Better Prediction of Mutation Score

SUBMISSION DATE / POSTED DATE

03-07-2021 / 14-07-2021

CITATION

Gil, Yossi; Ma'ayan, Dor (2021): Better Prediction of Mutation Score. TechRxiv. Preprint. https://doi.org/10.36227/techrxiv.14905032.v1

DOI

10.36227/techrxiv.14905032.v1

Better Prediction of Mutation Score

Joseph Gil, and Dor Ma'ayan
Computer Science Department, Technion

Abstract—Mutation score is widely accepted to be a reliable measurement for the effectiveness of software tests. Recent studies, however, show that mutation analysis is extremely costly and hard to use in practice. We present a novel direct prediction model of mutation score using neural networks. Relying solely on static code features that do not require generation of mutants or execution of the tests, we predict mutation score with an accuracy better than a quintile. When we include statement coverage as a feature, our accuracy rises to about a decile. Using a similar approach, we also improve the state-of-the-art results for binary test effectiveness prediction and introduce an intuitive, easy-to-calculate set of features superior to previously studied sets. We also publish the largest dataset of test-class level mutation score and static code features data to date, for future research. Finally, we discuss how our approach could be integrated into real-world systems, IDEs, CI tools, and testing frameworks.

Index Terms—Software Engineering, Mutation Analysis, Testing, Empirical Studies.

1 Introduction

The research community continues to invest significant efforts in finding trustworthy metrics and methodologies to assess software test quality. Among the studied metrics, mutation score [1], [2] was broadly accepted as a reliable and accurate indicator of test effectiveness and as the high-end test coverage criterion.

The concept of mutation testing for evaluating the quality of tests (see, e.g., [3] for an accessible introduction, or [4], [5] for a survey) was encapsulated as follows: A *mutation tool* configured by a set of *mutation operators* consumes as input *production code* and generates in return a large number of *mutants* of the input, i.e., variations of the input that are supposed to mimic real-world faults [6]. Then, the quality of the test suite associated with the input is assessed based on the number of mutants the suite is able to kill, i.e., disqualify as not passing the tests.

Studies show that mutation analysis provides a more accurate and trustworthy measure than other code coverage criteria [6]–[9].

To carry out mutation testing, one needs to run the entire test suite against all mutants. Therefore, full mutation testing tends to be costly. If the size of the production code is n, then the number of mutants tends to be $O(n)$ (see Tables 1 and 3 for actual numbers). Given that good test suites are in the same order of magnitude as the code (real-world numbers are presented in Tables 1 and 3 as well), the required resources are likely to be quadratic [3]. This limitation prevents mutation analysis from being widely used outside of academic research.

Offutt and Unch [10] asserted that there are three types of strategies for reducing the cost of mutation analysis: (i) *Do fewer*: seek ways of running fewer mutant programs without incurring excessive information loss, (ii) *Do smarter*: seek to distribute the computational expense over several machines or factor the expense over several executions, and (iii) *Do faster*: focus on ways of generating and running each mutant program as quickly as possible.

Do fewer is the most studied category to date, with many works trying to reduce mutants by data flow heuristics and machine learning techniques [11]–[13]. These approaches present some promising results. Nevertheless, recent studies by Gopinath et al. [14], [15] show that mutation reduction techniques are not superior to random sampling of mutants. Therefore, finding a way to estimate mutation score accurately and fast is still an open problem.

A recent paper by Grano et al. [16] suggested a drastically different approach. Using classic machine learning algorithms with a light set of source code quality metrics [17], and smells [18], [19], they assess the *effectiveness* of tests. Their binary classification model identifies effective and ineffective tests with good accuracy. We take their pioneering efforts a few steps further and show that:

1) neural networks improve the state of the art [16] in test effectiveness prediction,
2) neural networks elicit a quite accurate *direct prediction* of the mutation score itself (rather than a binary classification of effectiveness) in a test-class level, and,
3) an even lighter set of code features can be used for both tasks, which is superior both in prediction accuracy and engineering trade-offs, making it more applicable for real-world usage.

In addition, we introduce the largest dataset available to date of test-class level mutation score together with static code features, with nearly 3,800 pairs of test and production classes, accompanied by the mutation score, code coverage, and 130 various static features. We believe that this dataset will be useful for future studies in the field and for real-world applications.

Finally, we discuss the practical and theoretical implications of our results and offer potential future research directions and opportunities for practitioners.

Outline. The remainder of this paper is organized as follows. Sect. 2 provides background and a short review of the results obtained by Grano et al. [16]. Taking these

into consideration, we phrase specific research questions in Sect. 3. Sect. 4 describes our experimental setting. Answers to the questions from Sect. 3 are then presented in Sects. 5 to 8. Sect. 9 discusses threats to the validity of the results and Sect. 10 discusses our results and concludes.

2 Background

2.1 Using Machine Learning for Mutation Analysis

The use of machine learning in software engineering [20] is increasing at a staggering rate. Applications include generating code summaries [21], learning APIs [22], detection of malware [23] and vulnerabilities [24], program synthesis [25], automatically fixing method names [26], identifying meaningful code changes [27], fault localization, and many more.

Two features characterize the vast majority of the above-mentioned work. Firstly, the learned information pertains to the code's more static aspects, rather than to the measurement of its runtime performance. Secondly, the learning algorithms treat the code as a sequence of words, tokens, or paths along an Abstract Syntax Tree (AST). This trend has started to change, with new works using machine learning to predict dynamic aspects of the code as well, such as branch coverage [28] and mutation score.

Zhang et al. [29] developed a machine learning model for predicting the likelihood that a mutant survives. Their model is based on 12 static and dynamic features and shows a high correlation between statement coverage and a single mutant's quality. In a later work, Mao et al. [30] extended the set of features used by Zhang et al. and applied deep neural network models, improving both the accuracy of failure prediction and the time resources required for making these predictions. Zhang et al. [31] presented an unsupervised learning approach for predicting test effectiveness, which consumes as an input a test suit, production code, and a set of mutants and predicts for each of the mutants whether it will be killed or not.

All the described approaches, however, are limited to evaluating the quality of a single mutant, as opposed to the quality of test cases. Furthermore, they require a series of dynamic information and real-time generation and compilation of the mutants, which are computationally costly and require the usage of several tools. All of the above hampers the integration of these approaches in industry and real-world usage.

In contrast to these works, we stress that our approach is different: rather than focusing on specific mutants, which requires generating the mutant, we aim to predict a class's entire mutation score without referring to specific mutants. Moreover, we present models that avoid needing dynamic information and do not require test suit execution at all, making the prediction real-time and allowing immediate feedback for developers.

An orthogonal approach to the described efforts is to design prediction models for the *effectiveness* of tests, i.e., classification of tests into categories that estimate their mutation score at a high level based on code features and without any mutant execution. Jalbert et al. [32] suggested a classification model with three categories of effectiveness (small, medium, and high), which had limited success since it used a small dataset of eight projects. Grano et al. [16] continued these efforts, as described in the following subsection.

2.2 GPG

Here, we review the work and contribution of Grano, Palomba, and, Gall [16] (GPG). In Sect. 3, we describe our understanding of this contribution and the research questions that motivated our research.

Corpus. The GPG study was based on a corpus of 18 open-source Java projects (see Table 1). The corpus was mined using build tool (maven or gradle) information and class name matching to associate each test case with the tested data. This mining generated circa 2,400 data points, each being a pair of classes: a *test class* adjoined with the *production class* for which the test was intended.

TABLE 1: Corpus of 18 Java open-source projects [16]: Size and other essential statistics

#	Project	#Pairs	|production|*	|test|	$\frac{\|test\|}{\|production\|}$	#Mutants	$\frac{\#Mutants}{production}$
1	RxJava	442	109,978	159,044	145%	21,181	19%
2	cat	62	11,918	5,052	42%	9,850	83%
3	checkstyle	228	61,931	46,995	76%	64,330	104%
4	closure-compiler	308	140,264	165,600	118%	95,742	68%
5	commons-beanutils	56	15,293	20,465	134%	5,542	36%
6	commons-collections	103	27,950	23,344	84%	9,957	36%
7	commons-io	61	11,397	9,088	80%	4,315	38%
8	commons-lang	109	75,160	52,610	70%	39,975	53%
9	commons-math	409	133,248	95,589	72%	88,865	67%
10	fastjson	64	30,107	6,376	21%	36,903	123%
11	gson	23	8,691	4,979	57%	6,347	73%
12	guice	24	6,641	10,685	161%	2,649	40%
13	javapoet	12	3,589	4,938	138%	2,789	78%
14	jfreechart	315	165,631	67,185	41%	86,912	52%
15	joda-beans	11	3,939	2,712	69%	3,038	77%
16	jsoup	23	9,872	5,861	59%	7,974	81%
17	junit4	48	6,898	5,599	81%	3,066	44%
18	opengrok	113	36,342	20,912	58%	25,049	69%
	Total	2,411	858,849	707,034	82%	514,484	60%
	Min	11	3,589	2,712	21%	2,649	19%
	Mean	134	47,714	39,280	84%	28,582	63%
	Median	64	27,950	20,465	76%	9,957	63%
	Max	442	165,631	165,600	161%	95,742	123%

* All code size values are in lines of code (LOC)

Generation of mutants and mutation score. After assembling the corpus and collecting data points, the authors proceeded to carry out mutation testing for the corpus. First, by generating mutants using PIT[1] [33]—arguably, the most robust publicly available mutation testing analysis tool [34], and the tool of choice in several recent publications, e.g., [13], [35]. Another reason for the authors selecting PIT was its effectiveness in limiting the generation of *equivalent mutants* [36], a worrying concern in mutation testing. The tests were then run against the code and its mutants, to compute the *mutation score*, which was reported to be *the most* important code coverage criterion [4], and *one of the most* relevant indicators for developers [6], [9]. The following standard definition of a mutation score was used.

Definition 1. The *mutation score* μ of a test case is the number of mutants the test killed divided by the total number of mutants.

The denominator in the definition is the number of mutants in the tested class, rather than the total number of mutants, even though a test class may kill mutants of

1. https://pitest.org/

classes other than its associated production class; therefore, $0 \leq \mu \leq 1$.

In addition to the *mutation score*, GPG collected 67 code *features*[2]. Despite the huge work on the definition and validation of the code metrics (smells included) [17]–[19], [37]–[43], it is rare to see applications in which these are used as features for machine learning.

Broadly speaking, code features can be divided into several categories. *Dynamic* features are those that are gathered by executing the code, while *static* features are those characteristics that can be gathered from the source. (*History*, mostly of maintenance, yet a third category of features, is out of the scope of this study).

Of the 67 features, only one was dynamic: the statement coverage of each test, typically obtained by (the rather slow) execution of an instrumented version of binaries. Other dynamic features mentioned in the literature but not used by GPG include, e.g., *branch coverage*, *condition coverage* and, *statement coverage* [44]. Nevertheless, Gopinath et al. [45] showed that *statement coverage* is the feature most related to test-case effectiveness.

Table 2 summarizes the remaining 66 static features. Overall, the table enumerates 41 code features. Except for the eight "test smell" and "code smell" features, all features applied to both production and test classes.

The table groups the static features into five categories. In the first category, *size*, we find five size metrics, which are easy to define and compute: Lines of Code (LOC), Number of Attributes (NOA), Number of Public Attributes (NOPA), and Number of Packages (NOP). Features in this category apply both to the test and the code part of pairs in the dataset.

Next shown are the two categories of smell: The famous *code smells* of Fowler and Beck [18] and *test smells* [19].

Following are code complexity metrics, which, for ease of reference we divided into two sub-categories: *Literature* and *Complexity*. In the "Literature" category, Table 2 presents major complexity metrics that received significant attention in the research literature, beginning from McCabe's almost half century old cyclomatic complexity [43] and Halstead's [42] metrics, through the famous Chidamber and Kemerer's [17] metrics of object-oriented design, and ending with the relatively recent Buse and Weimer [38] metric and Daka et al.'s "readability" metric.

The second sub-category of other "complexity" metrics includes less famous metrics whose description can be found in textbooks, including [39]–[41].

Variables: Dependent and independent. The features listed in Table 2, together with the statement-coverage dynamic feature, constitute the independent variables of the GPG study. The independent variable is μ or, alternatively, the mutation scores of test cases. A principal objective of the GPG work was to "*gain a deeper understanding about the factors that might affect the effectiveness of test cases*". Probably, for this reason, their main effort was not the estimate of μ but rather of the weaker notion of the *effectiveness of the test case*.

2. Called "factors" by the authors

TABLE 2: GPG's static code features (production and test)

#	Category	No.	Features
1	**Size**	4	LOC, NOA, NOPA, NOP
2	**Code smells**[1] **[18]**	8	class data should be private, complex class, blob, spaghetti code, message chain, long method, feature envy, functional decomposition
3	**Test smells**[2] **[19]**	8	assertion roulette, eager test, lazy test, mystery guest, sensitive equality, resource optimism, for testers only, indirect testing
4	**Literature**	12	Readability [37], Buse & Weimer [38], Chidamber & Kemerer [17]: 5×LCOM, WMC, CBO, RFC, Halstead [42], McCabe [43],
5	**Complexity [40]**	9	MPC, IFC, DAC, DAC2, CONNECTIVITY, COH, TCC, LCC, ICH

1 Applicable to production code only
2 Applicable to test code only

Definition 2. A test case is *effective* if its mutation score is in the upper quartile of the mutation scores of all tests in a suite. It is *ineffective* if the score is in the lower quartile.

Accordingly, all tests that were neither effective nor ineffective, i.e., falling in the two middle quartiles, were eliminated from the study, i.e., esentially halving the dataset. With these approx. 1,200 points, GPG proceeded to study the impact of each feature together with the dyadic effectiveness judgment. For this purpose, they employed d, Cliff's non-parametric measure of how each feature is correlated with effectiveness. Breaking down the range of d values to *small*, *medium*, and *large*, while ignoring the case, $|d| \leq 0.147$.

The findings were that about 40% of features are statistically insignificant predictors of effectiveness, including six of the eight code smells, and seven of the eight test smells. The best predictor was statement coverage. Other good predictors were found among size and complexity features (mostly when evaluated on production, rather than on a test). An explanation of why particular features were better predictors than others was left for a future study.

Predicting effectiveness. GPG then explored three machine learning models for predicting effectiveness: (i) k-nearest neighbors algorithm (KNN), (ii) a support vector machine (SVM), and, (iii) a Random Forest Classifier (RFC), and concluded that RFC produces the best results. The model was also able to rank factors based on their relative contribution to the prediction. Using RFC, the authors' contribution was a prediction close to 95% in terms of the F1 and area under curve measures, provided that the statement-coverage feature was included. For the "*real-case scenario*", GPG recommended a practical solution that relies on static features only, with a decrease of about 9% in the indicators, yet being highly performing.

3 Research Questions

Having described the background to our work, we now present our understanding of GPG's contribution and pose our research questions.

The dataset. GPG's dataset consists of 18 projects, a relatively small number. The projects' sizes vary broadly; the biggest project contains 442 pairs of test and production classes, while the smallest contains 11. The significant variance in sizes and the small number of projects lead to concerns about potential biases in the prediction results.

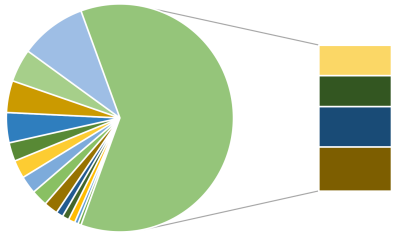

Fig. 1. The distribution of the test–production pairs among the GPG's dataset projects. Four projects make up 65% of the total number of pairs.

Fig. 1 shows the distribution of test–production pairs among the 18 GPG's dataset. We observe that four projects make up 65% of the total number of pairs. The concerns become more acute when considering the setting of the effectiveness prediction experiments, in which, as described in Sect. 2, only half the data (the highest and lowest quartiles of the mutation scores) are considered. A direct consequence is that the data from many projects are partially or entirely lost. Therefore, it is natural to ask how robust the results presented by GPG are when considering a bigger, more diverse dataset with additional projects.

> **RQ1:** How robust are the GPG results when the dataset changes?

Applicability of neural networks. As mentioned in Sect. 2, GPG considered three machine learning models in their study: the KNN, SVM, and RFC. Of these, RFC prevailed, achieving excellent prediction results at the 0.95 level while including dynamic information. Additionally, GPG recommend "*light weight*" prediction, in which dynamic information, specifically, statement coverage, is not used. This *static prediction*, however, comes at the cost of non-meager performance degradation. GPG did not explore the usage of deep neural networks (DNNs), leaving open the question of whether using neural networks would improve the results.

> **RQ2:** Can the *dynamic* and *static* predictions of test effectiveness be further improved by using DNNs, possibly, augmented with more static features?

Given the high quality already achieved in the dynamic model and that the static model is more applicable to real-world real-time applications, we are more interested in achieving improvement there.

Direct prediction of mutation score. Given an arbitrary test case, there is value even in its coarse classification as belonging to one of three categories: (i) effective, (ii) ineffective, and (iii) somewhere in between. The GPG prediction model, however, is not designed for making this judgment since it presumes that the prediction *assumes* that the given test is either effective or ineffective. In some sense, the title "*light weight prediction...*" is misleading since the prediction is contingent on whether the test is effective or ineffective. Also, the machine learning model ignores potentially valuable information found in tests in the middle two quartiles.

We may ask whether the prediction can be expanded to include all tests and to use information gathered from all tests. Even further, as GPG argued, the most important information is the mutation score. So, the natural question is whether we can predict the mutation score directly, instead of effectiveness; in the scope of this study, we plan to do this using DNNs.

> **RQ3:** How well can a DNN predict the mutation score directly, with and without statement coverage information?

Notice that existing approaches such as PMT and CBUA [29], [31] already offer a direct prediction of mutation score. Their prediction, however, was evaluated at a whole-project level. The meaning is that developers receive an estimation of the mutation score of all the tests in the project as an output. Most modern software systems follow object-oriented principles and are developed by multiple developers in a modular way. As a consequence, usually, for each production class, there is a corresponding test class. It would be useful for developers to have a live prediction of mutation score in a test-class level. With such a model, developers would be able to get a real-time estimation of the quality of their current test class and improve it on the fly. In Sect. 7 we intend to do so.

Features Selection. GPG spent much effort evaluating the quality of individual and groups of static features of the code as predictors of effectiveness. It is impossible, however, to conduct a detailed analysis of each of the many features. A somewhat surprising finding is that smell features are not as detrimental to prediction as complexity metrics.

Still, it is not clear why a particular complexity metric makes it more predictive than another. For example, why is that LCOM1-LCOM4 were relevant in the production code, whereas only LCOM5 was relevant in the test code, given that all of these are different implementations of the famous theoretical LCOM of Chidamber and Kemerer [17]. Or, why is it that larger values of the McCabe metric in the test code have a high, positive correlation with effectiveness—would we not expect simple, linear tests to be more effective than tests with complex control flow?

A deeper and more important concern is with regard to the initial selection of features, especially in the "complexity" category of Table 2. Surveying the literature [39]–[41], we find over a hundred different complexity metrics of object-oriented design and complexity. Would we not want to include them all and let the classifier prune those that are less relevant than others?

Conversely, intriguing is the failure of smells to serve as good predictors, despite the major work on their validation in the literature. In line with the same principle, one would expect the major "literature" metrics to be much more relevant than the rest of the "complexity" metrics, but the data do not seem to support this expectation. Together, these concerns raise doubt as to whether a judicious and careful selection of features and their evaluation lead to the desired

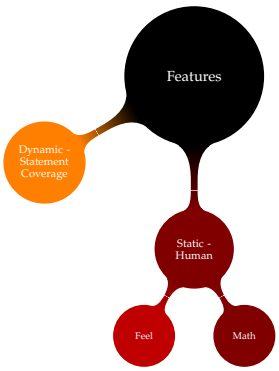

Fig. 2. GPG used statement coverage (a dynamic feature) and other human features that are based on mathematical relationships between software components, or subjective feelings.

results.

To put things in context, consider the nature of the studied features. Fig. 2 classifies the studied features into three categories. The first one, marked in orange, is of dynamic measurements, i.e., statement coverage. The other categories, marked in red, are of "human" features. The metrics in these categories are (i) mathematical relationships between basic software measurements, such as the complexity metrics, or (ii) subjective metrics, which are based on "feelings" and good practices for writing code such as test and code smells.

We argue that this set of features requires three kinds of significant engineering efforts:

1) *Innovation effort.* Developing the new features, finding their statistical relationships with other features, and empirically studying their influence on various software engineering activities.
2) *Tool-design effort.* To extract the various metrics, multiple tools are needed. Building tools for the extraction of software engineering metrics is a complex task requiring careful design and validation. Besides, in most cases, these tools are written for specific programming languages and do not provide simple migration for new programming languages and frameworks.
3) *Integration effort.* A major barrier when using multiple tools for mining features is the integration of the tools into one real-time system. A real-world system for predicting mutation score would need to incorporate multiple tools and use them on-the-fly to extract features from new code snippets.

To address these concerns, we assembled a collection of intuitive structural features that are easy to calculate and studied whether this collection is superior to GPG's features. The specific research question, answered below in Sect. 8, is then:

RQ4: How important is a judicious selection of features for the quality of effectiveness and mutation score prediction?

4 Method

This section describes the experimental methodology, including the data corpus (Sect. 4.1), features' set (Sect. 4.2), and neural networks structure (Sect. 4.3).

4.1 Dataset

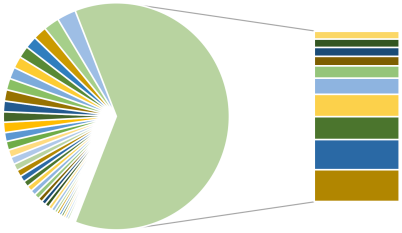

Fig. 3. The distribution of the test-production pairs among the Extended dataset with 51 projects. 10 projects make up 65% of the total number of pairs.

As a baseline, we used the same corpus of open–source JAVA projects used in [16]. Table 1 enumerates the 18 projects constituting the corpus. The corpus includes eight JAVA open-source projects that were used in past studies of mutation testing [13], [46], [47], augmented with ten other popular GitHub projects, most of which have been used in previous empirical software engineering research as well.

In view of the concerns raised in Sect. 3, we extended the baseline dataset using an additional 33 projects. We followed the same rigorous method of GPG by using parts of their replication package to identify test and production pairs, perform mutation analysis, and extract features. We made sure that the mutation score calculations were done using the same PIT version and using all the 13 mutation operators available by PIT.

We chose additional projects based on their popularity in GitHub and maintenance over the past few months. Generally, we favored projects maintained by well-known organizations such as Apache[3], Google[4], and Spring[5]. We also favored medium-sized projects with a few dozen pairs of test and production classes to prevent potential biases caused by size variance.

Table 3 enumerates the additional 33 new projects, which reflect a 60% increase in the overall number of test–production pairs. The average project size in the original GPG dataset was 134 pairs, while in the extended dataset, the average is 74. Since the influence of small projects is neglected in such large datasets, we measure the change in standard deviation (SD) only for projects with more than ten pairs; in these cases, we cut the SD of size by nearly 30% from $\sigma = 141$ to $\sigma = 100$. Fig. 3 shows the distribution of test–production pairs among the 51 projects in the extended dataset. Here, we observe more equal distribution of project sizes compared to Fig. 1.

The new corpus spans over two million lines of code, divided almost equally between production and test code.

3. https://github.com/apache
4. https://github.com/google
5. https://github.com/spring-projects

TABLE 3: Corpus of an additional 33 JAVA open-source projects: Size and other essential statistics

#	Project	#Pairs	|production|*	|test|	$\frac{\|test\|}{\|production\|}$	#Mutants	$\frac{\#Mutants}{production}$
1	commons-text	66	23,393	16,065	69%	3,888	17%
2	commons-cli	12	4,922	2,200	45%	596	12%
3	jackson-core	13	3,513	1,083	31%	701	20%
4	dagger	6	892	826	93%	1,769	198%
5	spring-ws	132	18,118	14,615	81%	13,387	74%
6	closure-stylesheets	85	20,587	16,218	79%	19,441	94%
7	commons-configuration	127	48,910	46,645	95%	3,980	8%
8	commons-dbutils	36	7,914	5,252	66%	4,102	52%
9	joda-collect	8	1,760	943	54%	3,059	174%
10	jimfs	30	7,950	7,495	94%	11,280	142%
11	exp4j	6	1,148	4,137	360%	3,924	342%
12	zxing	56	9,654	5,769	60%	41,944	434%
13	commons-validator	47	13,322	9,989	75%	15,251	114%
14	joda-time	64	30,782	26,904	87%	3,611	12%
15	commons-bcel	20	8,995	1,497	17%	15,513	172%
16	error-prone	58	7,579	6,430	85%	7,429	98%
17	commons-codec	41	18,709	13,203	71%	3,512	19%
18	joda-money	6	2,419	3,210	133%	2,333	96%
19	commons-email	14	5,171	4,254	82%	3,969	77%
20	spring-data-commons	165	37,806	28,179	74%	28,377	75%
21	commons-numbers	40	11,963	10,701	90%	42,647	356%
22	orson-charts	63	23,609	7,356	31%	65,058	276%
23	jackson-databind	29	19,491	5,222	27%	3,032	16%
24	joda-primitives	32	8,343	4,906	59%	15,005	180%
25	commons-net	34	14,386	5,823	40%	17,992	125%
26	undertow	19	8,092	2,354	29%	20,031	248%
27	raml-java-parser	3	385	452	117%	489	127%
28	spring-data-keyvalue	15	2,530	2,570	102%	1,895	75%
29	commons-fileupload	6	1,998	850	42%	175	9%
30	truth	37	11,739	14,111	120%	9,610	82%
31	highwheel	28	1,831	2,012	110%	2,570	140%
32	commons-pool	13	6,780	3,614	53%	4,144	61%
33	commons-imaging	73	12,473	4,590	37%	3,316	27%
	Total	1384	397,164	279,475	26%	374,030	40%
	Min	3	385	452	17%	175	8%
	Mean	42	12,035	8,468	80%	11,334	120%
	Median	32	8,343	5,222	75%	3,980	95%
	Max	165	4,8910	46,645	36%	65,058	43%

* All code size values are in lines of code (LOC)

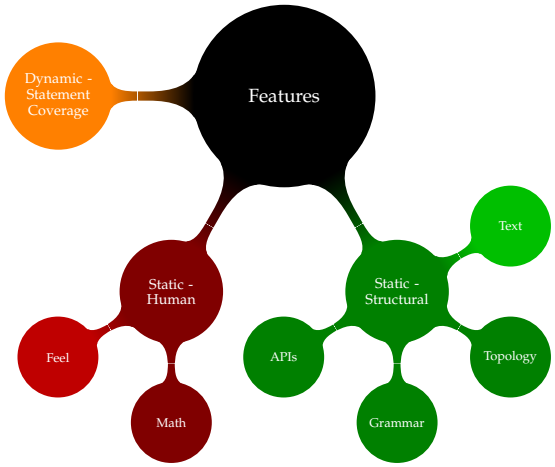

Fig. 4. Our lighter set of structural features that are more intuitive, simpler to calculate, and universal (marked in green).

Table 1 and Table 3 also draw attention to the familiar phenomena that production and test code tend to be roughly equal in size and that the number of mutants is linear in the size of the code.

4.2 A Lighter set of Features

After extending the dataset, we also created a new set of features. We added the new features carefully, using the same versions of the projects as in the original GPG dataset. For the sake of reproducibility, we ran a mining tool, which is part of the open-source Spartanizer project, used in previous studies [48], [49]. Here, we modified it slightly to meet our needs. Notice, however, that the Spartanizer project is a simple wrapping for Eclipse JDT - Abstract Syntax Tree library[6] and that any simple AST-based analysis tool or library could calculate the entire set of features.

Table 4 lists the 33 new features (also called *static code metrics*) introduced in this study, organizing these in four categories, as illustrated in Fig. 4:

1) Features in the "Text" category are computed by a simple textual analysis of the code, including the number of non-white characters and punctuation characters, and the number of identifiers.
2) The "Topology" category includes features that characterize the topology of the AST representation, including the average and maximal depth of the AST, the number of AST nodes and the degree of nodes. In this category, we find two features that characterize the tree branching level, i.e., the average squared degree of nodes and the averaged factorial of the nodes' degree.
3) The next category, "Grammar", tries to give a more inclusive and less aggregating perspective of McCabe's cyclomatic complexity. Recall that McCabe's complexity is a count of the number of branching points in the code. The set of features in this category allows a machine learning algorithm to distinguish between the different control structures and grammatical elements of the code without squeezing them into a single number.
4) The last category, "Application Programming Interfaces (APIs)", in Table 4 pertains to the testing framework's usage and is not related to the production code. In this category, we chose three popular JAVA testing frameworks. Notice that this category can be extended with additional frameworks when working on different programming languages or when using different testing libraries. Interestingly, we see a positive correlation between using mocking libraries such as Mockito and the effectiveness of tests.

The first three categories pertain both to the test code and the production code. The last two columns in the table show the magnitude of the observed differences using Cliff's d, a non-parametric effect size measure for ordinal data [50], between effective and noneffective tests. We see that except for a handful of features, the correlations have different signs, e.g., deeper ASTs in the test code are correlated with effectiveness, but deeper ASTs in the production code are anti-correlated with effectiveness (marked in red). This makes sense: we expect tests to generally be more effective if the test code is more complex. Conversely, more complex production code is less likely to have effective tests.

Also, an interesting but not surprising fact is that tests whose author failed to correctly use the advertised API of the Junit framework are less likely to be effective (i.e., they satisfied the #Misuse feature). To check where this occurred, we counted the number of occurrences where `assertTrue`

6. https://preview.tinyurl.com/y7zfvbzw

and `assertFalse` were in use for checking equality of elements or where `null` instead of more specialized assertions such as `assertEquals` and `assertNull` were used.

Generally, we see that production code features are more correlated to test effectiveness than test code features: the average absolute value of Cliff's d for test features is 0.162, and for production features, 0.171. This finding is in line with GPG's finding.

TABLE 4: Newly introduced static code features and the correlation (expressed in Cliff's d) of their values on test and production code with test effectiveness

	#	Type	Feature	Description	Cliff's d (Test)	Cliff's d (Prod)
Text	1	$\mathbb{N}$	#Characters	No. non-white space characters	+0.29	-0.30
	2	$\mathbb{N}$	#Words	No. identifiers (simple-name) used	+0.27	-0.32
	3	$\mathbb{N}$	Strings	Total length in characters of string literals	+0.08	-0.33
	4	$\mathbb{N}$	Vocabulary	No. distinct identifiers used	+0.29	-0.38
	5	$\mathbb{N}$	#Punctuation	No. non-alpha numeric characters	+0.30	-0.30
Topology	6	$\mathbb{N}$	#Nodes	No. nodes	+0.27	-0.33
	7	$\mathbb{R}^+$	Depth	Node depth (averaged)	+0.12	+0.34
	8	$\mathbb{R}^+$	Depth2	Node depth squared (averaged)	+0.38	+0.40
	9	$\mathbb{N}$	Max-depth	Maximal node depth	+0.32	-0.19
	10	$\mathbb{R}^+$	Deg2	Square of node degree (averaged)	+0.37	-0.44
	11	$\mathbb{R}^+$	Deg!	No. permutations of node children's (averaged)	+0.10	+0.03‡
Grammar	12	$\mathbb{N}$	#If	No. **`if`** statements	+0.08	-0.29
	13	$\mathbb{N}$	#Else	No. **`else`** clauses	+0.03	-0.23
	14	$\mathbb{N}$	#Loop	No. loop statements	+0.19	-0.10
	15	$\mathbb{N}$	#Break	No. **`break`** from loop statements	0.00‡	-0.02‡
	16	$\mathbb{N}$	#Continue	No. **`continue`** of loop statements	0.00‡	-0.03
	17	$\mathbb{N}$	#Try	No. **`try`** statements	+0.12	-0.08
	18	$\mathbb{N}$	#Catch	No. **`catch`** clauses	+0.03‡	-0.07
	19	$\mathbb{N}$	#&&	No. short-circuit conjunctions (**`&&`** operators)	-0.01‡	-0.15
	20	$\mathbb{N}$	#||	No. short-circuit disjunctions (**`||`** operators)	0.00‡	-0.11
	21	$\mathbb{N}$	#?:	No. conditional operators (**`?:`**)	0.00‡	-0.10
	22	$\mathbb{N}$	Comments	Length in characters of code comments	-0.13	-0.19
	23	$\mathbb{N}$	#Methods	No. methods in a class	+0.42	-0.27
	24	$\mathbb{N}$	Variety	No. distinct Java types spanned by nodes	+0.28	-0.31
	25	$\mathbb{N}$	#Expressions	No. expression nodes	+0.28	-0.33
	26	$\mathbb{N}$	#Invocations	No. method invocation nodes	+0.29	-0.32
	27	$\mathbb{N}$	#Access	No. field access nodes	+0.03	-0.20
	28	$\mathbb{N}$	#Primitive	No. primitive type nodes	+0.38	-0.25
	29	$\mathbb{N}$	#Literal	No. numeric literal nodes	+0.18	-0.22
API	30	$\mathbb{N}$	#Misuse	No. incidents of misuse of JUnit interface	-0.21	
	31	$\mathbb{B}$	Junit	Does test class use JUnit?	-0.03‡	
	32	$\mathbb{B}$	Hamcrest	Does the test class use Hamcrest?	-0.02‡	
	33	$\mathbb{B}$	Mockito	Does the test class use Mockito?	+0.25	

‡ Statistically insignificant: $p \geq 0.05$

Definition 3. The *lighter* set of 66 features consists of: features 1–29, from Table 4, measured both for test and production classes, and features 30–33 of Table 4, measured for test code only, and the four size metrics of Table 2.

Note that the lighter set of features is the same size as the light set. Only four features are common to the sets.

The lighter set ameliorates all the three kinds of engineering efforts discussed above in Sect. 3. It is based solely on intuitive and objective code structure measurements. Also, it does not require multiple tools to retrieve—only a simple analysis of the code's AST is needed. More importantly, the lighter set of features is universal and can easily be modified to fit any programming language by retrieving the new programming language AST's corresponding elements: PYTHON libraries such as AST[7] or C++ and C compiler front ends such as Clangast[8] are sufficient.

4.3 Neural Network Models

We now describe the design of the neural network models.

7. https://docs.python.org/3/library/ast.html
8. https://clang.llvm.org/docs/index.html

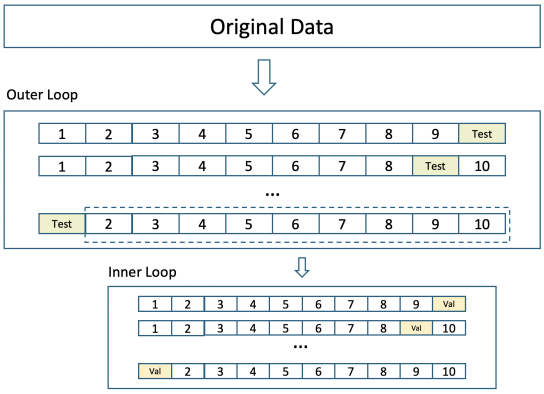

Fig. 5. 10-fold nested cross-validation.

Data normalization. The first step prior to training was data normalization [51]. In this step, features with numeric values were shifted to place their mean at zero and scaled to set their standard deviation to one.

Configuration. Model configuration has a substantial impact on the final performance of neural networks [52]; therefore, finding the best configuration should be done carefully to avoid biases.

When designing our neural network models, we considered many hyperparameters: the number of hidden layers, the number of neurons in each hidden layer, regularization techniques such as dropout, L_1 and L_2, activation and loss functions, optimizers, and termination conditions for the training. Each hyperparameter introduces a trade-off between the accuracy of the model, overfit, and training time. In practice, each hyperparameter is heavily dependent on other hyperparameters, e.g., it is wrong practice to find the best optimizer while the other hyperparameters are fixed and draw meaningful conclusions based on such assumptions. Although there are many "good practices" in the literature and online scientific blogs for fine-tuning hyperparameters [53]–[55], it is still essential to perform many training sessions that explore different combinations of hyperparameters. Notice that these massive computations are done only when designing the model and are not part of the end-user experience. With a trained model, a prediction of mutation score or effectiveness is immediate. For designing the neural networks, we used the Keras[9] PYTHON package [55], a common choice in research, and ran our experiments on a dedicated strong Linux server.

First, we filtered out hyperparameter values that lead to inferior results. To do so, we used a random grid search methodology [56], which randomly samples k hyperparameter values from the entire hyperparameter space. Studies [57] show that a random grid search can outperform a systematic grid search.

We started with 6750 potential configurations at this step, and after multiple random grid search sessions with small k values and 10-fold cross-validation [58], we filtered out hyperparameter values to arrive at a final number of 72 hyperparameter sets. We found that Adam and Adamx optimizers constantly achieved better results than other

9. https://keras.io/

optimizers we tested. Also, we found that a dropout rate between 0.1 and 0.2 in the first layer reduced overfit and that a relu activation function achieves better results than other activation functions. Also, we found that networks with 3–5 inner layers achieved better results than others. For this phase, we used the RANDOMGRIDSEARCHCV utility provided by scikit-learn[10].

Taking our 72 potential configurations, all of which led to similar high results, we proceeded to a final evaluation with a k-fold nested cross-validation strategy [59]. Nested cross-validation reduces biases by splitting the data into training, validation, and test sets in two separate loops: (i) an inner loop for fine-tuning the model hyperparameters and (ii) an outer loop to evaluate the model's performance. Fig. 5 illustrates a 10-fold nested cross-validation. A full description of the nested cross-validation approach can be found in Grano et al. [16].

We repeated these stages to produce the models for each of the results presented in this research.

Replication package. To enable replicability of our results, we will web-publish a GitHub repository of data and analysis tools, to be read in conjunction with [60].

5 THE ROBUSTNESS OF THE GPG MODEL

To assess the GPG models' robustness, we ran the same experiments conducted by GPG, with our extended dataset as described in Sect. 4.1.

Table 5 compares the original GPG results for the dynamic experiment to an identical model trained and evaluated using our extended dataset. Since test effectiveness is a dyadic judgment, the table employs the usual measures of comparison of the famous 2×2 confusion matrix: Accuracy (Acc.), Precision (Prec.), Recall (Rec.), and F1. In addition, the table employs two additional measures of error: Area Under Curve (AUC), and the Mean Absolute Error (MAE). Except as regards MAE, higher values of measures are better.

TABLE 5: Comparing the GPG RFC model with the original dataset and the extended dataset while including *dynamic* informataion

#	Dataset	Acc.	Prec.	Rec.	F1	AUC	MAE
(1)	Original	.948	.940	.960	.949	.949	.051
(2)	Extended	.946	.931	.965	.947	.946	.054
	(1)-(2)	.002	.009	-.005	-.002	.003	-.003

For the dynamic experiment, we see that, generally, the results do not change significantly when using the extended dataset. The drop in terms of all the measurements is small and there is a slight improvement of 0.5 point in Recall.

Table 6 compares the original GPG results for the static experiment to an identical model trained and evaluated using our extended dataset.

The static experiment results reflect a non-meager decrease in all the metrics, ranging between 0.5% to 4%. This significant decrease shows that the prediction model proposed by GPG is not robust to changes in the dataset when considering only static features. This finding raises concerns that GPG's excellent results were achieved thanks to the small dataset.

10. https://scikit-learn.org

TABLE 6: Comparing the GPG RFC model with the original dataset and the extended dataset while relying only on *static* information

#	Dataset	Acc.	Prec.	Rec.	F1	AUC	MAE
(1)	Original	.864	.864	.865	.864	.864	.137
(2)	Extended	.837	.822	.861	.840	.837	.163
	(1)-(2)	.027	.042	.004	.024	.027	-.026

The significant decrease in all the measurements when omitting statement coverage data is a sign of the significant correlation between dynamic coverage metrics (such as statement coverage) and mutation score. This finding confirms previous studies that found a high correlation between coverage criteria and mutation score [61]–[63] and contradicts other studies that did not find such a correlation [35].

The robustness of the dynamic model should dispel dataset-selection concerns raised by Zhang et al. [31] for supervised learning models. As demonstrated, the results were similar regardless of the dataset when we included dynamic information. However, Zhang et al. [31] concerns are in place for the static models, where we observed a significant drop in results when we changed the dataset. Considering that our dataset is significantly more diverse and balances better the influence of different projects, we argue that it is more appropriate than the GPG dataset to evaluate supervised learning algorithms for the prediction of test effectiveness and mutation score.

With these findings in mind, we used our extended dataset when conducting the rest of the experiments in Sects. 6 to 8. To validate our results, we also ran all the following experiments with the original GPG dataset and came to the same or very similar conclusions (we also observed the expected drop in results for the static models). Our replication package supports running the experiments with both datasets.

> **Answer to RQ1:** The GPG dynamic prediction model is robust against changes in the dataset. In contrast, the results for the static prediction model decrease significantly. These findings confirm previous results on the high correlation between statement coverage and mutation score.

6 PREDICTING THE EFFECTIVENESS OF TESTS

We now compare the quality of the state-of-the-art test effectiveness prediction of GPG that employed RFCs with our approach relying on DNNs.

6.1 Predicting effectiveness with dynamic information

Table 7 compares the quality of prediction *with* dynamic information (test coverage).

Row (1) in Table 7 describes the results of running the original GPG experiment with our extended dataset. It

TABLE 7: Quality of prediction of test effectiveness *with* dynamic information (test coverage) depending on (i) model: RFC *vs.* DNN, averaged over 10 independent runs, and, (ii) set of static features: light (Table 2) *vs.* full set (Table 2 and Definition 3)

#	Model	Features	Acc.	Prec.	Rec.	F1	AUC	MAE
(1)	RFC	Light	.946	.931	.965	.947	.946	.054
(2)	DNN	Light	.933	.916	.954	.934	.933	.067
		S.D.(2)	.254%	.697%	.532%	.225%	.254%	.254%
		(2)-(1)	-.013	-.015	-.011	-.013	-.013	.013
(3)	DNN	Full	.935	.920	.954	.936	.935	.065
		S.D. (3)	.21%	.66%	.79%	.23%	.21%	.21%
		(3)-(2)	0	0	0	0	0	0
		(3)-(1)	-.011	-.011	-.011	-.011	-.011	.011

functions as the baseline for our comparison. Evidently, this baseline sets the bar high, with, e.g., an F1 value of 0.946.

Row (2) in Table 7 provides the measures of prediction when using a DNN model, while using the "light" set of features, i.e., the features listed in Table 2. All values in this row were averaged over 10 independent runs of the model. Below this row, we report the *relative* standard deviation of the sample, which is less than 1%, i.e., the first two digits in the reported values of the measures are meaningful.

The next row labeled (2)-(1) compares the DNN model with the RFC model using the same set of features. As shown in the table, no statistically significant difference, i.e., greater than the standard deviation of values obtained over 10 runs, was found in either measure.

Row (3) in Table 7 shows the quality of prediction when the DNN model uses the full set of features, including both the "light" features, and the additional "lighter" static features reported in Definition 3; the row below (3) is, again, the relative standard error of samples over the 10 measurements.

Finally, the row labeled (3)-(1) checks whether the DNN model using the augmented set of features improves over the baseline. Here we see that although we receive slightly better results than (2), we still do not improve the overall results.

In conclusion, Table 7 provides a negative answer to the dynamic part of RQ2.

Answer to RQ2: (Dynamic) When statement coverage data is included, the DNN classifier, even when equipped with a richer set of features, was unable to improve the prediction of test effectiveness.

Indeed, the similarity of values in rows (1), (2) and (3) in Table 7, the high prediction values, and the findings regarding the robustness of the dynamic models lead us to conclude that the dynamic effectiveness prediction model could not be further improved.

6.2 Predicting effectiveness without dynamic information

Table 8 is structured much like Table 7, except that this time predictions of effectiveness are static, i.e., made based on information gathered statically from the code without actually running it.

TABLE 8: Quality of test effectiveness prediction: Same as Table 7, but *without* dynamic information

#	Model	Features	Acc.	Prec.	Rec.	F1	AUC	MAE
(1)	RFC	Light	.837	.822	.861	.840	.837	.163
(2)	DNN	Light	.851	.840	.867	.853	.851	.149
		S.D. (2)	.48%	.71%	.92%	.47%	.48%	.48%
		(2)-(1)	.018	.018	.006	.013	.014	-.014
(3)	DNN	Full	.878	.864	.898	.880	.878	.122
		S.D.(3)	.35%	.40%	.67%	.38%	.35%	.35%
		(3)-(2)	.027	.024	.031	.027	.027	-.027
		(3)-(1)	.041	.042	.037	.040	.041	-.041

Our results show that for the *static* models, neural networks achieve a significantly stronger performance of around 88%, i.e., an improvement of 4% over the state-of-the-art *static model*. This result is particularly important since the applications of the *static* model in real-world systems are more realistic than applications of the *dynamic* model. When excluding our new features, we observe a smaller yet significant 2% improvement over the state-of-the-art results. We see, first, that neural networks are more appropriate than classic machine learning for the task of predicting *effectiveness* of tests regardless of the additional features, and second, that including additional features in the future might improve the models even further and bring them even closer to the *dynamic* model.

Answer to RQ2: (Static) Neural network prediction models are more effective than classic learning models for static prediction of tests. Using additional (static) features improves the accuracy of the prediction model even further.

7 Predicting the Mutation Score of Tests

Unlike the RFC model whose natural product is categorical classification, the DNN model can easily be used to also produce a numerical value. We used the model described in Sect. 4 to predict μ, the mutation score of tests.

In these experiments, we used four different combinations of features, obtained by using the GPG's light set or the full set, and by including or excluding statement coverage data. The loss function used was the standard mean squared error. In contrast to the prediction of effectiveness experiment, this experiment included all 3,795 data points of pairs of test and production classes.

Table 9 presents the value of ρ, Pearson's coefficient of correlation between the obtained predicted mutation score and its actual value, a standard measurement for prediction quality assessment.

We stress that the coefficient was computed over the test portion of the data as is usual in learning algorithms. Values

TABLE 9: Pearson correlation between the actual and predicted mutation score

	Statement coverage	
Features	**Yes**	**No**
Light	$.848 \pm 0.34\%$	$.637 \pm 0.90\%$
Full	$.859 \pm 0.29\%$	$.683 \pm 0.85\%$

in the table were computed by averaging 10 independent nested cross-validation runs. The table also presents the standard error of the sample for the computed average. Evidently, the errors are relatively small, adding to the confidence in comparing and subtracting the concrete values shown in Table 9.

As expected, the best mutation score prediction, $\rho = 0.86$, is made when the set of features is full, and statement-coverage information is available. The scatter plot in Fig. 6 demonstrates the prediction quality visually.

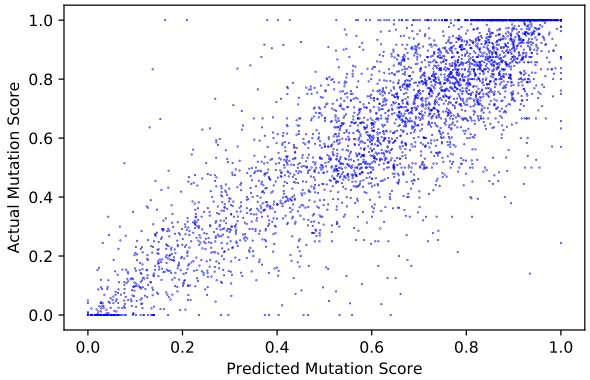

Fig. 6. Actual *vs.* prediction of mutation score with the full set of features and statement-coverage information

As seen in the figure, most data points fall close to the straight forty-five degree line and intercept zero.

Table 9 also tells us that the quality of prediction degrades, as expected, when the set of features is reduced by eliminating code features introduced in this paper (slightly degrading ρ by 0.01 to 0.85), the statement coverage information (much greater degradation of 0.18), or both (0.22 degradation). Qualitatively, we see that the major contributor to the quality of prediction is the single feature of statement coverage as in the prediction of effectiveness. A visual demonstration of this observation can be found by inspecting Fig. 7, a scatter plot of the prediction *vs.* actual μ, when this feature is missing, and comparing it to Fig. 6 above.

Table 10 gives the mean square error of the prediction depending on the dataset.

Inspecting Table 10, we can draw the same qualitative conclusions as in Table 9 regarding the major contribution that statement coverage makes to the prediction. More importantly, we note that the mean average error is less than 0.17. In other words, we can predict the mutation score with an error of *better than a quintile* for the static setting

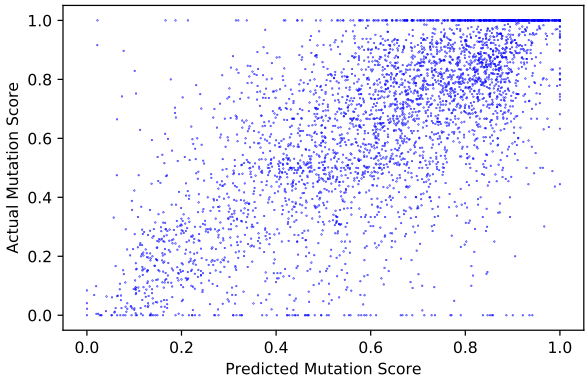

Fig. 7. Actual *vs.* prediction of mutation score with the full set of features, without statement-coverage information

TABLE 10: Mean average error of predicted mutation score

	Statement coverage	
Features	**Yes**	**No**
Light	$.112 \pm 0.81\%$	$.165 \pm 0.84\%$
Full	$.107 \pm 1.39\%$	$.156 \pm 2.42\%$

and of *about a decile* for the dynamic setting; in comparison, the prediction of effectiveness is, in essence, a prediction that tolerates an error of one or two quartiles, and for this tolerance level, the MAE values (see Table 8) are also in the range of 0.1. In other words, prediction of the mutation score instead of the effectiveness gives more detailed information (decile value), with little payment in expected error.

Our dynamic model results are competitive with the ability of PMT and CBUA [29], [31] to predict mutation score. As mentioned in Sect. 3, our prediction is made at the test class level rather than the whole-project level, providing meaningful information that existing approaches failed to provide to this date. Furthermore, we achieve our results relying only on statement coverage as a dynamic feature when the other approaches rely on various dynamic features that require multiple dedicated tools and engineering efforts to compute and require the generation of mutants.

With a slight drop in accuracy, our results for the static model are also promising since they provide a real-time prediction that does not require dynamic information.

Answer to RQ3: A DNN can effectively predict the mutation score of test cases. Our model can predict the mutation score at the granularity level of about one decile while including statement coverage, and reach better than a quintile when based solely on static features. This is a significant improvement compared to the binary classification presented by GPG. In addition, we are the first to evaluate a mutation score prediction model in a test-class level, providing developers with more meaningful real-time information.

8 Using a Lighter Set of Code Metrics

We now turn to compare the two sets of code features. In this comparison, we do not include statement coverage, which, as demonstrated, was found to be the most influential feature and to mask the influence of other features. We compare the two sets using only the DNN model since we established that it is superior to the RFC model when considering only static features.

Table 11 compares the two sets in terms of their ability to predict the effectiveness of test cases.

TABLE 11: Quality of prediction of test effectiveness with different sets of static features

#	Features	Acc.	Prec.	Rec.	F1	AUC	MAE
(1)	Full	.878	.864	.898	.880	.878	.122
(2)	Light	.851	.840	.867	.853	.851	.149
(3)	Lighter	.860	.844	.885	.863	.860	.140
	(1)-(3)	.018	.020	.013	.017	.018	-.013
	(3)-(2)	.009	.004	.018	.010	.009	-.009

Row (1) of Table 11 compares the two sets in terms of their prediction quality when using the *union* of the two sets of features, making one baseline for comparison. Row (2) is another baseline for comparison, providing prediction measures when the original GPG light features' set is used. The table compares these two rows with row (3) in which measures of quality of prediction are given for our lighter set of features, i.e., the code metrics presented in Definition 3.

The two subsequent rows of Table 11 compare the lighter set of features with the two baselines. To minimize clutter, the table does not supply information on the statistical error as was done, e.g., in Table 7. The differences between values of measures, however small, were proved to be statistically significant by a two-tailed t-test, $p < 0.01$.

As expected, we see in the row labeled (1)-(3) that the quality of prediction degrades, in terms of all measures, when reducing the set of features to the lighter set.

The difference in measures reported in row (3)-(2) show, surprisingly, that replacing the carefully selected set of features with the lighter set of features improves the prediction results. We gain a 1.0 and 1.8 point improvement in the F1 score and Recall, respectively, and nearly a 1.0 point improvement in Accuracy.

Table 12 repeats the comparison of Table 11, but this time in terms of the prediction quality of mutation score.

TABLE 12: Quality of prediction of mutation score using only the "lighter" (Definition 3) set of features using our DNN model and only static features

#	Features	MAE	ρ
(1)	Full	.156	.683
(2)	Light	.165	.637
(3)	Lighter	.161	.666
	(1)-(3)	-.005	.017
	(2)-(3)	.004	-.029

Both tables' rows are labeled identically; the measures of quality of prediction, however, are different. For the mutation score, we use, as before, MAE and the Pearson correlation.

Again, not surprisingly, we see that substituting the full set of features with the lighter set degrades quality, reducing ρ from 0.683 to 0.637 and MAE from 0.156 to 0.165. When considering only the lighter set, we observe an improvement over the light set both in MAE and ρ.

Combining the slight improvement in the results of both experiments with a significant reduction in the human engineering effort needed to deploy and streamline sophisticated tools for sniffing code and tests smells, and other object oriented programming, the results are promising. They suggest that based solely on universal, intuitive, and easy-to calculate structural metrics, it is possible to predict dynamic measures of code such as effectiveness and mutation score with accuracy which is at least as good as when using complex thoroughly-researched metrics.

Intuitively, this may be said to show that neural networks can capture some of the mathematical relationships that form the well-known metrics and smells during the training process without consuming them directly as features.

Answer to RQ4: Replacing the carefully selected GPG features with a lighter version of structural features improves the quality of the prediction of the effectiveness and the mutation score.

9 Threats to Validity

Threats to external validity. The GPG dataset, which we used in this study, was well justified and carefully analyzed by its composers. Nevertheless, as we discussed, the relatively small number of projects combined with the high variance in their sizes raised concerns. Therefore, we extended the dataset with an additional 33 new projects. We chose projects from diverse contributors and of various sizes. For comparison, in the original GPG dataset, four projects contributed over half of the data entries; we double this number in our dataset. Undoubtedly, an even larger dataset is still desirable and may influence the results.

While this is already a large-scale empirical study, extending the dataset and targeting different types of projects (for example, industrial projects) would still be advantageous and might affect the results. We also ran all the experiments using the original GPG dataset and came to

the same conclusions in all the experiments. This suggests that while the exact accuracy might change, the advantage of the DNN model over the RFC model is still valid.

Threats to construct validity. The main threat in this category is in the data extraction and analysis process. We also extended the data obtained by GPG with other features, as explained above. For mining these new features, we used the Spartanizer API, which has been used in the past for empirical software engineering studies [48], [49]. To validate the reliability of the mining, we manually sampled 10% of the test classes and 5% of the ASTs of test classes and made sure the automatic mining was accurate.

Another threat in studies involving mutation testing is equivalent and duplicated mutants. For example, previous work estimated 20% of mutants generated by PIT to be equivalent [64]; this might lead to underestimating the mutation score in the dataset and, therefore, have implications for our mutation score prediction models.

Since determining whether mutants are equivalent or not might involve considerable human effort [65], and due to the large number of mutants involved in our study (over 500,000)—conducting a manual inspection was not feasible; we maintained the presumption of GPG, which is based on Rothermel et al. [64], who considered killed mutants as non-equivalent.

That said, in this study, we tried to estimate the mutation score as calculated by PIT [33], which is the most popular mutation analysis tool for `Java`. Our model can be re-trained when the accuracy of PIT and other mutation analysis tools improves in the future. There are already preliminary efforts to improve equivalent mutants detection in PIT [66].

Threats to conclusion validity. As described, to choose the best neural network configuration, we used a nested cross-validation procedure with 10 folds both for the inner loop and the outer loop. Nested cross-validation was shown to be a sound technique for reducing biases while configuring hyperparameters [67]. For a careful comparison of our *effectiveness* prediction model to GPG, we exploited the same evaluation metrics, i.e., F1 Score, AUC, MAE, which also provide a more extensive overview of the performance of the devised model. We exploited MAE and Pearson's ρ as metrics for our mutation score estimation model and inspected the distribution graphs.

10 Discussion & Conclusion

We studied the usage of neural networks to predict the effectiveness and mutation score of tests.

The main findings of this work are:

- Dynamic prediction of test effectiveness, i.e., one that employs statement coverage, is robust against significant changes in the data-set and against changes in the prediction model.
- Neural networks improve the state of the art in predicting the effectiveness of tests when considering only static features.
- Neural networks can predict the mutation score of tests directly with good accuracy.
- The lighter set of code features is superior to the GPG set both in prediction accuracy and engineering trade-offs.

The remainder of this section discusses different aspects of our results and potential opportunities and applications for software engineering tools and research.

Efficiency vs. Accuracy. Our approach turns mutation analysis into immediate action; this, however, comes with a slight drop in accuracy. We argue that we make a good trade-off between efficiency vs. accuracy, considering the relatively small errors we achieve. Our models unveil an opportunity for projects that have never used mutation analysis and now can enjoy its many benefits without the demanding resources required by classical mutation analysis. For example, our methodology is useful for getting immediate feedback on a newly written test's quality, even prior to test execution. In particular, our approach might help determine in real-time whether a certain amount of tests are sufficient or whether additional tests are needed to achieve a better mutation score. A different use case for our approach is when a large automatically generated set of tests are to be filtered with limited execution resources [68]. In this case, the misclassification of a small number of tests is minor considering the significant performance gains. Future user studies should investigate new interaction techniques with our model and how programmers can benefit from immediate mutation analysis feedback.

Integration in tools. Our model can be integrated to test prioritization tools [69], [70] as an additional input factor. Another practical application is integration with tools such as SonarQube[11] [71] and CheckStyle[12], and integration with Continuous Integration (CI) tools [72] such as Travis CI[13]. Most of the studied features are already calculated using such tools, making the integration smoother.

We emphasize integrating with CI tools. Their capability to track changes in the code can translate into an additional dimension of features for neural networks. CI servers can update DNN models on-the-fly as a project evolves and, by this, raise the accuracy of results even more.

New metrics. Our mutation score's direct prediction results could be viewed as new stand-alone metrics for the quality of tests and not only as a replacement for a mutation score.

A significant weakness of code coverage is that it does not consider oracle information, i.e., a test might achieve 100% coverage without even validating the correctness of the program [35], [73]. In this sense, our dynamic prediction model is already superior to statement coverage since it augments many other quality indicators of the code, together with coverage, and combines them into a single metric with a runtime similar to statement coverage. Exciting future research directions include studying these new potential metrics and assessing their correlation, for example, with real-fault detection [74] and other coverage criteria, as was done for mutation analysis in the past.

Our large set of features also creates the opportunity to study new relationships between well-established metrics

11. https://www.sonarqube.org/
12. https://checkstyle.sourceforge.io/
13. https://travis-ci.com/

and code smells to structural features, and suggests even more potential test and production code quality metrics. We plan to follow this research direction using principal component analysis [75] and factor analysis [76].

Lighter set of features. Our results show that well-established code metrics and smells do not provide better prediction power than structural features obtained by a simple analysis of the code's AST. These results are encouraging for future studies involved machine learning in software engineering. Nowadays, mining smells and object-oriented metrics involves a significant engineering effort; see, for example, the considerable engineering efforts involved in a recent Java tool for detecting test smells [77]. Many programming languages lack robust analysis tools. Therefore, a universal and straightforward set of features that do not require sophisticated tools might promote studies that were not possible in the past.

A clear direction for future research is enhancing the lighter features' set, including extending it to include, e.g., averages of the degree and depth raised to higher and negative powers, other counts of node types, more refined calculation of the number of descendants, depending on the node type, etc.

It is important to point out that all the concerns and engineering efforts raised in Sect. 3 regarding the GPG features are valid to other works, such as PMT and CBUA [29], [31]. Future studies should explore whether our simple features can replace or enhance sets of features in works that focus on mutant-level prediction.

More domains. Future studies might explore whether using neural networks and our proposed lighter features set contribute to more algorithms involving machine learning for software engineering tasks, be these in the domain of testing, e.g., for predicting statement and branch coverage, or beyond, e.g., to predict proneness to change of the code or its maintainability. In initial experiments, we observed that our proposed approach improves the predictions of statement coverage. Our models could also be extended to support other programming languages, more testing frameworks, and mobile applications testing code.

REFERENCES

[1] R. G. Hamlet, "Testing programs with the aid of a compiler," *IEEE Trans. Soft. Eng.*, vol. SE-3, no. 4, pp. 279–290, 1977.

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

[3] T. A. Budd, "Mutation analysis of program test data," Ph.D. dissertation, Yale Uni., Comp. Science Dept. 51 Prospect St. New Haven, USA, 1981.

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

[5] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. L. Traon, and M. Harman, "Mutation testing advances: An analysis and survey," in *Advances Computers*, ser. Advances in Computers, A. M. Memon, Ed. Elsevier, 2019, vol. 112, pp. 275–378. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0065245818300305

[6] 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 *Proc. 22nd ACM SIGSOFT Int. Found. Soft. Eng.*, ser. FSE 2014. NY/USA: ACM, 2014, p. 654665.

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

[8] P. G. Frankl, S. N. Weiss, and C. Hu, "All-uses vs mutation testing: An experimental comparison of effectiveness," *J. Syst. & Soft.*, vol. 38, no. 3, pp. 235–253, 1997. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121296001549

[9] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *Proc. 27th Int. Conf. Soft. Eng.*, ser. ICSE 05. NY/USA: ACM, 2005, p. 402411. [Online]. Available: https://doi.org/10.1145/1062455.1062530

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

[11] B. Kurtz, P. Ammann, and J. Offutt, "Static analysis of mutant subsumption," in *2015 IEEE Int. Conf. Soft. Testing, Verification Validation Workshops (ICSTW)*, 2015, pp. 1–10.

[12] J. Strug and B. Strug, "Machine learning approach in mutation testing," in *IFIP Int. Conf. Testing Soft. Syst.* Springer, 2012, pp. 200–214.

[13] R. Just, M. D. Ernst, and G. Fraser, "Efficient mutation analysis by propagating and partitioning infected execution states," in *Proc. 2014 Int. Found. Soft. Testing Analysis*, ser. ISSTA 2014. NY/USA: ACM, 2014, p. 315326. [Online]. Available: https://doi.org/10.1145/2610384.2610388

[14] R. Gopinath, M. A. Alipour, I. Ahmed, C. Jensen, and A. Groce, "On the limits of mutation reduction strategies," in *Proc. 38th Int. Conf. Soft. Eng.*, ser. ICSE '16. NY/USA: ACM, 2016, p. 511522. [Online]. Available: https://doi.org/10.1145/2884781.2884787

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

[16] G. Grano, F. Palomba, and H. C. Gall, "Lightweight assessment of test-case effectiveness using source-code-quality indicators," *IEEE Trans. Soft. Eng.*, 2019.

[17] S. R. Chidamber and C. F. Kemerer, "A metrics suite for OO design," *IEEE Trans. Soft. Eng.*, vol. 20, no. 6, pp. 476–493, 1994.

[18] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley, 2018.

[19] A. Van Deursen, L. Moonen, A. Van Den Bergh, and G. Kok, "Refactoring test code," in *Proc. 2nd Int. Conf. extreme Prog. flexible processes Soft. Eng. (XP)*, 2001, pp. 92–95.

[20] D. Zhang and J. J. Tsai, "Machine learning and software engineering," *Soft. Quality J.*, vol. 11, no. 2, pp. 87–119, 2003.

[21] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *2019 IEEE/ACM 41st Int. Conf. Soft. Eng. (ICSE)*, 2019, pp. 783–794.

[22] L. Wei, Y. Liu, and S. Cheung, "PIVOT: learning API-device correlations to facilitate android compatibility issue detection," in *2019 IEEE/ACM 41st Int. Conf. Soft. Eng. (ICSE)*, 2019, pp. 878–888.

[23] M. Fan, X. Luo, J. Liu, M. Wang, C. Nong, Q. Zheng, and T. Liu, "Graph embedding based familial analysis of android malware using unsupervised learning," in *2019 IEEE/ACM 41st Int. Conf. Soft. Eng. (ICSE)*, 2019, pp. 771–782.

[24] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley, "Automated vulnerability detection in source code using deep representation learning," in *2018 17th IEEE Int. Conf. Machine Learning App. (ICMLA)*, 2018, pp. 757–762.

[25] A. Zohar and L. Wolf, "Automatic program synthesis of long programs with a learned garbage collector," in *Advances Neural Information Processing Syst. 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds. Curran Associates,, 2018, pp. 2094–2103.

[26] K. Liu, D. Kim, T. F. Bissyandé, T. Kim, K. Kim, A. Koyuncu, S. Kim, and Y. Le Traon, "Learning to spot and refactor inconsistent method names," in *2019 IEEE/ACM 41st Int. Conf. Soft. Eng. (ICSE)*, 2019, pp. 1–12.

[27] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshyvanyk, "On learning meaningful code changes via neural machine translation," in *2019 IEEE/ACM 41st Int. Conf. Soft. Eng. (ICSE)*, 2019, pp. 25–36.

[28] G. Grano, T. V. Titov, S. Panichella, and H. C. Gall, "How high will it be? using machine learning models to predict branch coverage in automated testing," in *2018 IEEE Work. Machine Learning Techniques for Soft. Quality Evaluation (MaLTeSQuE)*, 2018, pp. 19–24.

[29] J. Zhang, L. Zhang, M. Harman, D. Hao, Y. Jia, and L. Zhang, "Predictive mutation testing," *IEEE Trans. Soft. Eng.*, vol. 45, no. 9, pp. 898–918, 2019.

[30] D. Mao, L. Chen, and L. Zhang, "An extensive study on cross-project predictive mutation testing," in *2019 12th IEEE Conf. Soft. Testing, Validation Verification (ICST)*, 2019, pp. 160–171.

[31] P. Zhang, Y. Li, W. Ma, Y. Yang, L. Chen, H. Lu, Y. Zhou, and B. Xu, "Cbua: A probabilistic, predictive, and practical approach for evaluating test suite effectiveness," *IEEE Transactions on Software Engineering*, pp. 1–1, 2020.

[32] K. Jalbert and J. S. Bradbury, "Predicting mutation score using source code and test suite metrics," in *2012 1st Int. Work. Realizing AI Synergies Soft. Eng. (RAISE)*, 2012, pp. 42–46.

[33] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "PIT: A practical mutation testing tool for Java (demo)," in *Proc. 25th Int. Found. Soft. Testing Analysis*, ser. ISSTA 2016. NY/USA: ACM, 2016, pp. 449–452. [Online]. Available: https://doi.org/10.1145/2931037.2948707

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

[35] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in *Proc. 36th Int. Conf. Soft. Eng.*, ser. ICSE 2014. NY/USA: ACM, 2014, p. 435445. [Online]. Available: https://doi.org/10.1145/2568225.2568271

[36] L. Fernandes, M. Ribeiro, L. Carvalho, R. Gheyi, M. Mongiovi, A. Santos, A. Cavalcanti, F. Ferrari, and J. C. Maldonado, "Avoiding useless mutants," *SIGPLAN Not.*, vol. 52, no. 12, p. 187198, Dec. 2017. [Online]. Available: https://doi.org/10.1145/3170492.3136053

[37] E. Daka, J. Campos, G. Fraser, J. Dorn, and W. Weimer, "Modeling readability to improve unit tests," in *Proc. 2015 10th Joint Meeting Soft. Eng.*, ser. ESEC/FSE 2015. NY/USA: ACM, 2015, p. 107118. [Online]. Available: https://doi.org/10.1145/2786805.2786838

[38] R. P. L. Buse and W. R. Weimer, "Learning a metric for code readability," *IEEE Trans. Soft. Eng.*, vol. 36, no. 4, pp. 546–558, 2010.

[39] S. Ducasse and S. Demeyer, *The FAMOOS OO Reengineering Handbook*. Uni. Bern, 1999.

[40] B. Henderson-Sellers, *OO Metrics: Measures of Complexity*. USA: Prentice, 1995.

[41] M. Lorenz and J. Kidd, *OO Software. Metrics: A Practical Guide*. USA: Prentice, 1994.

[42] M. H. Halstead, *Elements of Software Science*. NY/USA: Elsevier, 1977.

[43] T. J. McCabe, "A complexity measure," *IEEE Trans. Soft. Eng.*, no. 4, pp. 308–320, 1976.

[44] J. C. Miller and C. J. Maloney, "Systematic mistake analysis of digital computer programs," *Communications ACM*, vol. 6, no. 2, pp. 58–63, 1963.

[45] R. Gopinath, C. Jensen, and A. Groce, "Code coverage for suite evaluation by developers," in *Proc. 36th Int. Conf. Soft. Eng.*, ser. ICSE 2014. NY/USA: ACM, 2014, p. 7282. [Online]. Available: https://doi.org/10.1145/2568225.2568278

[46] S. Romano and G. Scanniello, "Smug: a selective mutant generator tool," in *2017 IEEE/ACM 39th Int. Conf. Soft. Eng. Companion (ICSE-C)*, 2017, pp. 19–22.

[47] R. Just, "The major mutation framework: efficient and scalable mutation analysis for java," in *Proc. 2014 Int. Found. Soft. Testing Analysis*, ser. ISSTA 2014. NY/USA: ACM, 2014, p. 433436. [Online]. Available: https://doi.org/10.1145/2610384.2628053

[48] D. D. Maayan, "The quality JUnit tests: An empirical study report," in *Proc. 1st Int. Work. Soft. Qualities Their Dependencies*, ser. SQUADE 18. NY/USA: ACM, 2018, p. 3336. [Online]. Available: https://doi.org/10.1145/3194095.3194102

[49] Y. Gil, O. Marcovitch, and M. Orrú, "A nano-pattern language for Java," *J. Comp. Lang.*, vol. 54, p. 100905, 2019.

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

[51] M. A. Hall, "Correlation-based feature selection for machine learning," Ph.D. dissertation, Uni. Waikato Hamilton, 1999.

[52] U. Anders and O. Korn, "Model selection in neural networks," *Neural networks*, vol. 12, no. 2, pp. 309–323, 1999.

[53] M. Rafiq, G. Bugmann, and D. Easterbrook, "Neural network design for engineering applications," *Computers & Structures*, vol. 79, no. 17, pp. 1541–1552, 2001. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0045794901000396

[54] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep learning*. MIT press Cambridge, 2016, vol. 1, no. 2.

[55] A. Gulli and S. Pal, *Deep Learning with Keras*. Packt Publishing, 2017.

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

[57] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *J. Mach. Learn. Res.*, vol. 13, p. 281305, Feb. 2012.

[58] F. Mosteller and J. W. Tukey, "Data analysis, including statistics," *Handbook social psychology*, vol. 2, pp. 80–203, 1968.

[59] M. Stone, "Cross-validatory choice and assessment of statistical predictions," *J. Royal Statistical Society: Series B (Methodological)*, vol. 36, no. 2, pp. 111–133, 1974.

[60] G. Grano, F. Palomba, and H. C. Gall, "Lightweight assessment of test-case effectiveness using source-code quality indicators—replication package," online, Feb. 2019.

[61] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, "Using mutation analysis for assessing and comparing testing coverage criteria," *IEEE Trans. Soft. Eng.*, vol. 32, no. 8, pp. 608–624, 2006.

[62] P. G. Frankl and S. N. Weiss, "An experimental comparison of the effectiveness of the all-uses and all-edges adequacy criteria," in *Proc. Symp. Testing, analysis, & verification*, 1991, pp. 154–164.

[63] P. G. Frankl and O. Iakounenko, "Further empirical studies of test effectiveness," *SIGSOFT Soft. Eng. Notes*, vol. 23, no. 6, p. 153162, Nov. 1998. [Online]. Available: https://doi.org/10.1145/291252.288298

[64] G. Rothermel, R. H. Untch, Chengyun Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *IEEE Trans. Soft. Eng.*, vol. 27, no. 10, pp. 929–948, 2001.

[65] A. J. Offutt and Jie Pan, "Detecting equivalent mutants and the feasible path problem," in *Proc. 11th Ann. Conf. Comp. Assurance. COMPASS '96*, 1996, pp. 224–236.

[66] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "Learning how to mutate source code from bug-fixes," in *2019 IEEE Int. Conf. Soft. Maint. Evolution (ICSME)*, 2019, pp. 301–312.

[67] D. Krstajic, L. J. Buturovic, D. E. Leahy, and S. Thomas, "Cross-validation pitfalls when selecting and assessing regression and classification models," *J. cheminformatics*, vol. 6, no. 1, pp. 1–15, 2014.

[68] S. Nerur, R. Mahapatra, and G. Mangalaraj, "Challenges of migrating to agile methodologies," *Commun. ACM*, vol. 48, no. 5, p. 7278, May 2005. [Online]. Available: https://doi.org/10.1145/1060710.1060712

[69] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Comput. Surv.*, vol. 43, no. 2, Feb. 2011. [Online]. Available: https://doi.org/10.1145/1883612.1883618

[70] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Soft. testing, verification & reliability*, vol. 22, no. 2, pp. 67–120, 2012.

[71] G. Campbell and P. P. Papapetrou, *SonarQube in action*. Manning, 2013.

[72] K. Beck, "Embracing change with extreme programming," *Comp.*, vol. 32, no. 10, pp. 70–77, 1999.

[73] Y. Zhang and A. Mesbah, "Assertions are strongly correlated with test suite effectiveness," in *Proc. 2015 10th Joint Meeting Foundations Soft. Eng.*, ser. ESEC/FSE 2015. NY/USA: ACM, 2015, p. 214224. [Online]. Available: https://doi.org/10.1145/2786805.2786858

[74] D. B. Brown, M. Vaughn, B. Liblit, and T. Reps, "The care and feeding of wild-caught mutants," in *Proc. 2017 11th Joint Meeting Foundations Soft. Eng.*, ser. ESEC/FSE 2017. NY/USA: ACM, 2017, p. 511522. [Online]. Available: https://doi.org/10.1145/3106237.3106280

[75] H. Hotelling, "Analysis of a complex of statistical variables into principal components." *J. educational psychology*, vol. 24, no. 6, p. 417, 1933.

[76] S. A. Mulaik, *Foundations of factor analysis*. CRC press, 2009.

[77] A. Peruma, K. Almalki, C. D. Newman, M. W. Mkaouer, A. Ouni, and F. Palomba, "Tsdetect: An open source test smells detection tool," in *Proc. 28th ACM Joint Meeting Europ. Soft. Eng. Conf. & Symp. Foundations Soft. Eng.*, ser. ESEC/FSE 2020. NY/USA: ACM, 2020, p. 16501654. [Online]. Available: https://doi.org/10.1145/3368089.3417921