maven` or `Gradle`) used by the selected projects; the column *"pairs"* reports the number of $\langle CUT, test\rangle$ analyzed, *i.e.*, the pairs of associated production and test classes; the columns *"LOC CUTs"* and *"LOC Tests"* report the overall size of production and test classes in the considered systems; the column *"mutants"* shows the number of mutants generated for every project. We select such projects as follow: at first, we select 8 Java open source projects from the list of projects used by previous mutation testing studies [7], [58], [59]: these are marked with a '*' symbol in Table 1. Then, we rely on Google BigQuery[4] to select the most popular—based on the number of stars—GITHUB's Java projects in 2017. We include the SQL query in our replication package [18].

4.2 Linking Production to Test Classes

We consider the mutation score achieved by a test case when exercising the correspondent CUTs as a proxy measure of test-case effectiveness. Hence, starting from the JUnit test classes belonging to the considered systems, we need to identify the production class associated with each of them, *i.e.*, we need to link each test class to a production class.

To select such pairs, depending on the build tool, we rely either on the `pom` (for the MAVEN projects) or on the `build.gradle` file (for the GRADLE ones). They contain the rules to identify the test classes to execute when the projects need to be built or packaged. We proceed as follows: At first, we identify all the production and test classes by scanning the `pom` or the `gradle.build` file, *e.g.*, looking for the `sourceDirectory` and `testSourceDirectory` fields, in the former case. They indicate the location of the production and test code, respectively. If those fields are not explicitly reported, we consider the default source and test directories. After that, we use the `include` and `exclude` tag of the `maven-test-plugin` (or of the `test` task, in the case of GRADLE), so that we consider only the test cases that are actually ran when the project is built.

4. https://cloud.google.com/bigquery/

TABLE 1
Characteristics of the projects used for the empirical study

PROJECT	BUILD	PAIRS	LOC CUTs	LOC Tests	MUTANTS
RxJava	Gradle	442	109,978	159,044	21,181
cat	Gradle	62	11,918	5,052	9,850
checkstyle*	Maven	228	61,931	46,995	64,330
closure-compiler*	Maven	308	140,264	165,600	95,742
commons-beanutils*	Maven	56	15,293	20,465	5,542
commons-collections*	Maven	103	27,950	23,344	9,957
commons-io*	Maven	61	11,397	9,088	4,315
commons-lang*	Maven	109	75,160	52,610	39,975
commons-math*	Maven	409	133,248	95,589	88,865
fastjson	Maven	64	30,107	6,376	36,903
gson	Gradle	23	8,691	4,979	6,347
guice	Maven	24	6,641	10,685	2,649
javapoet	Maven	12	3,589	4,938	2,789
jfreechart*	Maven	315	165,631	67,185	86,912
joda-beans	Maven	11	3,939	2,712	3,038
jsoup	Maven	23	9,872	5,861	7,974
junit4	Maven	48	6,898	5,599	3,066
opengrok	Gradle	113	36,342	20,912	25,049
Total	-	**2,411**	**858,849**	**707,034**	**514,484**

In other words, we consider all test cases that developers of the subject systems ran when they test them, discarding those that are likely to be not reliable from the developers' perspective [60]. Once completed this filtering phase, we use a pattern matching approach based on naming conventions to find the production class related to a certain test class, as done in many other previous work [34], [61], [62]. Such an approach has been previously empirically assessed [63], showing an accuracy close to 85%, that is comparable with more sophisticated but less scalable techniques (*e.g.*, the slicing-based approach proposed by Qusef *et al.* [64]). We report in the following an example of `pom` file that refers to the APACHE COMMONS BEANUTILS project.

```
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-surefire-plugin</artifactId>
  <configuration>
    <includes>
      <include>**/*TestCase.java</include>
    </includes>
    <excludes>
      <exclude>**/*MemoryTestCase.java</exclude>
    </excludes>
    ...
</plugin>
```

It declares the include pattern `**/*TestCase.java`. We remove the detected pattern from the test filename and we use the obtained name to match the test with its CUT. For instance, given a test case `DoubleConverterTestCase.java` and the pattern `**/*TestCase.java`, we remove the word *TestCase* to determine the name of the CUT, *i.e.*, `DoubleConverter.java`. While most of the projects use the default `*Test.java` pattern, we rely on the described approach to tackle non-default text matching. In case of no `include` tags, we assume the default behavior.

5 On the Characteristics of Effective Tests

This section reports empirical study design and results aimed at answering to our **RQ$_1$**.

5.1 RQ$_1$ Design: Factors Analysis

With our first research question, we are interested in understanding to what extent the distribution of the values related

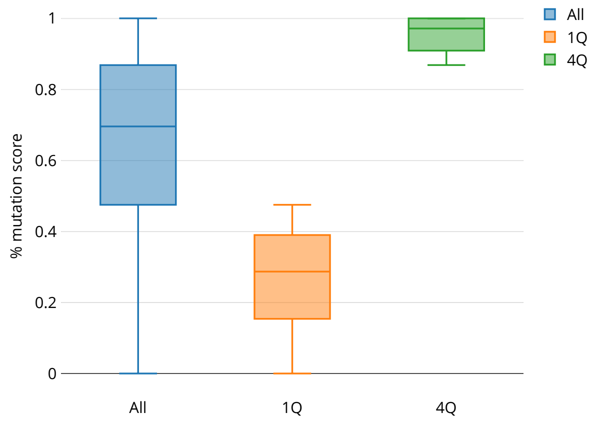

Fig. 1. Distribution of the mutation score for the entire set, the first (non-effective tests) and the fourth quartile (effective tests).

to the 67 considered factors differs for test cases having *high* or *low* mutation scores. To this aim we build two sets of test cases, named *effective* and *non-effective*, starting from the all test cases in the exploited dataset. To assign test cases to one of the two sets, we use the quartiles of the mutation score: those falling within the first quartile are assigned to the *non-effective* set, while the ones in the fourth quartile to the *effective* set. In this process, we discard test cases that fall between the first and fourth quartiles. This is a conscious design decision based on what has been reported in previous software engineering literature [65], [66] with respect to the so-called *discretization noise* [67]. This term refers to the introduction of biases in the data analysis due to the presence of data points that are not clearly assignable to a certain class. Since we aim at studying the characteristics of *effective* and *non-effective* tests, then we accept to not consider test classes having an average effectiveness and focus only on those that can be considered as having a high- or low-quality. Note that the impact of this design decision is further analyzed in Section 8. At the end of this process, the *effective* set is composed of 604 test classes, while the *non-effective* one of 605 tests. It is worth to remember that every test in those set comes with its correspondent CUT.

Figure 1 shows the distribution of mutation scores considering the entire dataset (*"All"*), the first (*"1Q"*), and the fourth (*"4Q"*) quartiles. As it is possible to observe, the *effective* set contains test classes often reaching the maximum mutation score (median = 0.97), meaning that they can actually considered as good test classes able to reveal faults in production code. As for the *non-effective* set, we observe that the mutation score is much more scattered (median = 0.28) and has 0.48 as maximum value: this means that the set contains test classes that are at most able to *"kill"* almost half of the mutants generated on the production class.

To answer **RQ**$_1$, we compare the distribution of each factors in the two sets of test classes applying the Wilcoxon Rank Sum statistical test [68] with α-value = 0.05 as significance threshold. Since we performed multiple tests, we adjusted ρ-values using the Bonferroni-Holm's correction procedure [69]: it firstly sorts the ρ-values resulting from n tests in ascending order of values, multiplying the smallest ρ-value by n, the next by $n-1$, and so on. Then, each resulting ρ-value is then compared with the desired significance level (*i.e.*, 0.05) to determine whether there is statistically significant difference in the distribution of two factors within the two sets of test classes. In the second place, we also estimated the *magnitude* of the observed differences using the Cliff's Delta (or d), a non-parametric effect size measure for ordinal data [70]. We interpret the effect size values following well-established guidelines [70], *i.e.*, small for $0.147 < d < 0.33$, as medium for $0.33 \leq d < 0.474$ and large for $d \geq 0.474$. If the differences in the metric distributions of *effective* and *non-effective* tests are statistically significant and with a large effect size, then we verify the possible existence of a relationship between a certain factor and the effectiveness of test cases.

TABLE 2
Relation between each factor and mutation score. Rel. = relationship. "+" indicates that tests with higher mutation score have significantly higher value on this factor; "-" indicates the opposite case

DIMENSION	METRICS	REL	D-VALUE
Coverage	**statement coverage**	**+**	**0.84 (large)**
Test Smells	Eager Test	-	0.31 (small)
CUT's Code Metrics	**LOC**	**-**	**0.43 (medium)**
	HALSTEAD	**-**	**0.40 (medium)**
	RFC	**-**	**0.62 (large)**
	CBO	**-**	**0.38 (medium)**
	MPC	**-**	**0.58 (large)**
	IFC	-	0.29 (small)
	DAC	**-**	**0.35 (medium)**
	DAC2	**-**	**0.34 (medium)**
	LCOM1	**-**	**0.60 (large)**
	LCOM2	**-**	**0.49 (large)**
	LCOM3	**-**	**0.38 (medium)**
	LCOM4	**-**	**0.49 (large)**
	CONNECTIVITY	-	0.15 (small)
	LCOM5	**-**	**0.39 (medium)**
	COH	**-**	**0.37 (medium)**
	TCC	**-**	**0.33 (medium)**
	LCC	**-**	**0.39 (medium)**
	ICH	**-**	**0.36 (medium)**
	WMC	**-**	**0.61 (large)**
	NOA	**-**	**0.35 (medium)**
	NOPA	-	0.23 (small)
	NOP	**-**	**0.44 (medium)**
	McCABE	**-**	**0.62 (large)**
Test Code Metrics	LOC	+	0.22 (small)
	HALSTEAD	+	0.17 (small)
	RFC	**+**	**0.37 (medium)**
	MPC	**+**	**0.34 (medium)**
	LCOM1	**+**	**0.44 (medium)**
	LCOM2	**+**	**0.40 (medium)**
	LCOM4	**+**	**0.34 (medium)**
	CONNECTIVITY	+	0.25 (small)
	LCC	**+**	**0.35 (medium)**
	ICH	+	0.19 (small)
	WMC	**+**	**0.45 (medium)**
	McCABE	**+**	**0.40 (medium)**
Code Smell	**MC**	**+**	**0.33 (medium)**
	FE	-	0.31 (small)
Readability	production	-	0.19 (small)
	test	-	0.18 (small)

5.2 RQ$_1$ Results: Factors Analysis

Table 2 reports the results achieved for **RQ**$_1$. For the sake of space, we only show the factors having statistically significant difference, *i.e.*, $p < 0.05$ and at least a small effect size. Factors having a $d \geq 0.33$, *i.e.*, at least a medium effect size between *effective* and *non-effective* tests are presented in **bold**. The relationship direction is also reported: A "+" sign indicates a positive relationship, *i.e.*, that tests with higher mutation scores exhibit higher values for the correspondent factor; On the contrary, a "-" sign indicates a negative relationship, *i.e.*, when the test is more effective, the factor tends to be lower.

From the results we can observe that in most cases (41 factors out of 67, *i.e.*, ≈61%) the differences between the distributions of *effective* and *non-effective* tests are statistically significant. Perhaps more importantly, 21 factors report a medium effect size and 8 a large one. This result seems highlighting that the two sets of test classes have relevant differences with respect to the qualitative parameters taken into account, possibly indicating the *relevance of source-code-quality metrics —of both production and test code— for assessing test-case effectiveness*. Deeper investigating the single dimensions, interesting findings arise. The first noticeable one is the relationship between statement coverage and mutation score, which is the strongest achieved in the entire set of collected factors ($d = 0.84$), *i.e.*, the difference in terms of statement coverage between *effective* and *non-effective* tests is large and statistically significant. From the relationship direction we can claim that *the more the statements executed by a test case, the higher its effectiveness*. It is worth noting that this result partially contradicts studies that found code coverage to be *not* associated to the ability of tests to reveal faults in production code [30], [32]. On the contrary, we can rather confirm the findings obtained by Gopinath *et al.* [17] and Zhang *et al.* [16] on the relevance of statement coverage as a metric related of test-case effectiveness.

A second noticeable finding concerns the relation between test-case effectiveness and the metrics representing code complexity. Looking at the metrics computed on the production classes, 20 of them seem to have a relevant impact (*i.e.*, *medium* at least) on test-case effectiveness: These are related to size (*i.e.*, LOC), complexity (*i.e.*, Halstead, RFC, WMC, and McCabe), coupling (*i.e.*, CBO, MPC and DAC), cohesion (*i.e.*, LCOM1, LCOM3, LCOM4, LCOM5, TCC, and LCC). As for size and complexity, our results confirm the "common wisdom" reporting that if the *production code is large and complex, then test cases suffer more and cannot properly reveal faults* [71]. Cohesion and coupling metrics support the result obtained for complexity: indeed, low cohesion and high coupling heavily contribute to the increase of source code complexity as well the decrease of source code maintainability [72], [73].

Similarly, we observe analogous relations between the code metrics computed on test code: indeed, 8 metrics have at least a *medium* impact on test-effectiveness. They are again mostly related to complexity (*i.e.*, RFC, McCabe and WMC), coupling (*i.e.*, MPC) and cohesion (*i.e.*, LCOM1, LCOM2, LCOM5 and LCC). However, in this case the direction of the relations is positive: this seems to suggest a tendency for which *the higher the quality and the complexity of test code, the higher its ability to find faults in the production code*. We cannot confirm this hypothesis based on our results, however we plan future studies on the relation between test-code quality and test-case effectiveness.

Looking at the relation between test-effectiveness and test smells, we do not observe important statistically significant differences. Nevertheless, we show an *almost medium* result for the *Eager Test* smell. Such a smell arises when a test checks more than one method of the class to be tested [15]. As reported in previous literature [41], [74], eager tests are harder to understand, being therefore hardly usable as documentation; moreover, production code tested by tests affected by this smell tends to be more fault-prone [37]. As a direct consequence, the test is likely to not be well-designed to effectively find faults and, at the same time, the production class badly-designed to be tested in isolation.

A similar phenomenon can be observed looking at code smells affecting the production code. Indeed, 2 out of 8 factors in this dimension have a statistically significant negative relationship with mutation score, meaning that *the lower the number of code smells, the higher the ability of tests to find faults*. This result is somehow expected, since code smells indicate the presence of design issues that make the source code more complex and harder to maintain [75], thus making the corresponding tests less effective.

Finally, we do observe a *small* difference in terms of readability between *effective* and *non-effective* tests for both production and test code. This may indicate that writing effective tests might be harder if the production code is less readable; on the other hand, a low test readability can be a symptom of general poor test quality [55].

The results discussed so far estimate test effectiveness as mutation scores obtained by using the 13 operators altogether. We also conduct additional analysis to estimate the impact on the relation between factors and effectiveness for each operator individually. This would indicate if, in practice, it would be more convenient to exploit subsets of the considered factors to estimate the effectiveness of tests with respect to specific mutations. To this aim, we re-run the experiment done in $\mathbf{RQ}_1$ by considering as dependent variable the mutation score achieved when running PIT on individual operators. Our analysis reveals that the strength of the relation between the source code quality factors and the mutation score is way lower than the one obtained when considering the operators altogether. Note that having fewer operators would also decrease the effectiveness of mutation testing itself [7]. This is true for all the individual operators. From a practical perspective, this means that the larger the number of mutation operators used to estimate the effectiveness of a test, the higher the ability of source code quality indicators to be impactful in its estimation. The detailed results of this additional analysis are available in our replication package [18].

In Summary. Effective tests statistically differ from non-effective ones for 41/67 of the investigated factors. A test case tends to be more effective when it has a high statement coverage and does not contain test smells. The absence of design flaws in the CUTs and its quality represent strong factors for test effectiveness.

6 On the Prediction of Effective Tests

Based on the results achieved in $\mathbf{RQ}_1$, in this section we present design and results of the empirical study conducted to answer $\mathbf{RQ}_2$ and $\mathbf{RQ}_3$.

6.1 RQ$_2$-RQ$_3$ Design: Evaluating the Capabilities of a Test-Case Effectiveness Prediction Model

To answer $\mathbf{RQ}_2$ and $\mathbf{RQ}_3$, we (i) devise and evaluate a test-case effectiveness prediction model only exploiting static code quality factors, (ii) compare the latter with a model that includes the statement coverage as independent variable, and (iii) analyze what are the most relevant features that allows the model to estimate the effectiveness of tests. The following subsections detail the methodological steps conducted to answer our research questions.

6.1.1 Independent and Dependent Variables

As *independent variables*, we evaluate two different configurations of the factors selected in Section 3, leading to the construction of two test-case effectiveness prediction models. In the first configuration, we consider all factors: while in our preliminary study we already identified a number of relevant factors that might be potentially used as predictors of test-case effectiveness, it is important to point out that in this research question we are adopting Machine Learning algorithms, which might (i) use different independent variables to properly estimate the dependent variable [57], [76], [77] and (ii) take into account interactions among independent variables, as opposite to the individual statistical tests ran in $\mathbf{RQ}_1$. In other words, it is not said that the relevant factors identified through statistical tests ($\mathbf{RQ}_1$) are the same used by the Machine Learning algorithm to estimate test-case effectiveness ($\mathbf{RQ}_2$). For this reason, we involved all the factors in the construction of the first prediction model, letting the exploited classifier (Section 6.1.2) to decide on the their importance for classification. From now on, we refer to this model as the *dynamic* one.

In the second configuration, we decide to exclude the *statement coverage* as independent variable. This factor represents the only dynamic metric used in the study: as such, in a real-case scenario it might be costly to compute since it requires the execution of all test cases of a software system. For this reason, we aim at measuring the extent to which a prediction model containing *only* statically computable code-quality features can estimate test-case effectiveness as opposed to a model that mixes both static and dynamic analysis being therefore, more computationally intensive. As a side effect of this design choice, we can also evaluate the actual gain (if any) given by code coverage to the performance of the prediction model. In the remaining of this paper, we refer to this model as *static* while we refer to the model exploiting statement coverage as *dynamic*.

As *dependent variable* we adopt the boolean classification of test-case effectiveness coming from $\mathbf{RQ}_1$, *i.e., non-effective* and *effective* tests are based on the first and fourth quartiles of the mutation score distribution, respectively—the effect of discarding other tests is discussed in Section 8.

6.1.2 Selection of the Classifier

As shown in previous literature [78], the classifier used for prediction purposes can strongly influence the model performance. For this reason, we test different classifiers before selecting the one that fits better our prediction model: we compare RANDOM FOREST (RFC), K-NEIGHBORS (KNN), and SUPPORT VECTOR MACHINES (SVM), as these are (i) those more frequently adopted for the prediction of testing-related properties (*e.g.,* Zhang *et al.* [16] and Strug and Strug [20] exploited these algorithms in their works), and (ii) they make different assumptions on the underlying data, as well as have different advantages and drawbacks in terms of execution speed and over-fitting [57]. The outcome of this step is the creation of six combination of prediction models, *i.e., dynamic* and *static* for each classifier, which are trained and evaluated as reported in the following.

6.1.3 Preprocessing Steps

Before being able to properly evaluate the prediction models, some preprocessing steps are required: Song *et al.* [79] proposed a general framework that defines an appropriate learning pipeline that includes (i) data normalization, (ii) feature selection, and (iii) classifier configuration. In our work, we included all these steps as detailed below. We *do not* apply any data balancing strategy [80], since the two classes—*effective* and *non-effective* tests—are naturally balanced (604 vs 605).

Data normalization. In both the configurations of features we perform the feature scaling (a.k.a., data normalization) [81], as recommended in previous works [82], [83]. This technique 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, whose make more likely the possibility that some of them get more influence than they should [81]. 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. This scalarization is important especially for Support Vector Machine algorithms [84], since they assume the data to be in a standard range.

Feature Selection. While RFC is able to automatically filter out non-relevant features—thus avoiding problems related to multi-collinearity [85]—this is not true for KNN and SVM. To perform a fair comparison, in these cases we apply the WRAPPER feature selection algorithm [57], that systematically exercises all the possible subsets of features in order to identify the one giving the best performance.

Classifier Configuration. Finally, we also take into account the problem of configuring the classifiers, as it has a strong impact on the final performance achieved by prediction models [86]. To this aim, we apply to all the models—following the procedure described in the next section—the well-known *Grid Search* method [84], which performs a systematic exploration of the parameter space to find the configuration giving the best performance. We rely on the `GridSearchCV` utility[5] provided by `scikit-learn`.

6.1.4 Training and Testing Procedures

To train and validate the experimented models, we use a nested cross-validation strategy [87]. This selection follows the advances achieved in the field of Machine Learning research [87], [88], which showed that nested cross-validation allows to reliably estimate generalization performance of a learning pipeline involving both parameters tuning and models evaluation. Indeed, model selection without nested cross-validation uses the same data for both the tuning and the evaluation: information might thus leak into the model overfitting the data, depending on the dataset size

5. https://goo.gl/9nj7WS

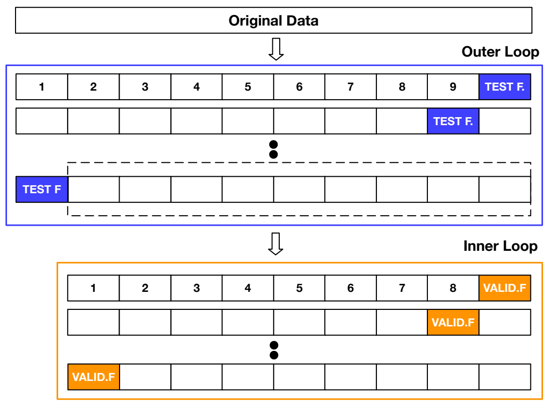

Fig. 2. 10-fold nested cross-validation. The outer loop is contained in the blue box, while the inner loop used for parameters tuning is contained in the orange one.

and on the model stability [89]. Nested cross-validation avoids such a problem by using a set of train, validation and test splits in two separate loops: (1) an *inner* loop, used to tuning the model parameters and (2) an *outer* loop, used to evaluate the performance of the model. To better explain the procedure, Figure 2 shows an example of 10-fold nested cross-validation. Let assume that we aim at tuning and evaluating a certain model M that has a parameter k; and let assume that the goal is to find an assignment of k in the set $\mathcal{K} = \{10, 100, 1000, 10000\}$ that maximize the performance of the model.

The first step adopted by the nested cross-validation consists of dividing the entire set of data in 10 folds. One of these folds is retained as test (the blue box in Figure 2) and left untouched until the end of the computations done in the inner loop. The remaining nine folds are instead used within the inner loop: amongst them, one fold is reserved for the validation (the orange box in Figure 2), while eight of them are used to train M for each $k \in \mathcal{K}$. Once the training phase is completed, the resulting model is evaluated against the validation fold. This gives as output four performance measurements, one for each value of k. The procedure is repeated nine times, allowing each of the nine folds to be the test set exactly once: this leads to $9 \cdot 4$ performance indicators (that is, nine folds multiplied by four possible values of k). Afterwards, the k that minimizes the average training error over the nine folds is selected and used to evaluate M on the test folder previously left out the outer loop. Such a process is then repeated ten times, so that each fold of the outer loop is used as test once. The overall accuracy of the model is finally estimated using the mean of the evaluation measures over the ten test folds, *i.e.*, the model with the best average is chosen.

Nested cross-validation allows to select an arbitrary number of folds for the inner and outer loop. In our study, we rely on 10-folds for both the two loops. To obtain such folds, we use a random stratified split approach: in this technique, each split contains approximately the same percentage of samples of each target class as the complete set. To accurately evaluate the trained models, we rely on 7 different widely-adopted evaluation measures, *i.e.*, accuracy, precision, recall, F1 Score, AUC-ROC, Mean Absolute Error (MAE) and Brier Score [90]. From our experiments, the model based on the RANDOM FOREST classifier performs better than the others in terms of all the evaluation metrics considered. For the sake of space limitations, in the remaining of the paper we only report and discuss the results for this model, while a detailed report of the performance of the models built using the KNN and SVM as classifiers is available in our replication package [18].

6.1.5 *Feature Analysis*

Besides evaluating the test-case effectiveness prediction model as a whole, we also conduct a fine-grained analysis to understand which are the most influential factors it uses to actually estimate the dependent variable. This fine-grained analysis aims at answering to $\textbf{RQ}_3$.

To perform this examination, we rely on the built-in features of RANDOM FOREST: as explained above, the model built using this classifier performs better than the other experimented ones. Thus, we focus our feature analysis based on the characteristics of this specific model. In particular, RANDOM FOREST can automatically select the most relevant features that influence the dependent variable. In doing so, it relies on the so-called *Gini* index (a.k.a., *Mean Decrease in Impurity*) [91], which indicates the relevance of a certain feature in terms of the reduction it provides to the overall entropy of the model, *i.e.*, how much the model gains by having a feature as independent variable. By computing the Gini index for all the considered features during every validation run of the model, we can assess the gain provided by each feature. Then, we can rank the features according to the average Gini index achieved over the 10 different validation runs. The `scikit-learn` implementation of the RANDOM FOREST algorithm stores the information about the Gini index of each feature in the `feature_importances_` vector variable of the model.

It is worth noting that while with the Gini index we can precisely estimate the contribution given by each predictor to the actual predictions performed by the model and understand which factors are more relevant for the outcome, we cannot statistically verify the importance of the features. For this reason, as suggested in literature [92], [93] we complement our feature importance analysis by adopting the Scott-Knott Effect Size Difference (ESD) test [94], which allows us to verify the statistical ranking of the model features with respect to their contribution. This test represents an effect-size aware variant of the original Scott-Knott test [95] that (i) uses hierarchical cluster analysis to partition the set of treatment means into statistically distinct groups according to their influence in the RANDOM FOREST classification, (ii) corrects the non-normal distribution of an input dataset, and (iii) merges any two statistically distinct groups that have a negligible effect size into one group to avoid the generation of trivial groups. To measure the effect size, the tests uses the Cliff's Delta [70]. In this work, we employed the `ScottKnottESD` implementation[6] provided by Tantithamthavorn *et al.* [94].

6.2 $\textbf{RQ}_2$-$\textbf{RQ}_3$ Results: Evaluating the Capabilities of a Test-Case Effectiveness Prediction Model

Table 3 shows the performance of the RANDOM FOREST classifier for the seven considered evaluation metrics. As

6. https://github.com/klainfo/ScottKnottESD

explained in Section 6.1.2, we build two test-case effectiveness prediction models: one containing all the factors (row *dynamic* in the table), one excluding the statement coverage (row *static*), *i.e.*, the only dynamic metrics requiring the execution of the code.

As shown, the model exploiting both coverage and static metrics has extremely strong performance, not only considering the F-Measure (95%) but also when analyzing AUC-ROC, MAE and Brier Score for which we observe values reaching 95%, 0.053 and 0.037, respectively. *These results clearly suggest that Machine Learning methods can be effectively adopted to assess test-case effectiveness.* It is worth to note that we select projects coming from different domains; therefore, we are confident that our prediction model might be generally usable in different contexts.

To better understand the features allowing the model to be so performing, Figure 3 depicts a bar plot showing the most 20 relevant features used by the model together with the information about their Gini index. To show the dominant contribution of the statement coverage, we plot two bars, one for the statement coverage only, and one that stacks the remaining 19 factors. Indeed, statement coverage is the feature providing the major contribution (Gini index=0.7). On the one hand, this result confirms the observations made in **RQ**$_1$, where we found this measure to be the main characteristic discriminating *effective* and *non-effective* tests. On the other hand, we can confirm again previous findings that reported on the usefulness of statement coverage in the context of mutation testing [16], [17]. The second most important feature is represented by the presence of *Assertion Roulette* instances, while other features mainly used by the prediction model to classify *effective* and *non-effective* tests regard both production and test code metrics. In particular, our results show that cohesion, coupling, and complexity of both production and test-source-code are three important aspects that developers should take into account to ensure a high effectiveness of test cases.

Our findings are generally in line with those of **RQ**$_1$, confirming that source code quality indicators can be exploited to discriminate the effectiveness of tests. Nevertheless, we notice some mismatches between the specific features assessed in **RQ**$_1$ and **RQ**$_3$. These are basically due to test and code smells. While in **RQ**$_1$ the *Eager Test* feature had a statistically significant relation with test code effectiveness, in **RQ**$_3$ the smells considered by RANDOM FOREST are *Assertion Roulette* and *Mystery Guest*. The likely reason behind this mismatch is the interaction that occurs between the features: indeed, as shown by Tufano *et al.* [34], the presence of *Assertion Roulette* and *Mystery Guest* instances induce the co-presence of an *Eager Test* instance, while *Assertion Roulette* and *Mystery Guest* provide two orthogonal information on the quality of tests. Thus, the proposed model exploits only two of the three features when predicting the effectiveness of tests. Similarly, the absence of code smells from the set of relevant features adopted by the RANDOM FOREST can be explained by the relations that such smells have with the other production code metrics considered. As an example, the *Message Chains* smell—that had a statistically significant relation with test code effectiveness in **RQ**$_1$—indicates the existence of a long chain of external calls performed by a production method. This smell is naturally related to

TABLE 3
Performance of the RFC on nested cross-validation. We report accuracy (Acc.), precision (Prec.), recall (Rec.), F1 Score (F1), AUC-ROC (AUC), Mean Absolute Error (MAE) and Brier Score (Brier)

	ACC.	PREC.	REC.	F1	AUC	MAE	BRIER
Dynamic	0.948	0.940	0.960	0.949	0.949	0.051	0.035
Static	0.864	0.864	0.865	0.864	0.864	0.137	0.095

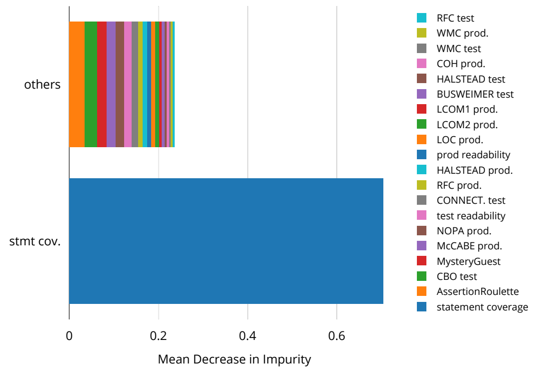

Fig. 3. Feature importance for the Random Forest Classifier with the statement coverage

complexity metrics like the Halstead or readability ones [14]; since our explanatory model considers the metrics altogether, the contribution given by the smell is limited by the co-presence of other complexity metrics.

The discussion on the most relevant variables done so far is also supported by statistical analysis. Indeed, we observe that statement coverage consistently appeared in the top Scott-Knott ESD rank (which was computed on the basis of the Gini index values), followed by the other metrics in the same order as discussed above: for the top 20 factors, the test builds 10 distinct groups, where the group 1, *i.e.*, the most influent, contains the statement coverage only. We report the script and the raw data needed to replicate such a statistical test in our replication package [18].

While the analysis of the *dynamic* model reports that statement coverage represents a key indicator for predicting test-case effectiveness, we also investigate whether its computation is actually needed to obtain good prediction performance. Looking at the results achieved by the *static* model, we can claim that *the exclusion of statement coverage does not drastically decrease the prediction capabilities of the devised model.* More specifically, both F-Measure and AUC-ROC reach 86%, being therefore ≈8% less accurate than the model including the statement coverage, yet still highly performing. This is confirmed by both MAE and Brier coefficients (0.14 and 0.10), that indicates how (i) the prediction error done by the model is pretty limited and (ii) the accuracy of the predictions is high. It is important to point out that the lower performance of the *static* model is expected given the importance of code coverage for mutation testing. However, in our opinion the results achieved by this model are much more important than those of the *dynamic* one from a practical perspective. Indeed, they highlight that *developers can accurately estimate the effectiveness of test cases without actually executing them.*

Figure 4 shows the most relevant features for the *static* model. We observe that it exploits test and code metrics in a more balanced way with respect to the *dynamic*, *i.e.*, there is no feature having a much higher Gini index with respect to the others. In other words, the model has more difficulties in classifying the effectiveness of test cases because of the lack of a strong information like the statement coverage: for this reason it weights features differently in order to gather sufficient knowledge to correctly perform a prediction. This is especially true for the weights assigned by the model to production code attributes: indeed, while in the *dynamic* model the four most relevant variables are all related to test-related characteristics, the *static* one mainly relies on production code complexity factors such the as McCabe and RFC metrics. This means that a fully static model requires different information to balance the lack of statement coverage, yet having high performance.

More in general, 11 of the top 20 features are related to production code size, cohesion, coupling, and complexity. At the same time, it is interesting to observe how also test-code quality comes into play: 9 test-related metrics involving cohesion, coupling, and complexity of tests are still in the top 20 factors according to their Gini index. The results are all statistically significant, and the ranking provided by the Scott-Knott ESD test (reported in our replication package [18]) reflects the most important features discussed so far: indeed, the same top 20 factors are all reported in the first 10 groups created by the test.

To better understand the *static* model performance as compared to the coverage-including one, the first two authors of the paper manually analyze *all* the wrong predictions given by the two models. By relying on (i) the source code of the misclassified tests and (ii) a document reporting the metrics computed on each of them, they perform a code review of the tests aimed at understanding which characteristics may have led to a misclassification. The process is conducted in two joint meetings of eight hours each: this allows the two inspectors to discuss together about the possible reasons behind the errors done by the *static* model with respect to the *dynamic* one. As a result, we first observe that the number of misclassified tests is balanced between the two models considering both false negatives (FN), *i.e.*, tests wrongly predicted as *non-effective*, and false positive (FP), *i.e.*, tests wrongly predicted as *effective*: we have 84 versus 82 FPs and 32 versus 22 FNs for the *static* and *dynamic* models, respectively.

Considering the FN cases of the *static* model, we observe that in 94% of the cases these predictions are biased by factors that characterize *non-effective* tests. For instance, these tests have high values for production-code complexity-metrics (*e.g.*, RFC) and, at the same time, low values for test-code cohesion ones (*e.g.*, LCOM metrics): from **RQ**$_1$ we see that this is the case for non-effective tests. Similarly, the CUTs have high values for complexity metrics, while the same is not true for the correspondent test code. On the contrary, statement coverage is high in 90% of those cases: while the *static* model misclassifies them, the *dynamic* model is instead able to give correct predictions. Also for false positive tests, the *static* model misclassifies the ones having metric values that characterize *effective* tests. Indeed, we observe that 88% of these tests (i) are not smelly, (ii) have a pretty high complexity and coupling values, while the corresponding production code has low complexity, and (iii) the cohesion and coupling metrics for the production code are counterbalanced by similar values in the tests. In these cases, the *static* model misclassifies the tests independently from their level of coverage; on the contrary, the *dynamic* model misclassifies only those tests having a very high coverage. While this qualitative analysis identifies some limitations of the *static* model, it is important to point out that the number of misclassified cases remains limited, thus indicating once again the high ability of source-code-quality indicators to distinguish *effective* from *non-effective* tests.

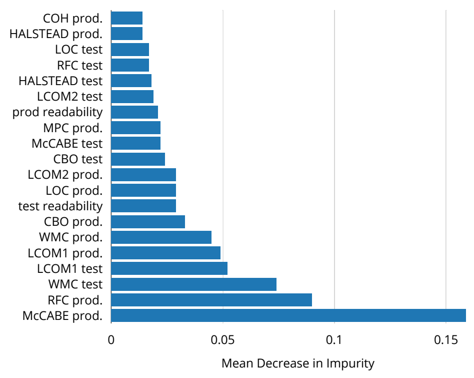

Fig. 4. Feature importance for the Random Forest Classifier without the statement coverage

In Summary. Prediction models can be effectively exploited to classify test-case effectiveness. A model relying on both dynamic and static information achieves performance close to 95% in terms of F-Measure and AUC-ROC, while the performance of a model only using static indicators decreases of ≈9%, yet being highly performing and a more practical solution in a real-case scenario.

7 Discussion

The key result of our study points out the role of source code quality with respect to the effectiveness of test cases. While the results reported so far already demonstrate the accuracy of our automated technique, in this section we further discuss our findings, especially in relation to the motivations behind the achieved results and how the proposed model can be used by practitioners in a real-case scenario.

7.1 Why Source-Code Metrics can Estimate the Test-Effectiveness

The foremost finding of our analysis is concerned with the high ability of static source-code metrics in predicting test-code effectiveness. On the one hand, this is surprising since none of the considered source-code metrics explicitly take into account the specific instruction types where a mutation can be injected (*e.g.*, in `if-statements`). On the other hand, most of the considered factors have a relation with the degree of source-code complexity (that directly impacts, for instance, the number and quality of `if-statements`) as well as other properties of production/test code that might have an effect on the effectiveness of test cases. To better understand the motivations behind the performance of the proposed prediction model, we perform a fine-grained qualitative analysis on the tests of our dataset. More specifically, we cannot proceed in the same way as the qualitative analysis presented in Section 6.2. Indeed, while the number of misclassified tests was relatively low (116 and 104 for the *static* and *dynamic* model, respectively), the number of correctly classified tests—which are the subject of this analysis—is prohibitively large to be analyzed exhaustively (*i.e.*, the static model outputs 1,093 true positive predictions). For this reason, we considered a set of 285 correct predictions: such a set represents a 95% statistically significant sample with a 5% confidence interval of the 1,093 total correct predictions of the model. Then, similarly to the previous qualitative analysis, the first two authors of the paper jointly manually review the source code of the sample tests, having the possibility to also analyze the metric values associated with them. In this case, the process require four meetings of eight hours each and allow them to discuss about the characteristics and peculiarities that make some metrics more effective in estimating the effectiveness of tests. As a result, we identify five main motivation that give a rationale of the obtained performance.

McCabe Cyclomatic Complexity. The McCabe cyclomatic complexity of production code has a large explanatory power, according to the analysis done in **RQ**$_1$. Similarly, it is the most relevant factor employed by RANDOM FOREST when a purely static model is trained (**RQ**$_3$). These findings indicate that an important aspect making test cases effective is represented by the complexity of the source code under test. From a practical point of view, a high complexity of production code indicates the presence of several linearly independent paths [43]. This aspect naturally makes the design of the corresponding test harder, because it should be able to exercise every linearly independent path present in the production code. Considering that the generated mutants can be injected in an arbitrary linearly independent path, this makes complex tests more prone to miss them. Thus, McCabe cyclomatic complexity is confirmed to be a good indicator of testing effort, as indicated in the past [43].

Response for a Class. The second most relevant metric of the purely static model is RFC (Response for a Class), which measures the complexity of a class in terms of method calls. A high RFC indicates that the production class has a number of methods that can potentially be executed in response to a message received by a test. Still in this case, designing effective tests is harder [96] and this seems to have an important consequence: diving into our data, we discovered that in such cases the generated mutants are more prone to be left alive by a test because the production code executes paths that do not include the mutated instructions, *i.e.*, methods that do not contain mutants are exercised instead of the one including the actual mutation.

Coupling-related Metrics. A similar discussion can be delineated in the cases of CBO (Coupling Between Object Classes) and MPC (Message Passing Coupling): both these metrics assess the degree of coupling of a class. High values indicate that a class makes several external invocations, thus reducing the ability of tests to find mutants because the production code executes paths that have not been mutated. In other words, the importance of complexity and coupling metrics tell us that the number of paths possibly executed in the production code represents an important aspect for test-case effectiveness. As a consequence, these metrics are pretty useful for the prediction of the capabilities of a test to find errors in production code.

Halstead Metrics. Another observation can be made when looking at the results achieved by the Halstead metric. It measures the complexity of individual expressions in terms of number of operands and operators. The higher the value of the metric the higher the complexity of the lines of production code, *i.e.*, the likelihood to have complex statements in production code that are composed of multiple operands and operators is higher. This aspect has a direct effect on the likelihood of a test to kill a mutant: indeed, mutants can be injected in operands that are not executed by the test (*e.g.*, the right-hand side of an expression is not executed in case of OR conditions). As such, the *higher* the value of this metric, the *lower* the ability of a test to kill mutants. A similar discussion can be done when considering the readability metric. In its formulation, it considers structural features of the source code (*e.g.*, number of parenthesis in a statement) that are likely to increase the complexity of production lines of code. As a consequence, mutations may be injected in statements that are not actually executed by a test, limiting the effectiveness of the test itself.

Test-Case Design. The overall design of a test impacts its effectiveness. The importance of test size, cohesion, and coupling metrics indicate that non-focused tests have reduced capabilities in finding mutants in production code. Therefore, they do not focus uniquely on testing the correspondent production code, thus, limiting their scope. Recent findings have shown that this lack of focus influences the fault-proneness of production code [37]: our findings support such a thesis by showing that the greedy nature of these tests also produces reduced mutation testing capabilities. This aspect is also influenced by the fact that cohesion (LCOM, LCOM1, LCOM2, COH) metrics play a role, as they have the effect of leading a test to be not focused on a specific portion of code: indeed, tests exercising non-cohesive classes naturally lead to exercise different methods of the production code [34].

All in all, both the performance of the model and the qualitative analysis aimed at understanding the motivations behind our results suggest that keeping source-code-quality under control it is possible to improve the effectiveness of test cases. In the following section, we discuss how practitioners can actually use the output of the proposed model in a real-case scenario.

7.2 On the Practical Usage of the Model

The test-case effectiveness prediction model proposed in this paper has a number of concrete applications in practice. In the first place, it has the potential to raise the *developers' awareness* on the effectiveness of a test suite. We envision the proposed model to be integrated within existing software analytics dashboard (*e.g.*, Bitergia[7]) from which practitioners can diagnose the health status of their test suites, possibly

7. https://bitergia.com

becoming aware of the poor effectiveness of some tests. Differently from existing mutation testing solutions, our technique allows a *lightweight* feedback mechanism that might lead to speed-up the time required to identify non-effective tests. Indeed, most of the metrics that we consider in the *static* model are already computed by software analytics dashboards or can be quickly calculated using widely-adopted static analysis tools like CHECKSTYLE or SONARQUBE: for this reason, the data required by our model is already available to practitioners and this notably eases the construction of the prediction model, as opposed to the execution of dynamic mutation testing tools.

The second practical application comes as a natural consequence of the first one and involves different informed decisions that can be taken by developers. More specifically, the output of the prediction model might be used by practitioners for (i) test selection, *i.e.*, to prevent non-effective tests from running at every commit in Continuous Integration, (ii) scheduling preventive actions to improve non-effective tests, and (iii) running additional mutation analysis to understand which are the specific operators that a non-effective test is not able to identify. To better explain a possible practical usage of the model, let consider the test case shown in Listing 1.

```java
@Test
public void testDao() {
m_appDatabasePruner = lookup(TaskBuilder.class, AppDatabasePruner.
      ID);
Date period = ((AppDatabasePruner) m_appDatabasePruner).
      queryPeriod(-1);

try {
((AppDatabasePruner) m_appDatabasePruner).pruneAppCommandTable(
      period, 1);
((AppDatabasePruner) m_appDatabasePruner).pruneAppSpeedTable(
      period, 1);
} catch (DalException e) {
e.printStackTrace();
}
}
```

Listing 1. Example of a non-effective test case from the `cat` project

This test belongs to the class `AppDatabasePruner` of the `cat` project, and has been classified as *non-effective* by our technique. In this case PIT performed 73 mutations on the production code, 3 of which have been detected by the test; the statement coverage is 24%. The test is clearly *non-effective* for various reasons. At first, the test lacks of focus, as it calls three different production methods, *i.e.*, `queryPeriod`, `pruneCommandTable` and `pruneAppSpeedTable`. Secondly, the test does not contain any assertion statement: it only calls the production methods, without actually verifying their behavior. Moreover, the correspondent production class suffers from poor cohesion (LCOM=45) and high complexity (McCabe=27). On the basis of the corresponding production and test code metrics, our technique estimates this test as *non-effective* with a probability p=0.87.

In the first place, the information given by the devised test case effectiveness model can be immediately exploited to decide on whether to remove a test from the set of tests running in Continuous Integration pipelines. In particular, one of critical problems when performing continuous testing is related to the excessive time, combined with a limited time budget, required to run regression tests [97]: our model may be combined with other criteria (*e.g.*, recency or prior bug-proneness [31]) to improve test reduction techniques and support developers in the removal of those tests whose impact is likely not to produce effects, like the one shown in Listing 1. Similarly, test reduction based on mutation score may be exploited in later stages of the testing process, such as nightly testing or integration testing.

Secondly, practitioners can be preventively informed of the presence of non-effective tests. As the model is based on quality-related attributes, developers can act on non-effective tests to further diagnose and improve their design. As an example, in the case of the test shown in Listing 1, a practitioner might decide on devoting some maintenance effort to make the test more focused on the production code (*e.g.*, by applying an Extract Method refactoring [14]), or even if it worth to apply refactoring operations targeting the correspondent production code (*e.g.*, through an Extract Class refactoring [14] it would be possible to make the production code more cohesive and testable).

Finally, we argue that the proposed prediction model can be adopted as a *complementary* alternative to standard mutation testing tools. More specifically, when a new *non-effective* test is identified, a developer could be interested in further diagnosing the issues of the test by running existing mutation testing tools that provide a fine-grained overview of the reasons preventing the test to catch faults. For instance, while source-code quality can be adopted—using the devised model—for early estimation of test code effectiveness and for understanding the quality-related factors that influence more the estimation, a practitioner can analyze the case of Listing 1 employing PIT to have finer-grained information on the mutation operators that the test is not able to identify. In this way, the execution of more expensive mutation testing tools would be limited to those tests that actually require further investigation. We believe that the reasons above have the potential to make mutation testing more usable in practice.

8 THREATS TO VALIDITY

A number of factors might have threatened our study.

Threats to construct validity. The main threats in this category regard possible imprecisions in the data extraction and analysis process. Besides considering 8 systems that were previously used in mutation testing studies, the dataset selection process was performed by relying on Google BigQuery and aimed at extracting the 10 open-source projects having the highest number of starts. This might have introduced some sort of selection bias [98]: however, our results hold on the entire dataset, including the 8 projects that were not selected based on the number of stars. This make us confident of the ecological validity of our findings; nevertheless, further replications would be desirable. When extracting test classes from the subject systems we only considered those tests available under the `include` tag of the MAVEN `pom` file. We adopted this procedure to exclude tests that are not ran when the `test` or `package` MAVEN commands are executed [60].

To automatically compute the considered factors over the exploited dataset we relied on existing tools. In this regard, we employed tools which have been shown as effective in previous literature [35], [50], [56]. Whenever possible, we also evaluated their suitability in the scope of our study, finding them to be a good choice for us. The prediction model we used to compute the readability factor has been trained on about 600 code snippets from both production and test code [99]: therefore, it can be generally used for both production and test readability. To detect code smells, we relied on DECOR: our re-assessment showed its high accuracy; We do not believe that the results would have changed drastically in case of a more accurate detector.

We approximated test-case effectiveness using the mutation score, relying on the assumption that this measure can be actually representative of the quality of a test case. We did so on the basis of existing literature that clearly demonstrated how mutation score can be considered as the "high-end test coverage criterion" [11], [100]–[102]. Such mutation score has been computed at unit-test level: we cannot exclude that a focus on integration testing would affect the observations provided in this paper. Unfortunately, we do not have data to speculate on this point, since there are no mature toolkits allowing to perform mutation testing at integration level [103]. Nevertheless, we still argue that our unit-test level solution can be useful for developers and testers in order to improve the quality of unit test cases and spread mutation analysis in practice.

To study the characteristics and the capabilities of prediction models in predicting *effective* and *non-effective* tests, we excluded those tests having an average effectiveness—as indicated by the distribution of mutation scores—in order to account for the so-called *discretization noise* [67]. However, this can be considered a threat: indeed, the statistically significant differences found might have biased by the absence of several test classes. To measure the extent to which this factor has influenced our findings, we completely re-ran the analyses made in our study taking into account all tests, considering as *effective* those having a mutation score higher than the median of the distribution, and as *non-effective* those tests whose mutation score was lower than or equal to the median. As a result, we did not observe important differences with respect to the findings discussed herein, meaning that the factors that we found to be important are actually confirmed to be significant even in presence of noise. The scripts in our replication package allows the full replication of this additional analysis.

Equivalent and duplicated mutants represent a common threat for studies involving mutation testing. Determining whether a mutant is equivalent is undecidable [104] and, in practice, can involve considerably human effort. As showed by a recent literature review on mutation testing [11], about half of the studies in the field does not adopt any approach for solving the equivalent mutant problem. The remaining ones rely either on manual analysis [105] or make some assumptions (treating mutants not killed as either equivalent or non-equivalent [106]). Given the enormous amount of mutants involved in our study (over 500,000), a manual evaluation was not a feasible option. Therefore, as done in previous work [106], we assume all mutants not killed as possibly not-equivalent. Previous work estimated 20% of mutants generated by PIT to be equivalent [29]: this might lead to underestimate the mutation scores computed in our work. However, since this study focuses more on using the mutation score to discern effective from non-effective tests, rather than predicting its exact value, we do not believe that this would drastically change our results.

Threats to conclusion validity. When investigating the differences in the factors distributions between *effective* and *non-effective* tests, we employed a well-established statistical test such as the Wilcoxon Rank Sum [68], adjusting its results with the Bonferroni-Holm's correction procedure [69]. Furthermore, we exploited the Cliff's delta test [70] to assess the magnitude of the observed differences.

To estimate test-case effectiveness we compared RANDOM FOREST, K-NEIGHBORS and SUPPORT VECTOR MACHINE. To select and validate the best model we used a nested cross-validation procedure with 10-folds for both the inner and the outer loop: we configured the parameters of each model with the aim of relying on the most effective configuration. Moreover, to reduce interpretation biases and to deal with the randomness arising from using different data splits we repeated the validation 10 times. Then, we exploited a number of evaluation metrics, *i.e.*, F1 Score, AUC-ROC, MAE and Brier Score, 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 RFC prediction model to estimate test-case effectiveness, we relied on the *Gini* index, which has been shown to be an accurate measure [91]. Moreover, we statistically confirmed our observations by exploiting the Scott-Knott ESD test [94].

Threats to external validity. We considered 67 factors related to 5 different categories: of course, there might be other additional factors influencing test-case effectiveness that we did not consider. We plan to enlarge the set of factors (*e.g.*, considering the role of the complexity of code changes [107]) as part of our future work. As for the size of the experiment, we analyzed a dataset composed of 2,411 pairs of test and production classes coming from 18 different software systems using two different build tools. While this already represents a large-scale empirical study, replications targeting different types of projects are still desirable.

9 CONCLUSIONS & FUTURE WORK

In this paper, we first studied the relation between 67 factors —related to both production and test code— and test-case effectiveness, measured by means of mutation score. Then, we devised and evaluated a test-case effectiveness prediction model able to distinguish *effective* and *non-effective* tests.

Summing up, the contributions made are as follow:

1) A large scale empirical study involving 2,411 pairs of test and production classes, aimed at understanding the relation between 67 production and test code metrics and test-case effectiveness. It revealed peculiar characteristics distinguishing *effective* and *non-effective* tests, such as higher statement coverage and higher quality of the corresponding production code.
2) A novel test-case effectiveness prediction model, only based on static factors, *i.e.*, practical to use for developers in a real-case scenario since it does not require the execution of the tests. Such a model is able to achieve about 86% of both F-Measure and AUC-ROC. Moreover, our study reveals that the exclusion of dynamic attributes does not substantially decrease the performance of the prediction model.

3) A comprehensive replication package [18], whose aim is twofold: ensuring the full replication of our study and posing a baseline against which future approaches aimed at more accurately classifying the effectiveness of test code can be tested.

Our future research agenda considers the main output of this work. We aim at enlarging the study by considering (i) how other additional factors influence the effectiveness of test cases, possibly contributing to higher prediction performance, (ii) how the proposed model can be exploited to support different programming languages, (iii) how it can be adopted at higher granularity levels (*e.g.*, integration mutation testing), and (iv) how it can be exploited for other testing-related activities such as test case selection, minimization, and prioritization [25]. Moreover, we plan to work on improving existing code-quality checkers to better support developers during the assessment of software reliability. Finally, we also plan to exploit the technique proposed by Brown *et al.* [108] to investigate how the proposed model works when predicting potential faults that are more closely coupled with changes made by actual programmers.

Acknowledgements

Grano and Gall acknowledge the support of the Swiss National Science Foundation (SNSF) through the Project no. 200021_166275; Palomba acknowledges the support of the SNSF through the Project no. PP00P2_170529.

References

[1] L. S. Pinto, S. Sinha, and A. Orso, "Understanding myths and realities of test-suite evolution," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*.

[2] G. Gousios, A. Zaidman, M.-A. Storey, and A. Van Deursen, "Work practices and challenges in pull-based development: the integrator's perspective," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 358–368.

[3] M. Beller, G. Gousios, and A. Zaidman, "Oops, my tests broke the build: An explorative analysis of travis ci with github," in *Proceedings of the 14th International Conference on Mining Software Repositories*. IEEE press, 2017, pp. 356–367.

[4] Y. Zhang and A. Mesbah, "Assertions are strongly correlated with test suite effectiveness," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 214–224.

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

[6] J. Offutt, "A mutation carol: Past, present and future," *Information and Software Technology*, vol. 53, no. 10, pp. 1098–1107, 2011.

[7] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 654–665.

[8] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *Proceedings of the 27th international conference on Software engineering*. ACM, 2005, pp. 402–411.

[9] T. A. Budd, "Mutation analysis of program test data," Ph.D. dissertation, New Haven, CT, USA, 1980, aAI8025191.

[10] A. J. Offutt and R. H. Untch, "Mutation 2000: Uniting the orthogonal," in *Mutation testing for the new century*. Springer, 2001, pp. 34–44.

[11] Q. Zhu, A. Panichella, and A. Zaidman, "An investigation of compression techniques to speed up mutation testing," in *IEEE Conference on Software Testing, Validation and Verification*, 04 2018.

[12] R. Gopinath, I. Ahmed, M. A. Alipour, C. Jensen, and A. Groce, "Mutation reduction strategies considered harmful," *IEEE Transactions on Reliability*, vol. 66, no. 3, pp. 854–874, 2017.

[13] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on software engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[14] M. Fowler and K. Beck, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[15] A. Van Deursen, L. Moonen, A. van den Bergh, and G. Kok, "Refactoring test code," in *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001)*, 2001, pp. 92–95.

[16] J. Zhang, L. Zhang, M. Harman, D. Hao, Y. Jia, and L. Zhang, "Predictive mutation testing," *IEEE Transactions on Software Engineering*, 2018.

[17] R. Gopinath, C. Jensen, and A. Groce, "Code coverage for suite evaluation by developers," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 72–82.

[18] G. Grano, F. Palomba, and H. C. Gall, "Lightweight Assessment of Test-Case Effectiveness using Source-Code-Quality Indicators - Replication Package," Feb. 2019. [Online]. Available: https://doi.org/10.5281/zenodo.2571468

[19] B. Kurtz, P. Ammann, and J. Offutt, "Static analysis of mutant subsumption," in *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on*, pp. 1–10.

[20] J. Strug and B. Strug, "Machine learning approach in mutation testing," in *IFIP International Conference on Testing Software and Systems*. Springer, 2012, pp. 200–214.

[21] R. Just, M. D. Ernst, and G. Fraser, "Efficient mutation analysis by propagating and partitioning infected execution states," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 2014, pp. 315–326.

[22] E. Daka, J. Campos, G. Fraser, J. Dorn, and W. Weimer, "Modeling readability to improve unit tests," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 107–118.

[23] R. P. Buse and W. R. Weimer, "Learning a metric for code readability," *IEEE Transactions on Software Engineering*, vol. 36, no. 4, pp. 546–558, 2010.

[24] G. Grano, T. Titov, S. Panichella, and H. Gall, "How high will it be? using machine learning models to predict branch coverage in automated testing," in *MaLTeSQuE (Workshop on Machine Learning Techniques for Software Quality Evaluation) 2018, Campobasso, Italy*, 2018.

[25] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *Softw. Test. Verif. Reliab.*, vol. 22, no. 2, pp. 67–120, Mar. 2012. [Online]. Available: http://dx.doi.org/10.1002/stv.430

[26] Z. Li, M. Harman, and R. M. Hierons, "Search algorithms for regression test case prioritization," *IEEE Transactions on software engineering*, vol. 33, no. 4, 2007.

[27] S. Yoo and M. Harman, "Pareto efficient multi-objective test case selection," in *Proceedings of the 2007 international symposium on Software testing and analysis*. ACM, 2007, pp. 140–150.

[28] M. Delahaye and L. Bousquet, "Selecting a software engineering tool: lessons learnt from mutation analysis," *Software: Practice and Experience*, vol. 45, no. 7, pp. 875–891, 2015.

[29] L. Fernandes, M. Ribeiro, L. Carvalho, R. Gheyi, M. Mongiovi, A. Santos, A. Cavalcanti, F. Ferrari, and J. C. Maldonado, "Avoiding useless mutants," in *ACM SIGPLAN Notices*, vol. 52, no. 12. ACM, 2017, pp. 187–198.

[30] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 435–445.

[31] A. Shi, A. Gyori, M. Gligoric, A. Zaytsev, and D. Marinov, "Balancing trade-offs in test-suite reduction," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 246–256.

[32] Y. Wei, B. Meyer, and M. Oriol, "Is branch coverage a good measure of testing effectiveness?" in *Empirical Software Engineering and Verification*. Springer, 2012, pp. 194–212.

[33] G. Meszaros, *xUnit test patterns: Refactoring test code*. Pearson Education, 2007.

[34] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "An empirical investigation

into the nature of test smells," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016, pp. 4–15.

[35] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley, "Are test smells really harmful? an empirical study," *Empirical Software Engineering*, vol. 20, no. 4, pp. 1052–1094, 2015.

[36] F. Palomba, D. Di Nucci, A. Panichella, R. Oliveto, and A. De Lucia, "On the diffusion of test smells in automatically generated test code: An empirical study," in *Proceedings of the 9th International Workshop on Search-Based Software Testing*. ACM, 2016, pp. 5–14.

[37] D. Spadini, F. Palomba, A. Zaidman, M. Bruntink, and A. Bacchelli, "On the relation of test smells to software code quality," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 1–12.

[38] F. Palomba and A. Zaidman, "Does refactoring of test smells induce fixing flaky tests?" in *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*. IEEE, 2017, pp. 1–12.

[39] M. Greiler, A. Van Deursen, and M.-A. Storey, "Automated detection of test fixture strategies and smells," in *Software Testing, Verification and Validation (ICST)*, 2013, pp. 322–331.

[40] F. Palomba, A. Zaidman, and A. De Lucia, "Automatic Test Smell Detection using Information Retrieval Techniques," in *International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, p. to appear.

[41] B. Van Rompaey, B. Du Bois, S. Demeyer, and M. Rieger, "On the detection of test smells: A metrics-based approach for general fixture and eager test," *IEEE Transactions on Software Engineering*, vol. 33, no. 12, pp. 800–817, 2007.

[42] K. Klaus, *Content analysis: An introduction to its methodology*. Sage Publications, 1980.

[43] T. J. McCabe, "A complexity measure," *IEEE Transactions on software Engineering*, no. 4, pp. 308–320, 1976.

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

[45] B. W. Lo and H. Shi, "A preliminary testability model for object-oriented software," in *Software Engineering: Education & Practice, 1998. Proceedings. 1998 International Conference*, pp. 330–337.

[46] P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical debt: From metaphor to theory and practice," *Ieee software*, vol. 29, no. 6, pp. 18–21, 2012.

[47] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia, "On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation," *Empirical Software Engineering*, pp. 1–34, 2017.

[48] M. Abbes, F. Khomh, Y.-G. Gueheneuc, and G. Antoniol, "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension," in *Software maintenance and reengineering (CSMR), 2011 15th European conference on*, 2011, pp. 181–190.

[49] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, and A. De Lucia, "Do they really smell bad? a study on developers' perception of bad code smells," in *Software maintenance and evolution (ICSME), 2014 IEEE international conference on*. IEEE, 2014, pp. 101–110.

[50] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur, "Decor: A method for the specification and detection of code and design smells," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20–36, 2010.

[51] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "Mining version histories for detecting code smells," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 462–489, 2015.

[52] F. Palomba, A. Panichella, A. De Lucia, R. Oliveto, and A. Zaidman, "A textual-based technique for smell detection," in *Program Comprehension (ICPC), 2016 IEEE 24th International Conference on*. IEEE, 2016, pp. 1–10.

[53] K. M. Lui and K. C. Chan, "Pair programming productivity: Novice–novice vs. expert–expert," *International Journal of Human-computer studies*, vol. 64, no. 9, pp. 915–925, 2006.

[54] A. Marcus, D. Poshyvanyk, and R. Ferenc, "Using the conceptual cohesion of classes for fault prediction in object-oriented systems," *IEEE Transactions on Software Engineering*, vol. 34, no. 2, pp. 287–300, 2008.

[55] G. Grano, S. Scalabrino, R. Oliveto, and H. Gall, "An empirical investigation on the readability of manual and generated test cases," in *Proceedings of the 26th International Conference on Program Comprehension, ICPC*, 2018.

[56] S. Scalabrino, M. Linares-Vásquez, D. Poshyvanyk, and R. Oliveto, "Improving code readability models with textual features," in *Program Comprehension (ICPC), 2016 IEEE 24th International Conference on*. IEEE, 2016, pp. 1–10.

[57] D. E. Goldberg and J. H. Holland, "Genetic algorithms and machine learning," *Machine learning*, vol. 3, no. 2, pp. 95–99, 1988.

[58] S. Romano and G. Scanniello, "Smug: a selective mutant generator tool," in *Proceedings of the 39th International Conference on Software Engineering Companion*. IEEE Press, 2017, pp. 19–22.

[59] R. Just, "The major mutation framework: Efficient and scalable mutation analysis for Java," in *Proceedings of the 2014 international symposium on software testing and analysis*. ACM, 2014, pp. 433–436.

[60] M. Beller, G. Gousios, A. Panichella, S. Proksch, S. Amann, and A. Zaidman, "Developer testing in the IDE: Patterns, beliefs, and behavior," *IEEE Transactions on Software Engineering*, vol. PP, no. 99, pp. 1–1, 2017.

[61] Z. Lubsen, A. Zaidman, and M. Pinzger, "Using association rules to study the co-evolution of production & test code," in *Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on*. IEEE, 2009, pp. 151–154.

[62] C. Marsavina, D. Romano, and A. Zaidman, "Studying fine-grained co-evolution patterns of production and test code," in *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*. IEEE, 2014, pp. 195–204.

[63] A. Qusef, G. Bavota, R. Oliveto, A. D. Lucia, and D. Binkley, "Evaluating test-to-code traceability recovery methods through controlled experiments," *Journal of Software: Evolution and Process*, vol. 25, no. 11, pp. 1167–1191, 2013.

[64] A. Qusef, G. Bavota, R. Oliveto, A. De Lucia, and D. Binkley, "Scotch: Test-to-code traceability using slicing and conceptual coupling," in *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*. IEEE, 2011, pp. 63–72.

[65] Y. Tian, M. Nagappan, D. Lo, and A. E. Hassan, "What are the characteristics of high-rated apps? a case study on free android applications," in *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*. IEEE, 2015, pp. 301–310.

[66] J. S. Shirabad, T. C. Lethbridge, and S. Matwin, "Supporting maintenance of legacy software with data mining techniques," in *Proceedings of the 2000 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 2000, p. 11.

[67] C. Sun, A. Shrivastava, S. Singh, and A. Gupta, "Revisiting unreasonable effectiveness of data in deep learning era," in *2017 IEEE International Conference on Computer Vision (ICCV)*, pp. 843–852.

[68] W. Conover, *Practical nonparametric statistics*, 3rd ed., ser. Wiley series in probability and statistics. New York, NY [u.a.]: Wiley, 1999.

[69] S. Holm, "A simple sequentially rejective multiple test procedure," *Scandinavian journal of statistics*, pp. 65–70, 1979.

[70] R. J. Grissom and J. J. Kim, *Effect sizes for research: A broad practical approach*. Lawrence Erlbaum Associates Publishers, 2005.

[71] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *Acm sigplan notices*, vol. 39, no. 12, pp. 92–106, 2004.

[72] S. Henry, D. Kafura, and K. Harris, "On the relationships among three software metrics," *ACM SIGMETRICS Performance Evaluation Review*, vol. 10, no. 1, pp. 81–88, 1981.

[73] L. C. Briand, C. Bunse, and J. W. Daly, "A controlled experiment for evaluating quality guidelines on the maintainability of object-oriented designs," *IEEE Transactions on Software Engineering*, vol. 27, no. 6, pp. 513–530, 2001.

[74] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley, "An empirical analysis of the distribution of unit test smells and their impact on software maintenance," in *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. IEEE, 2012, pp. 56–65.

[75] N. Zazworka, M. A. Shaw, F. Shull, and C. Seaman, "Investigating the impact of design debt on software quality," in *Proceedings of the 2nd Workshop on Managing Technical Debt*. ACM, 2011, pp. 17–23.

[76] A. L. Blum and P. Langley, "Selection of relevant features and examples in machine learning," *Artificial intelligence*, vol. 97, no. 1-2, pp. 245–271, 1997.

[77] G. H. John, R. Kohavi, and K. Pfleger, “Irrelevant features and the subset selection problem,” in *Machine Learning Proceedings 1994*. Elsevier, 1994, pp. 121–129.

[78] D. Di Nucci, F. Palomba, R. Oliveto, and A. De Lucia, “Dynamic selection of classifiers in bug prediction: An adaptive method,” *IEEE Transactions on Emerging Topics in Computational Intelligence*, vol. 1, no. 3, pp. 202–212, 2017.

[79] Q. Song, Z. Jia, M. Shepperd, S. Ying, and J. Liu, “A general software defect-proneness prediction framework,” *IEEE Transactions on Software Engineering*, vol. 37, no. 3, pp. 356–370, 2011.

[80] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, “Smote: synthetic minority over-sampling technique,” *Journal of artificial intelligence research*, vol. 16, pp. 321–357, 2002.

[81] M. A. Hall, “Correlation-based feature selection for machine learning,” Ph.D. dissertation, 1999.

[82] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener, “Defect prediction from static code features: current results, limitations, new approaches,” *Automated Software Engineering*, vol. 17, no. 4, pp. 375–407, 2010.

[83] E. Kocaguneli, G. Gay, T. Menzies, Y. Yang, and J. W. Keung, “When to use data from other projects for effort estimation,” in *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM, 2010, pp. 321–324.

[84] C.-W. Hsu, C.-C. Chang, C.-J. Lin *et al.*, “A practical guide to support vector classification,” 2003.

[85] R. M. O’brien, “A caution regarding rules of thumb for variance inflation factors,” *Quality & Quantity*, vol. 41, no. 5, pp. 673–690, 2007.

[86] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, “Automated parameter optimization of classification techniques for defect prediction models,” in *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*. IEEE, 2016, pp. 321–332.

[87] M. Stone, “Cross-validatory choice and assessment of statistical predictions,” *Journal of the royal statistical society. Series B (Methodological)*, pp. 111–147, 1974.

[88] D. Krstajic, L. J. Buturovic, D. E. Leahy, and S. Thomas, “Cross-validation pitfalls when selecting and assessing regression and classification models,” *Journal of cheminformatics*, vol. 6, no. 1, p. 10, 2014.

[89] G. C. Cawley and N. L. Talbot, “On over-fitting in model selection and subsequent selection bias in performance evaluation,” *Journal of Machine Learning Research*, vol. 11, no. Jul, pp. 2079–2107, 2010.

[90] R. Baeza-Yates, B. Ribeiro-Neto *et al.*, *Modern information retrieval*. ACM press New York, 1999, vol. 463.

[91] J. L. Grabmeier and L. A. Lambe, “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*, vol. 2, no. 2, pp. 213–226, 2007.

[92] S. Kabinna, C.-P. Bezemer, W. Shang, M. D. Syer, and A. E. Hassan, “Examining the stability of logging statements,” *Empirical Software Engineering*, vol. 23, no. 1, pp. 290–333, 2018.

[93] H. Li, W. Shang, Y. Zou, and A. E. Hassan, “Towards just-in-time suggestions for log changes,” *Empirical Software Engineering*, vol. 22, no. 4, pp. 1831–1865, 2017.

[94] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, “An empirical comparison of model validation techniques for defect prediction models,” *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 1–18, 2017.

[95] A. J. Scott and M. Knott, “A cluster analysis method for grouping means in the analysis of variance,” *Biometrics*, pp. 507–512, 1974.

[96] V. R. Basili, L. C. Briand, and W. L. Melo, “A validation of object-oriented design metrics as quality indicators,” *IEEE Transactions on software engineering*, vol. 22, no. 10, pp. 751–761, 1996.

[97] S. Nerur, R. Mahapatra, and G. Mangalaraj, “Challenges of migrating to agile methodologies,” *Communications of the ACM*, vol. 48, no. 5, pp. 72–78, 2005.

[98] H. Borges and M. T. Valente, “What’s in a github star? understanding repository starring practices in a social coding platform,” *Journal of Systems and Software*, vol. 146, pp. 112–129, 2018.

[99] S. Scalabrino, M. Linares-Vásquez, R. Oliveto, and D. Poshyvanyk, “A comprehensive model for code readability,” *Journal of Software: Evolution and Process*, vol. 30, no. 6, p. e1958, 2018.

[100] A. P. Mathur and W. E. Wong, “An empirical comparison of data flow and mutation-based test adequacy criteria,” *Software Testing, Verification and Reliability*, vol. 4, no. 1, pp. 9–31, 1994.

[101] P. G. Frankl, S. N. Weiss, and C. Hu, “All-uses vs mutation testing: an experimental comparison of effectiveness,” *Journal of Systems and Software*, vol. 38, no. 3, pp. 235–253, 1997.

[102] N. Li, U. Praphamontripong, and J. Offutt, “An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage,” in *Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops*, 2009, pp. 220–229.

[103] M. Grechanik and G. Devanla, “Mutation integration testing,” in *Software Quality, Reliability and Security (QRS), 2016 IEEE International Conference on*. IEEE, 2016, pp. 353–364.

[104] A. J. Offutt and J. Pan, “Detecting equivalent mutants and the feasible path problem,” in *Computer Assurance, 1996. COMPASS’96, Systems Integrity. Software Safety. Process Security. Proceedings of the Eleventh Annual Conference on*. IEEE, 1996, pp. 224–236.

[105] M.-H. Liu, Y.-F. Gao, J.-H. Shan, J.-H. Liu, L. Zhang, and J.-S. Sun, “An approach to test data generation for killing multiple mutants,” in *Software Maintenance, 2006. ICSM’06. 22nd IEEE International Conference on*. IEEE, 2006, pp. 113–122.

[106] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, “Prioritizing test cases for regression testing,” *IEEE Transactions on software engineering*, vol. 27, no. 10, pp. 929–948, 2001.

[107] A. E. Hassan, “Predicting faults using the complexity of code changes,” in *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*. IEEE, 2009, pp. 78–88.

[108] D. B. Brown, M. Vaughn, B. Liblit, and T. Reps, “The care and feeding of wild-caught mutants,” in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 511–522.

Giovanni Grano has been, since November 2016, a research assistant at the University of Zurich, in the software and evolution architecture group lab, lead by Prof. Harald Gall. His main research goal is to automate and support developers during testing-related tasks with the aim to increase the quality of testing. His research interests include Search-Based Software Engineering (SBSE), with a main focus on Search-Based Software Testing (SBST), Software Maintenance and Evolution and Empirical Software Engineering.

Fabio Palomba is a Senior Research Associate at the University of Zurich, Switzerland. He received the PhD degree in Management & Information Technology from the University of Salerno, Italy, in 2017. His research interests include software maintenance and evolution, empirical software engineering, and source code quality. He serves and has served as a program committee member of various international conferences and as referee for flagship journals in the fields of software engineering.

Harald C. Gall is Dean of the Faculty of Business, Economics, and Informatics at the University of Zurich (UZH). He is professor of software engineering in the Department of Informatics at UZH. His research interests are software evolution, software architecture, software quality, and cloud-based software engineering. Since 1997, he has worked on devising ways in which mining there repositories can help to better understand and improve software development.