Better Prediction of Mutation Score

Dor David Ma’ayan
Better Prediction of Mutation Score

Research Thesis

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

Dor David Ma’ayan

Submitted to the Senate of the Technion — Israel Institute of Technology
Shevat 5781 Haifa January 2021
This research was carried out under the supervision of Prof. Yossi Gil, in the Faculty of Computer Science.

Some results in this thesis have been published as articles by the author and research collaborators in conferences and journals during the course of the author’s research period, the most up-to-date versions of which being:


More results are under review for *IEEE Transaction on Software Engineering*.

**Acknowledgements**

I am lucky to work with so many talented people. These people make me a better person and researcher, and I would like to thank everyone who takes part in my journey through academia and life.

First, I would like to thank my advisor, Professor Yossi Gil, for all his enthusiasm and support. The professional and intense guidance of Yossi pushed me forward, helped me to make the impossible possible and to achieve goals I never dreamed of.

During my studies, I was a visiting researcher at the Penrose research group at Carnegie Mellon University, hosted by Joshua Sunshine. Josh, thank you for showing me how to do top-notch academic research, for opening your home for me, and for introducing me to so many great people in Pittsburgh’s lovely Jewish community. I would also like to thank other faculty members at CMU, especially Jonathan Aldrich, Keenan Crane, and Chinmay Kulkarni. My CMU adventure was not the same without the gifted students I had the privilege to work with, especially Wode Ni, Katherine Ye, and Jenna Wise.

I want to thank Ori Roth for endless hours of fascinating discussions on research and academia. This intense period wouldn’t have been the same without you.

Last but not least, I would like to thank my dear family, my parents, and my sister. Without your tremendous support and love, nothing could happen!

The Technion’s funding of this research is hereby acknowledged.
# Contents

List of Figures

List of Tables

Abstract

Abbreviations and Notations

1 Introduction

2 Background

  2.1 Mutation Analysis

  2.2 Using Machine Learning for Mutation Analysis

  2.3 The Work of Grano et al.

3 Research Questions

4 Method

  4.1 Dataset

  4.2 A Lighter set of Features

  4.3 Neural Network Models

  4.4 Measurements

5 Results

  5.1 The Robustness of the GPG Model

  5.2 Predicting the Effectiveness of Tests

    5.2.1 Predicting effectiveness with dynamic information

    5.2.2 Predicting effectiveness without dynamic information

  5.3 Predicting the Mutation Score of Tests

  5.4 Using a lighter set of code metrics

6 Threats to validity

7 Discussion and Conclusion
List of Figures

2.1 Source Code ................................................................. 7
2.2 Test Suite ................................................................. 7
2.3 Negate Conditionals Mutator ........................................ 8
2.4 Math Mutator ............................................................. 8
2.5 Invert Negatives Mutator ............................................. 8

3.1 The distribution of the test–production pairs among the GPG’s dataset projects, each color represents a single project. Four projects (in the rectangle) make up 65% of the total number of pairs (in the circle). . . 15
3.2 GPG used statement coverage (a dynamic feature) and other human features that are based on mathematical relationships between software components, or subjective feelings. ................................. 18

4.1 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. 19
4.2 Our lighter set of structural features that are more intuitive, simpler to calculate, and universal (marked in green). ................................. 20
4.3 10-fold nested cross-validation. ................................. 25
4.4 2 × 2 confusion matrix ............................................. 26

5.1 Actual vs. prediction of mutation score with the full set of features and statement-coverage information ................................. 31
5.2 Actual vs. prediction of mutation score with the full set of features, without statement-coverage information ................................. 32
List of Tables

2.1 Corpus of 18 JAVA open-source projects (Grano et al. 2019a): Size and other essential statistics .................................................. 11
2.2 GPG’s static code features (production and test) ........................... 12
4.1 Corpus of additional 33 Java open source projects: size and other essential statistics ................................................................. 21
4.2 Newly introduced static code features and the correlation (expressed in Cliff’s d) of their values on test and production code with test effectiveness 23
5.1 Comparing the GPG RFC model with the original dataset and the extended dataset while including dynamic information .................. 27
5.2 Comparing the GPG RFC model with the original dataset and the extended dataset while relying only on static information ............... 27
5.3 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 2.2) vs. full set (Table 2.2 and Definition 4.2.1) .................................................. 29
5.4 Quality of test effectiveness prediction: Same as Table 5.3, but without dynamic information ........................................................... 30
5.5 Pearson correlation between the actual and predicted mutation score .. 31
5.6 Mean average error of predicted mutation score .............................. 32
5.7 Quality of prediction of test effectiveness with different sets of static features ........................................................................ 34
5.8 Quality of prediction of mutation score using only the “lighter” (Definition 4.2.1) set of features using our DNN model and only static features 34
A.1 Quality of prediction of test effectiveness with different test or production static features ............................................................ 43
A.2 Quality of prediction of mutation score using only test or production set of features using DNN only static feature .......................... 44
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. The research community invested significant efforts to reduce these costs; however, no practical and real-time solutions were proposed so far.

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 gain a 4% improvement in the state-of-the-art results for binary test effectiveness prediction. Intending to propose a practical approach, we introduce an intuitive, easy-to-calculate set of features based on a simple analysis of the Code’s Abstract Syntax Tree (AST). Our new set of features is superior to previously studied sets of features in its prediction performance, and it also significantly reduces the engineering trade-offs involved in the extraction process.

We publish the largest dataset of test-class level mutation score and static code features to date, for future research. Finally, we discuss how our approach could be integrated into real-world systems, IDEs, Continuous Integration (CI) tools, and testing frameworks and the implication of our findings for future studies in the field.
Abbreviations and Notations

Acc. Accuracy

API Application Programming Interface

AST Abstract Syntax Tree

AUC Area Under Curve

CI Continuous Integration

DNN Deep Neural Network

GPG the work and contribution of Grano, Palomba, and, Gall (Grano et al. 2019a)

KNN $k$-Nearest Neighbors algorithm

LOC Lines of Code

MAE Mean Absolute Error

NOA Number of Attributes

NOP Number of Packages

NOPA Number of Public Attributes

Prec. Precision

Rec. Recall

RFC Random Forest Classifier

SVM Support Vector Machine

TDD Test-Driven Development
Chapter 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 (Hamlet 1977; DeMillo et al. 1978) 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., (Budd 1981) for an accessible introduction, or (Jia and Harman 2011; Papadakis et al. 2019) 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 (Just et al. 2014b). 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 (Mathur and Wong 1994; Frankl et al. 1997; Just et al. 2014b; Andrews et al. 2005).

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 2.1 and 4.1 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 2.1 and 4.1 as well), the required resources are likely to be quadratic (Budd 1981). This limitation prevents mutation analysis from being widely used outside of academic research.

Offutt and Untch (Offutt and Untch 2001) 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 (Kurtz et al. 2015; Strug and Strug 2012; Just et al. 2014a). These approaches present some promising results. Nevertheless, recent studies by Gopinath et al. (2016; 2017) 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. (2019a) suggested a drastically different approach. Using classic machine learning algorithms with a light set of source code quality metrics (Chidamber and Kemerer 1994), and smells (Fowler 2018; Van Deursen et al. 2001), 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 (Grano et al. 2019a) 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. Chapter 2 provides background about mutation analysis and a short review of the results obtained by Grano et al. (2019a). Taking these into consideration, we phrase specific research questions in Chapter 3. Chapter 4 describes our experimental setting. Answers to the questions from Chapter 3 are then presented in Chapter 5. Chapter 6 discusses threats to the validity of the results and Chapter 7 discusses our results and concludes.
Chapter 2

Background

2.1 Mutation Analysis

The research community studies various metrics for measuring the quality of software tests. A prominent family of metrics for that purpose are coverage criteria such as statement or line coverage. These metrics validate that software tests run through every component of the source code.

However, there is a subtle difference between measuring that tests run through every component of the source code and measuring that tests verify the correctness of every element of it. For example, traditional test coverage measurements can easily be fooled by tests that run through every code component. The goal of mutation analysis (Budd 1981) is to provide a coverage criterion that takes correctness verification into account.

For an intuitive example of mutation analysis, consider the simple Java code in Fig. 2.1 which computes Fibonacci’s n\textsuperscript{th} element, and assume the test suite in Fig. 2.2 validates it. How can we know that the test suite validates the correctness of the code thoroughly?

In mutation analysis, a set of mutants of the source-code is generated automatically. Mutants are small faulty variants of the code that are supposed to mimic potential unintentional bugs that do not break the build but impair the program’s correctness. Figs. 2.3 to 2.5 show simple examples of mutants for the source code from Fig. 2.1. Most of the generated mutants are tiny logical or arithmetic changes of the code. For example, in Fig. 2.4 the math mutator changes a single - operator into +.
```
static int fib(int n) {
    if (n > 1)
        return n;
    return fib(n-1) + fib(n-2);
}
```

Figure (2.3) Negate Conditionals Mutator

```
static int fib(int n) {
    if (n <= 1)
        return n;
    return fib(n+1) + fib(n-2);
}
```

Figure (2.4) Math Mutator

```
static int fib(int n) {
    if (n <= 1)
        return -n;
    return fib(n-1) + fib(n-2);
}
```

Figure (2.5) Invert Negatives Mutator

Against each of the generated mutants, a mutation analyzer executes the entire test suite. In case of failure, we call the mutant a killed mutant. Killed mutants are a positive sign that the test suite is able to detect potential bugs of the production code and that the tests validate the correctness of the code in addition to its coverage.

After running all the mutants, the analyzer calculates the mutation score of a test suite as defined at Definition 2.1.1.

Definition 2.1.1. The mutation score $\mu$ 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 classes other than its associated production class; therefore, $0 \leq \mu \leq 1$.

Studies show that mutation analysis provides developers with a better and more reliable measurement of test effectiveness with respect to other criteria (Just et al. 2014b; Andrews et al. 2006).

Despite its many benefit, mutation analysis still suffers from one major limitation: it is an extremely costly activity which requires (i) generation of mutants, (ii) their compilation, and (iii) the execution of the test suite against each of them. This process becomes more expensive with a bigger size of code and a higher frequency of commits (Andrews et al. 2005).
Research has been conducted in the last years to reduce the computational costs of mutation analysis. Offutt and Untch (2001) grouped the efforts in the field into three categories (i) do fewer, (ii) do smarter, and (iii) do faster. The first category (do fewer) has been the most investigated one so far. Kurtz et al. (2015) used subsumptions graphs to reduce the number of considered mutants. Another approach, by Strug and Strug (2012) used machine learning algorithms on hierarchical graphs representation of mutants to detect similar mutants. Just et al. (2014a) used run-time information to detect mutants that lead the code to the same state and considered them identical. A study by Gopinath et al. (2017) empirically showed that mutation reduction techniques do not perform better than a simple random sampling of mutants while considering the tiny run time improvement they offer and the significant overhead in the methods themselves. The main bottleneck of all the described approaches is that the inherent problem of generating and compiling mutants and running the entire test suite against each of them is still in place. Our work eliminates this bottleneck by estimating the mutation score of tests without generating mutants.

2.2 Using Machine Learning for Mutation Analysis

The use of machine learning in software engineering (Zhang and Tsai 2003) is increasing at a staggering rate. Applications include generating code summaries (Zhang et al. 2019a), learning APIs (Wei et al. 2019), detection of malware (Fan et al. 2019) and vulnerabilities (Russell et al. 2018), program synthesis (Zohar and Wolf 2018), automatically fixing method names (Liu et al. 2019), identifying meaningful code changes (Tufano et al. 2019a), 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 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 (Grano et al. 2018) and mutation score.

Zhang et al. (2019b) 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. (2019) 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. (2020) presented an unsupervised learning approach for predicting test effectiveness, which consumes as an input a test suite, 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 suite 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. (2012) 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. (Grano et al. 2019a) continued these efforts, as described in the following subsection.

2.3 The Work of Grano et al.

Here, we review the work and contribution of Grano, Palomba, and, Gall (Grano et al. 2019a) (GPG). In Chapter 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 2.1). 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.

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\(^1\) (Coles et al. 2016)—arguably, the most robust publicly available mutation testing analysis tool (Delahaye and Du Bousquet 2015), and the tool of choice in several recent publications, e.g., (Just et al. 2014a; Inozemtseva and Holmes 2014). Another reason for the authors selecting PIT was its effectiveness in limiting the generation of equivalent mutants (Fernandes et al. 2017), 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 (Jia and Harman 2011), and one of the most relevant indicators for developers (Just et al. 2014b; Andrews et al. 2005).

\(^1\)https://pitest.org/
Table (2.1) Corpus of 18 JAVA open-source projects (Grano et al. 2019a): Size and other essential statistics

<table>
<thead>
<tr>
<th># Project</th>
<th>#Pairs</th>
<th>production</th>
<th>test</th>
<th>test production</th>
<th>#Mutants production</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 RxJava</td>
<td>442</td>
<td>109,978</td>
<td>159,044</td>
<td>145%</td>
<td>21,181</td>
</tr>
<tr>
<td>2 cat</td>
<td>62</td>
<td>11,918</td>
<td>5,052</td>
<td>42%</td>
<td>9,850</td>
</tr>
<tr>
<td>3 checkstyle</td>
<td>228</td>
<td>61,931</td>
<td>46,995</td>
<td>76%</td>
<td>64,330</td>
</tr>
<tr>
<td>4 closure-compiler</td>
<td>308</td>
<td>140,264</td>
<td>165,600</td>
<td>118%</td>
<td>95,742</td>
</tr>
<tr>
<td>5 commons-beanutils</td>
<td>56</td>
<td>15,293</td>
<td>20,465</td>
<td>134%</td>
<td>5,542</td>
</tr>
<tr>
<td>6 commons-collections</td>
<td>103</td>
<td>27,950</td>
<td>23,344</td>
<td>84%</td>
<td>9,957</td>
</tr>
<tr>
<td>7 commons-io</td>
<td>61</td>
<td>11,397</td>
<td>9,088</td>
<td>80%</td>
<td>4,315</td>
</tr>
<tr>
<td>8 commons-lang</td>
<td>109</td>
<td>75,160</td>
<td>52,610</td>
<td>70%</td>
<td>39,975</td>
</tr>
<tr>
<td>9 commons-math</td>
<td>409</td>
<td>133,248</td>
<td>95,589</td>
<td>72%</td>
<td>88,865</td>
</tr>
<tr>
<td>10 fastjson</td>
<td>64</td>
<td>30,107</td>
<td>6,376</td>
<td>21%</td>
<td>36,903</td>
</tr>
<tr>
<td>11 gson</td>
<td>23</td>
<td>8,691</td>
<td>4,979</td>
<td>57%</td>
<td>6,347</td>
</tr>
<tr>
<td>12 guice</td>
<td>24</td>
<td>6,641</td>
<td>10,685</td>
<td>161%</td>
<td>2,649</td>
</tr>
<tr>
<td>13 javapoet</td>
<td>12</td>
<td>3,589</td>
<td>4,938</td>
<td>138%</td>
<td>2,789</td>
</tr>
<tr>
<td>14 jfreechart</td>
<td>315</td>
<td>165,631</td>
<td>67,185</td>
<td>41%</td>
<td>86,912</td>
</tr>
<tr>
<td>15 joda-beans</td>
<td>11</td>
<td>3,939</td>
<td>2,712</td>
<td>69%</td>
<td>3,038</td>
</tr>
<tr>
<td>16 jsoup</td>
<td>23</td>
<td>9,872</td>
<td>5,861</td>
<td>59%</td>
<td>7,974</td>
</tr>
<tr>
<td>17 junit4</td>
<td>48</td>
<td>6,898</td>
<td>5,599</td>
<td>81%</td>
<td>3,066</td>
</tr>
<tr>
<td>18 opengrok</td>
<td>113</td>
<td>36,342</td>
<td>20,912</td>
<td>58%</td>
<td>25,049</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>2,411</td>
<td>858,849</td>
<td>707,034</td>
<td>82%</td>
<td>514,484</td>
</tr>
</tbody>
</table>

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

In addition to the mutation score, GPG collected 67 code features\(^2\). Despite the huge work on the definition and validation of code metrics (smells included) (Daka et al. 2015; Buse and Weimer 2010; Van Deursen et al. 2001; Fowler 2018; Ducasse and Demeyer 1999; Henderson-Sellers 1995; Chidamber and Kemerer 1994; Lorenz and Kidd 1994; Halstead 1977; McCabe 1976), 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 (Miller and Maloney 1963). Nevertheless, Gopinath et al. (Gopinath et al. 2014) showed that statement coverage is the feature most related to test-case effectiveness.

Table 2.2 summarizes the remaining 66 static features. Overall, the table enumerates 41 code features. Except for the eight “test smell” and “code smell” features, all

\(^2\)Called “factors” by the authors
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 (Fowler 2018) and test smells (Van Deursen et al. 2001).

Following are code complexity metrics, which, for ease of reference we divided into two sub-categories: Literature and Complexity. In the “Literature” category, Table 2.2 presents major complexity metrics that received significant attention in the research literature, beginning from McCabe’s almost half century old cyclomatic complexity (McCabe 1976) and Halstead’s (1977) metrics, through the famous Chidamber and Kemerer’s (1994) metrics of object-oriented design, and ending with the relatively recent Buse and Weimer (Buse and Weimer 2010) 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 (Lorenz and Kidd 1994; Henderson-Sellers 1995; Ducasse and Demeyer 1999).

<table>
<thead>
<tr>
<th>#</th>
<th>Category</th>
<th>No. Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Size</td>
<td>4 LOC, NOA, NOPA, NOP</td>
</tr>
<tr>
<td>2</td>
<td>Code smells¹ (Fowler 2018)</td>
<td>8 class data should be private, complex class, blob, spaghetti code, message chain, long method, feature envy, functional decomposition</td>
</tr>
<tr>
<td>3</td>
<td>Test smells² (Van Deursen et al. 2001)</td>
<td>8 assertion roulette, eager test, lazy test, mystery guest, sensitive equality, resource optimism, for testers only, indirect testing</td>
</tr>
<tr>
<td>5</td>
<td>Complexity (Henderson-Sellers 1995)</td>
<td>9 MPC, IFC, DAC, DAC2, CONNECTIVITY, COH, TCC, LCC, ICH</td>
</tr>
</tbody>
</table>

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

Variables: Dependent and independent  The features listed in Table 2.2, together with the statement-coverage dynamic feature, constitute the independent variables of the GPG study. The dependent variable is \( \mu \) 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 \( \mu \) but rather of the weaker notion of the effectiveness of the test case.
Definition 2.3.1. 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., essentially 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, $|d| \leq 0.147$, which represents insignificant.

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 (Baeza-Yates et al. 1999), 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 performant.
Chapter 3

Research Questions

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

Figure (3.1) The distribution of the test–production pairs among the GPG’s dataset projects, each color represents a single project. Four projects (in the rectangle) make up 65% of the total number of pairs (in the circle).

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 has led to concerns about potential biases in the prediction results. Fig. 3.1 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. 2.3, 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. 2.3, GPG considered three machine learning models in their study: 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 (Zhang et al. 2019b,
2020) 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. 5.3 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 is 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 (Chidamber and Kemerer 1994). 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 2.2. Surveying the literature (Lorenz and Kidd 1994; Henderson-Sellers 1995; Ducasse and Demeyer 1999), 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 results.

To put things in context, consider the nature of the studied features. Fig. 3.2 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.
GPG used statement coverage (a dynamic feature) and other human features that are based on mathematical relationships between software components, or subjective feelings.

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. 5.4, is then:

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

Figure (4.1) 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 (Grano et al. 2019a). Table 2.1 enumerates the 18 projects constituting the corpus. The corpus includes eight Java open-source projects that were used in past studies of mutation testing (Just et al. 2014a; Romano and Scanniello 2017; Just 2014), 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 Chapter 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

19
Figure (4.2) Our lighter set of structural features that are more intuitive, simpler to calculate, and universal (marked in green).

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\(^1\), Google\(^2\), and Spring\(^3\). 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 4.1 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. 4.1 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. 3.1.

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

Table 2.1 and Table 4.1 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.

---

\(^1\)https://github.com/apache
\(^2\)https://github.com/google
\(^3\)https://github.com/spring-projects
Table (4.1) Corpus of additional 33 Java open source projects: size and other essential statistics

<table>
<thead>
<tr>
<th># Project</th>
<th>#Pairs</th>
<th>production*</th>
<th>test</th>
<th>Mutants</th>
<th>Mutants</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 commons-text</td>
<td>66</td>
<td>23,393</td>
<td>16,065</td>
<td>3,888</td>
<td>17%</td>
</tr>
<tr>
<td>2 commons-cli</td>
<td>12</td>
<td>4,922</td>
<td>2,200</td>
<td>596</td>
<td>12%</td>
</tr>
<tr>
<td>3 jackson-core</td>
<td>13</td>
<td>3,513</td>
<td>1,083</td>
<td>701</td>
<td>20%</td>
</tr>
<tr>
<td>4 dagger</td>
<td>6</td>
<td>892</td>
<td>826</td>
<td>1,769</td>
<td>198%</td>
</tr>
<tr>
<td>5 spring-ws</td>
<td>132</td>
<td>18,118</td>
<td>14,615</td>
<td>13,387</td>
<td>74%</td>
</tr>
<tr>
<td>6 closure-stylesheets</td>
<td>85</td>
<td>20,587</td>
<td>16,218</td>
<td>19,441</td>
<td>94%</td>
</tr>
<tr>
<td>7 commons-configuration</td>
<td>127</td>
<td>48,910</td>
<td>46,645</td>
<td>3,980</td>
<td>8%</td>
</tr>
<tr>
<td>8 commons-dbutils</td>
<td>36</td>
<td>7,914</td>
<td>5,252</td>
<td>4,102</td>
<td>52%</td>
</tr>
<tr>
<td>9 joda-collect</td>
<td>8</td>
<td>1,760</td>
<td>943</td>
<td>3,059</td>
<td>174%</td>
</tr>
<tr>
<td>10 jimfs</td>
<td>30</td>
<td>7,950</td>
<td>7,495</td>
<td>11,280</td>
<td>142%</td>
</tr>
<tr>
<td>11 exp4j</td>
<td>6</td>
<td>1,148</td>
<td>4,137</td>
<td>3,924</td>
<td>342%</td>
</tr>
<tr>
<td>12 zxing</td>
<td>56</td>
<td>9,654</td>
<td>5,769</td>
<td>41,944</td>
<td>434%</td>
</tr>
<tr>
<td>13 commons-validator</td>
<td>47</td>
<td>13,322</td>
<td>9,989</td>
<td>15,251</td>
<td>114%</td>
</tr>
<tr>
<td>14 joda-time</td>
<td>64</td>
<td>30,782</td>
<td>26,904</td>
<td>3,611</td>
<td>12%</td>
</tr>
<tr>
<td>15 commons-beam</td>
<td>20</td>
<td>8,995</td>
<td>1,497</td>
<td>15,513</td>
<td>172%</td>
</tr>
<tr>
<td>16 error-prone</td>
<td>58</td>
<td>7,579</td>
<td>6,430</td>
<td>7,429</td>
<td>98%</td>
</tr>
<tr>
<td>17 commons-compiler</td>
<td>41</td>
<td>18,709</td>
<td>13,203</td>
<td>3,512</td>
<td>19%</td>
</tr>
<tr>
<td>18 joda-money</td>
<td>6</td>
<td>2,419</td>
<td>3,210</td>
<td>2,333</td>
<td>96%</td>
</tr>
<tr>
<td>19 commons-email</td>
<td>14</td>
<td>5,171</td>
<td>4,254</td>
<td>3,969</td>
<td>77%</td>
</tr>
<tr>
<td>20 spring-data-commons</td>
<td>165</td>
<td>37,806</td>
<td>28,179</td>
<td>28,377</td>
<td>75%</td>
</tr>
<tr>
<td>21 commons-numbers</td>
<td>40</td>
<td>11,963</td>
<td>10,701</td>
<td>42,647</td>
<td>356%</td>
</tr>
<tr>
<td>22 orson-charts</td>
<td>63</td>
<td>23,609</td>
<td>7,356</td>
<td>65,058</td>
<td>270%</td>
</tr>
<tr>
<td>23 jackson-databind</td>
<td>29</td>
<td>19,491</td>
<td>5,222</td>
<td>3,032</td>
<td>16%</td>
</tr>
<tr>
<td>24 joda-primitives</td>
<td>32</td>
<td>8,343</td>
<td>4,906</td>
<td>15,005</td>
<td>180%</td>
</tr>
<tr>
<td>25 commons-net</td>
<td>34</td>
<td>14,386</td>
<td>5,823</td>
<td>17,992</td>
<td>125%</td>
</tr>
<tr>
<td>26 undertow</td>
<td>19</td>
<td>8,092</td>
<td>2,354</td>
<td>20,031</td>
<td>248%</td>
</tr>
<tr>
<td>27 raml-java-parser</td>
<td>3</td>
<td>385</td>
<td>452</td>
<td>439</td>
<td>127%</td>
</tr>
<tr>
<td>28 spring-data-keyvalue</td>
<td>15</td>
<td>2,530</td>
<td>2,570</td>
<td>1,895</td>
<td>75%</td>
</tr>
<tr>
<td>29 commons-fileupload</td>
<td>6</td>
<td>1,998</td>
<td>850</td>
<td>175</td>
<td>9%</td>
</tr>
<tr>
<td>30 truth</td>
<td>37</td>
<td>11,739</td>
<td>14,111</td>
<td>9,610</td>
<td>82%</td>
</tr>
<tr>
<td>31 highwheel</td>
<td>28</td>
<td>1,831</td>
<td>2,012</td>
<td>2,570</td>
<td>140%</td>
</tr>
<tr>
<td>32 commons-pool</td>
<td>13</td>
<td>6,780</td>
<td>3,614</td>
<td>1,444</td>
<td>61%</td>
</tr>
<tr>
<td>33 commons-imaging</td>
<td>73</td>
<td>12,473</td>
<td>4,590</td>
<td>3,316</td>
<td>27%</td>
</tr>
</tbody>
</table>

| Total            | 1384   | 397,164 279,475 | 26% | 374,030 40% |
| Min              | 3      | 385 452 | 17% | 175 8% |
| Mean             | 42     | 12,035 8,408  | 80% | 11,334 120% |
| Median           | 32     | 8,343 5,222  | 75% | 3,980 95% |
| Max              | 165    | 4,8910 46,645  | 30% | 65,058 43% |

* All code size values are in Lines of Code (LOC)

For the sake of reproducibility, we ran a mining tool, which is part of the open-source Spartanizer project, used in previous studies (Ma’ayan 2018; Gil et al. 2019). 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 and that any simple AST-based analysis tool or library could calculate the entire set of features.

Table 4.2 lists the 33 new features (also called static code metrics) introduced in this study, organizing these in four categories, as illustrated in Fig. 4.2:

 bows

https://preview.tinyurl.com/y7zfvbzw
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 4.2 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 (Grissom and Kim 2005), 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` 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 (4.2) Newly introduced static code features and the correlation (expressed in Cliff’s $d$) of their values on test and production code with test effectiveness

<table>
<thead>
<tr>
<th>#</th>
<th>Type</th>
<th>Feature</th>
<th>Description</th>
<th>Cliff’s $d$ (Test)</th>
<th>Cliff’s $d$ (Prod)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Text</td>
<td>#Characters</td>
<td>No. non-white space characters</td>
<td>+0.29</td>
<td>-0.30</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>#Words</td>
<td>No. identifiers (simple-name) used</td>
<td>+0.27</td>
<td>-0.32</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>Strings</td>
<td>Total length in characters of string literals</td>
<td>+0.08</td>
<td>-0.33</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>Vocabulary</td>
<td>No. distinct identifiers used</td>
<td>+0.29</td>
<td>-0.38</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>#Punctuation</td>
<td>No. non-alpha numeric characters</td>
<td>+0.30</td>
<td>-0.30</td>
</tr>
<tr>
<td></td>
<td>Topology</td>
<td>#Nodes</td>
<td>No. nodes</td>
<td>+0.27</td>
<td>-0.33</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>Depth</td>
<td>Node depth (averaged)</td>
<td>+0.12</td>
<td>+0.34</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>Depth2</td>
<td>Node depth squared (averaged)</td>
<td>+0.38</td>
<td>+0.40</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>Max-depth</td>
<td>Maximal node depth</td>
<td>+0.32</td>
<td>-0.19</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>Deg2</td>
<td>Square of node degree (averaged)</td>
<td>+0.37</td>
<td>-0.44</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>Deg!</td>
<td>No. permutations of node children’s (averaged)</td>
<td>+0.10</td>
<td>+0.03‡</td>
</tr>
<tr>
<td></td>
<td>Grammar</td>
<td>#If</td>
<td>No. if statements</td>
<td>+0.08</td>
<td>-0.29</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td>#Else</td>
<td>No. else clauses</td>
<td>+0.03</td>
<td>-0.23</td>
</tr>
<tr>
<td>14</td>
<td></td>
<td>#Loop</td>
<td>No. loop statements</td>
<td>+0.19</td>
<td>-0.10</td>
</tr>
<tr>
<td>15</td>
<td></td>
<td>#Break</td>
<td>No. break from loop statements</td>
<td>0.00‡</td>
<td>-0.02‡</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td>#Continue</td>
<td>No. continue of loop statements</td>
<td>0.00‡</td>
<td>-0.05‡</td>
</tr>
<tr>
<td>17</td>
<td></td>
<td>#Try</td>
<td>No. try statements</td>
<td>+0.12</td>
<td>-0.08</td>
</tr>
<tr>
<td>18</td>
<td></td>
<td>#Catch</td>
<td>No. catch clauses</td>
<td>+0.03‡</td>
<td>-0.07</td>
</tr>
<tr>
<td>19</td>
<td></td>
<td>#&amp;&amp;</td>
<td>No. short-circuit conjunctions ($&amp;&amp;$ operators)</td>
<td>-0.01†</td>
<td>-0.15</td>
</tr>
<tr>
<td>20</td>
<td></td>
<td>#</td>
<td></td>
<td></td>
<td>No. short-circuit disjunctions ($</td>
</tr>
<tr>
<td>21</td>
<td></td>
<td>#?:</td>
<td>No. conditional operators (?:)</td>
<td>0.00†</td>
<td>-0.10</td>
</tr>
<tr>
<td>22</td>
<td></td>
<td>Comments</td>
<td>Length in characters of code comments</td>
<td>-0.13</td>
<td>-0.19</td>
</tr>
<tr>
<td>23</td>
<td></td>
<td>#Methods</td>
<td>No. methods in a class</td>
<td>+0.42</td>
<td>-0.27</td>
</tr>
<tr>
<td>24</td>
<td></td>
<td>Variety</td>
<td>No. distinct Java types spanned by nodes</td>
<td>+0.28</td>
<td>-0.31</td>
</tr>
<tr>
<td>25</td>
<td></td>
<td>#Expressions</td>
<td>No. expression nodes</td>
<td>+0.28</td>
<td>-0.33</td>
</tr>
<tr>
<td>26</td>
<td></td>
<td>#Invocations</td>
<td>No. method invocation nodes</td>
<td>+0.29</td>
<td>-0.32</td>
</tr>
<tr>
<td>27</td>
<td></td>
<td>#Access</td>
<td>No. field access nodes</td>
<td>+0.03</td>
<td>-0.20</td>
</tr>
<tr>
<td>28</td>
<td></td>
<td>#Primitive</td>
<td>No. primitive type nodes</td>
<td>+0.38</td>
<td>-0.25</td>
</tr>
<tr>
<td>29</td>
<td></td>
<td>#Literal</td>
<td>No. numeric literal nodes</td>
<td>+0.18</td>
<td>-0.22</td>
</tr>
<tr>
<td>30</td>
<td></td>
<td>#Misuse</td>
<td>No. incidents of misuse of JUnit interface</td>
<td>-0.21</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td></td>
<td>Junit</td>
<td>Does test class use JUnit?</td>
<td>-0.03‡</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td></td>
<td>Hamcrest</td>
<td>Does the test class use Hamcrest?</td>
<td>-0.02‡</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td></td>
<td>Mockito</td>
<td>Does the test class use Mockito?</td>
<td>+0.25</td>
<td></td>
</tr>
</tbody>
</table>

‡ Statistically insignificant: $p \geq 0.05$

**Definition 4.2.1.** The *lighter* set of 66 features consists of: features 1–29, from Table 4.2, measured both for test and production classes, and features 30–33 of Table 4.2, measured for test code only, and the four size metrics of Table 2.2.

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 Chapter 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\(^5\) or C++ and

\(^5\)https://docs.python.org/3/library/ast.html
C compiler front ends such as Clangast⁶ are sufficient.

### 4.3 Neural Network Models

We now describe the design of the neural network models.

**Data normalization** The first step prior to training was data normalization (Hall 1999). 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 (Anders and Korn 1999); 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 (Rafiq et al. 2001; Goodfellow et al. 2016; Gulli and Pal 2017), 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 Keras⁷ Python package (Gulli and Pal 2017), 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 (Hsu et al. 2003), which randomly samples $k$ hyperparameter values from the entire hyperparameter space. Studies (Bergstra and Bengio 2012) 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 (Mosteller and Tukey 1968), 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 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

---

⁶https://clang.llvm.org/docs/index.html
⁷https://keras.io/
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 \texttt{RandomGridSearchCV} utility provided by scikit-learn\textsuperscript{8}.

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 (Stone 1974). 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. 4.3 illustrates a 10-fold nested cross-validation. A full description of the nested cross-validation approach can be found in Grano et al. (2019a).

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

\textbf{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 (Grano et al. 2019b).

\section*{4.4 Measurements}

Since test effectiveness is a binary judgment, we used the famous $2 \times 2$ confusion matrix described in Fig. 4.4 to measure the prediction quality. We used the following common measurements:

\textbf{Accuracy (Acc.):}
\[
\text{Acc.} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}
\]

\textbf{Precision (Prec.):}
\[
\text{Prec.} = \frac{\text{TP}}{\text{TP} + \text{FP}}
\]

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|}
\hline
\textbf{Class} & \textbf{TP} & \textbf{FP} & \textbf{FN} \\
\hline
\textbf{Positive} & \text{True Positives} & \text{False Positives} & \text{False Negatives} \\
\hline
\textbf{Negative} & \text{True Negatives} & & \text{True Negatives} \\
\hline
\end{tabular}
\end{table}

\textsuperscript{8}https://scikit-learn.org
Figure (4.4) $2 \times 2$ confusion matrix

Recall (Rec.):
\[
\frac{TP}{TP + FN}
\]

F1:
\[
2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}
\]

We also employed two additional measures of error, Mean Absolute Error (MAE):
\[
\frac{\sum_{i=1}^{n}|e_i|}{n}
\]

and Area Under Curve (AUC). Except as regards MAE, higher values of measures are better.
Chapter 5

Results

5.1 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 5.1 compares the original GPG results for the dynamic experiment to an identical model trained and evaluated using our extended dataset.

Table (5.1) Comparing the GPG RFC model with the original dataset and the extended dataset while including dynamic information

<table>
<thead>
<tr>
<th># Dataset</th>
<th>Acc.</th>
<th>Prec.</th>
<th>Rec.</th>
<th>F1</th>
<th>AUC</th>
<th>MAE</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) Original</td>
<td>.948</td>
<td>.940</td>
<td>.960</td>
<td>.949</td>
<td>.949</td>
<td>.051</td>
</tr>
<tr>
<td>(2) Extended</td>
<td>.946</td>
<td>.931</td>
<td>.965</td>
<td>.947</td>
<td>.946</td>
<td>.054</td>
</tr>
<tr>
<td>(1)-(2)</td>
<td>.002</td>
<td>.009</td>
<td>-.005</td>
<td>-.002</td>
<td>-.003</td>
<td>-.003</td>
</tr>
</tbody>
</table>

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

Table (5.2) Comparing the GPG RFC model with the original dataset and the extended dataset while relying only on static information

<table>
<thead>
<tr>
<th># Dataset</th>
<th>Acc.</th>
<th>Prec.</th>
<th>Rec.</th>
<th>F1</th>
<th>AUC</th>
<th>MAE</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) Original</td>
<td>.864</td>
<td>.864</td>
<td>.865</td>
<td>.864</td>
<td>.864</td>
<td>.137</td>
</tr>
<tr>
<td>(2) Extended</td>
<td>.837</td>
<td>.822</td>
<td>.861</td>
<td>.840</td>
<td>.837</td>
<td>.163</td>
</tr>
<tr>
<td>(1)-(2)</td>
<td>.027</td>
<td>.042</td>
<td>.004</td>
<td>.024</td>
<td>.027</td>
<td>-.026</td>
</tr>
</tbody>
</table>

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.

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 (Andrews et al. 2006; Frankl and Weiss 1991; Frankl and Iakounenko 1998) and contradicts other studies that did not find such a correlation (Inozemtseva and Holmes 2014).

The robustness of the dynamic model should dispel dataset-selection concerns raised by Zhang et al. (2020) for supervised learning models. As demonstrated, the results were similar regardless of the dataset when we included dynamic information. However, Zhang et al.’s (2020) 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. 5.2 to 5.4. 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.

### 5.2 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.

#### 5.2.1 Predicting effectiveness with dynamic information

Table 5.3 compares the quality of prediction with dynamic information (test coverage).

Row (1) in Table 5.3 describes the results of running the original GPG experiment with our extended dataset. It 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 5.3 provides the measures of prediction when using a DNN model,
Table (5.3) 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 2.2) vs. full set (Table 2.2 and Definition 4.2.1)

<table>
<thead>
<tr>
<th>#</th>
<th>Model</th>
<th>Features</th>
<th>Acc.</th>
<th>Prec.</th>
<th>Rec.</th>
<th>F1</th>
<th>AUC</th>
<th>MAE</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>RFC</td>
<td>Light</td>
<td>.946</td>
<td>.931</td>
<td>.965</td>
<td>.947</td>
<td>.946</td>
<td>.054</td>
</tr>
<tr>
<td>(2)</td>
<td>DNN</td>
<td>Light</td>
<td>.933</td>
<td>.916</td>
<td>.954</td>
<td>.934</td>
<td>.933</td>
<td>.067</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S.D.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>.254%</td>
<td>.697%</td>
</tr>
<tr>
<td></td>
<td>(2)-(1)</td>
<td></td>
<td></td>
<td>-.013</td>
<td>-.015</td>
<td>-.011</td>
<td>-.013</td>
<td>-.013</td>
</tr>
<tr>
<td>(3)</td>
<td>DNN</td>
<td>Full</td>
<td>.935</td>
<td>.920</td>
<td>.954</td>
<td>.936</td>
<td>.935</td>
<td>.065</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S.D.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>.21%</td>
<td>.66%</td>
</tr>
<tr>
<td></td>
<td>(3)-(2)</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>(3)-(1)</td>
<td></td>
<td></td>
<td>-.011</td>
<td>-.011</td>
<td>-.011</td>
<td>-.011</td>
<td>-.011</td>
</tr>
</tbody>
</table>

while using the “light” set of features, i.e., the features listed in Table 2.2. 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 5.3 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 4.2.1; 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 5.3 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 5.3, 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.
5.2.2 Predicting effectiveness without dynamic information

Table 5.4 is structured much like Table 5.3, 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 (5.4) Quality of test effectiveness prediction: Same as Table 5.3, but without dynamic information

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

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.

5.3 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 Chapter 4 to predict \( \mu \), 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 5.5 presents the value of $\rho$, Pearson’s coefficient of correlation between the obtained predicted mutation score and its actual value, a standard measurement for prediction quality assessment.

Table (5.5)  Pearson correlation between the actual and predicted mutation score

<table>
<thead>
<tr>
<th>Features</th>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Light</td>
<td>.848 ± 0.34%</td>
<td>.637 ± 0.90%</td>
</tr>
<tr>
<td>Full</td>
<td>.859 ± 0.29%</td>
<td>.683 ± 0.85%</td>
</tr>
</tbody>
</table>

We stress that the coefficient was computed over the test portion of the data as is usual in learning algorithms. Values 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 5.5.

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. 5.1 demonstrates the prediction quality visually.

![Figure (5.1) Actual vs. prediction of mutation score with the full set of features and statement-coverage information](image)

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

Table 5.5 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 \( \rho \) 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. 5.2, a scatter plot of the prediction vs. actual \( \mu \), when this feature is missing, and comparing it to Fig. 5.1 above.

![Scatter plot of predicted vs. actual mutation score](image)

Figure (5.2) Actual vs. prediction of mutation score with the full set of features, without statement-coverage information

Table 5.6 gives the mean square error of the prediction depending on the dataset.

Table (5.6) Mean average error of predicted mutation score

<table>
<thead>
<tr>
<th>Features</th>
<th>Statement coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>Light</td>
<td>( .112 \pm 0.81%)</td>
</tr>
<tr>
<td>Full</td>
<td>( .107 \pm 1.39%)</td>
</tr>
</tbody>
</table>

Inspecting Table 5.6, we can draw the same qualitative conclusions as in Table 5.5 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 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 5.4) 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 (Zhang et al. 2019b, 2020) to predict mutation score. As mentioned in Chapter 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.

### 5.4 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 5.7 compares the two sets in terms of their ability to predict the effectiveness of test cases.

Row (1) of Table 5.7 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 4.2.1.

The two subsequent rows of Table 5.7 compare the lighter set of features with
Table (5.7) Quality of prediction of test effectiveness with different sets of static features

<table>
<thead>
<tr>
<th># Features</th>
<th>Acc.</th>
<th>Prec.</th>
<th>Rec.</th>
<th>F1</th>
<th>AUC</th>
<th>MAE</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) Full</td>
<td>.878</td>
<td>.864</td>
<td>.898</td>
<td>.880</td>
<td>.878</td>
<td>.122</td>
</tr>
<tr>
<td>(2) Light</td>
<td>.851</td>
<td>.840</td>
<td>.867</td>
<td>.853</td>
<td>.851</td>
<td>.149</td>
</tr>
<tr>
<td>(3) Lighter</td>
<td>.860</td>
<td>.844</td>
<td>.885</td>
<td>.863</td>
<td>.860</td>
<td>.140</td>
</tr>
</tbody>
</table>

(1)-(3) .018 .020 .013 .017 .018 -.013
(3)-(2) .009 .004 .018 .010 .009 -.009

the two baselines. To minimize clutter, the table does not supply information on the statistical error as was done, e.g., in Table 5.3. 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 5.8 repeats the comparison of Table 5.7, but this time in terms of the prediction quality of mutation score.

Table (5.8) Quality of prediction of mutation score using only the “lighter” (Definition 4.2.1) set of features using our DNN model and only static features

<table>
<thead>
<tr>
<th># Features</th>
<th>MAE</th>
<th>ρ</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) Full</td>
<td>.156</td>
<td>.683</td>
</tr>
<tr>
<td>(2) Light</td>
<td>.165</td>
<td>.637</td>
</tr>
<tr>
<td>(3) Lighter</td>
<td>.161</td>
<td>.666</td>
</tr>
</tbody>
</table>

(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 pro-
gramming, 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.
Chapter 6

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 (Ma’ayan 2018; Gil et al. 2019). 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 (Rothermel et al. 2001); 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 (Offutt and Jie Pan 1996), 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. (2001), who considered killed mutants as non-equivalent.

That said, in this study, we tried to estimate the mutation score as calculated by PIT (Coles et al. 2016), 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 (Tufano et al. 2019b).

**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 (Krstajic et al. 2014). 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.
Chapter 7

Discussion and 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 while considering 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 (Nerur et al. 2005). 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 (Nie and Leung 2011; Yoo and Harman 2012) as an additional input factor. Another practical application is integration with tools such as SonarQube\(^1\) (Campbell and Papapetrou 2013) and CheckStyle\(^2\), and integration with CI tools (Beck 1999) such as Travis CI\(^3\). 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 (Inozemtseva and Holmes 2014; Zhang and Mesbah 2015). 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 (Brown et al. 2017) 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 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 (Hotelling 1933) and factor analysis (Mulaik 2009).

**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 consider-

---

1 https://www.sonarqube.org/
2 https://checkstyle.sourceforge.io/
3 https://travis-ci.com/
able engineering efforts involved in a recent Java tool for detecting test smells (Peruma et al. 2020). 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 Chapter 3 regarding the GPG features are valid to other works, such as PMT and CBUA (Zhang et al. 2019b, 2020). Future studies should explore whether our simple features can replace or enhance sets of features in works that focus on mutant-level prediction.

**Identifying testable code and testful tests** Our models use test code and production code measurements as features. It would be interesting to explore how using a subset of the features would change the results.

In Appendix A, we present an initial experiment that explores test effectiveness and mutation score prediction relying solely on test or production code. Generally, we observe a decrease in prediction power while training only with a single group of features. However, the results reflect that it is still possible to predict the mutation score or effectiveness of tests relying solely on either test or production features.

In other words, we can detect production code leading to good tests or, conversely, estimate mutation score by inspecting the tests before the production code was even written. A measurement relying solely on test features is useful in Test-Driven Development (TDD) (Beck 2003; Astels 2003), where tests are written before the code.

**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.
Appendix A

Identifying Testable Code and Testful Tests

In this appendix, we present preliminary results of evaluation of our DNN models with test-code features and production-code features. Here, we used the original GPG data-set.

Table A.1 compares the ability of test and production code metrics to predict the effectiveness of tests.

Table (A.1) Quality of prediction of test effectiveness with different test or production static features

<table>
<thead>
<tr>
<th># Features</th>
<th>Acc.</th>
<th>Prec.</th>
<th>Rec.</th>
<th>F1</th>
<th>AUC</th>
<th>MAE</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) Full</td>
<td>.911</td>
<td>.903</td>
<td>.921</td>
<td>.911</td>
<td>.911</td>
<td>.089</td>
</tr>
<tr>
<td>(2) Test</td>
<td>.831</td>
<td>.827</td>
<td>.836</td>
<td>.830</td>
<td>.831</td>
<td>.169</td>
</tr>
<tr>
<td>(3) Production</td>
<td>.809</td>
<td>.794</td>
<td>.834</td>
<td>.812</td>
<td>.809</td>
<td>.191</td>
</tr>
</tbody>
</table>

| (1)-(2)      | .080 | .076  | .085 | .081 | .080| -.080|
| (1)-(3)      | .102 | .109  | .087 | .099 | .102| -.102|
| (2)-(3)      | .022 | .033  | .002 | .018 | .022| -.022|

Table row (1) presents the baseline for our comparisons, which is our measures of quality of prediction while using the union of the two sets of features. Rows (2) and (3) provide prediction measures while considering only test code or production code features, respectively. As expected, we see in rows labeled with (1)-(2) and (1)-(3) that the quality of prediction degrades drastically, in terms of all measures, when considering only the test or production features. However, we see that a model considers only test code features performs significantly better than the one considers only production code in all measures. This is a surprising result since the statistics on our features, as reported in Definition 4.2.1, and statistics from GPG demonstrate higher correlations between production code features and effectiveness than the correlation of test code features.
Table A.2 repeats the comparisons of Table A.1, but in terms of quality of prediction of mutation score.

Table A.2 Quality of prediction of mutation score using only test or production set of features using DNN only static feature

<table>
<thead>
<tr>
<th># Features</th>
<th>MAE</th>
<th>ρ</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) Full</td>
<td>.128</td>
<td>.753</td>
</tr>
<tr>
<td>(2) Test</td>
<td>.172</td>
<td>.557</td>
</tr>
<tr>
<td>(3) Production</td>
<td>.187</td>
<td>.517</td>
</tr>
<tr>
<td>(1)-(2)</td>
<td>-.044</td>
<td>.196</td>
</tr>
<tr>
<td>(1)-(3)</td>
<td>-.059</td>
<td>.236</td>
</tr>
<tr>
<td>(2)-(3)</td>
<td>-.015</td>
<td>.040</td>
</tr>
</tbody>
</table>

Again, we see that using only test code features provides slightly better results in all measures, which confirms our findings from Table A.1.
Bibliography


K. Beck. 1999. Embracing change with extreme programming. *Computer* 32, 10 (1999), 70–77. DOI:http://dx.doi.org/10.1109/2.796139


Martin Fowler. 2018. *Refactoring: improving the design of existing code*. Addison-Wesley, Boston, USA.


Harold Hotelling. 1933. Analysis of a complex of statistical variables into principal components. *Journal of educational psychology* 24, 6 (1933), 417.


אנומ Shame the initial efforts that were launched, and that are efforts in which we are currently making efforts to advance them, and not to continue with problems, are expanded in order to reach 2019.

If we open the data in the problems of the set of data from the article mentioned in the study, it becomes clear that the findings of the original study and the return to the study of the system, when we use the static properties of the code, there is a significant decrease in its ability to determine the effectiveness when we use the classical model.

After that, we examined whether the use of deep neural networks can be used to predict the effectiveness of the systems. For this purpose, we replaced the classical learning model mentioned in the previous article with a deep learning model. We found that the use of deep neural networks leads to better results in predicting the effectiveness of the model of the classical methods when we use the static properties of the code.

In the case of using deep neural networks, we presented, for the first time, a model of prediction that indicates that the code is not a normal that is a code, but rather a code that can be split into different parts. Furthermore, according to the code, it is possible to classify into effective and not effective, we demonstrate training of the neural network on the basis of splitting the code into different parts, in order to use static properties of the code.

The results of the model can be used as a resource in a broader sense of the problem, when we rely on static properties, and in the future, in the context of other studies, we will use the properties "human," that is, the properties that are a result of the study of the equation that was known for years and includes mathematical connections between basic properties of the code, for example, complexity of the code, as well as subjective measures of the code, such as the study of the community research.

Finally, we examine the implications of our results, in various potential ways in different industries, and the potential for the study of a wide range of areas in software engineering, where we found our results and draw conclusions from them.

As a result, our study is based on the data that was mentioned in the article, as mentioned in other studies in the field, we use the properties "human," that is, the properties that are a result of the study of the equation that was known for years and includes mathematical connections between basic properties of the code, for example, complexity of the code, as well as subjective measures of the code, such as the study of the community research.

In conclusion, our study is based on the data that was mentioned in the article, as mentioned in other studies in the field, we use the properties "human," that is, the properties that are a result of the study of the equation that was known for years and includes mathematical connections between basic properties of the code, for example, complexity of the code, as well as subjective measures of the code, such as the study of the community research.

In conclusion, our study is based on the data that was mentioned in the article, as mentioned in other studies in the field, we use the properties "human," that is, the properties that are a result of the study of the equation that was known for years and includes mathematical connections between basic properties of the code, for example, complexity of the code, as well as subjective measures of the code, such as the study of the community research.

In conclusion, our study is based on the data that was mentioned in the article, as mentioned in other studies in the field, we use the properties "human," that is, the properties that are a result of the study of the equation that was known for years and includes mathematical connections between basic properties of the code, for example, complexity of the code, as well as subjective measures of the code, such as the study of the community research.

In conclusion, our study is based on the data that was mentioned in the article, as mentioned in other studies in the field, we use the properties "human," that is, the properties that are a result of the study of the equation that was known for years and includes mathematical connections between basic properties of the code, for example, complexity of the code, as well as subjective measures of the code, such as the study of the community research.

In conclusion, our study is based on the data that was mentioned in the article, as mentioned in other studies in the field, we use the properties "human," that is, the properties that are a result of the study of the equation that was known for years and includes mathematical connections between basic properties of the code, for example, complexity of the code, as well as subjective measures of the code, such as the study of the community research.
תקציר
לאורכ שנים רבבותה המחקרים וה://מ órgãoיםומייצים על מתן מענה מדדיים יעילים תח eSports מaska של הבידון המחקר

מא_EmKeyboard
ומספר מארזיו ישנה מספק את בובו ולפי למקצ コזוד יאני והבר בידיו גודל והבר בידיו

 RCSY שורוד.

בבדיקת כיסוי מונטיון נוצרו מונטיון של הקוד, באופן, גריסאות שشهاد של הקוד שאר כחללי

ב olduk ניסיון להקטיןعدد הקולאזים ואת מחרזים לדלתות טעויות פמוזליאית שמתכנתה היה

משי להכניס קוד ביהח העתון. ג洿 של אוס אופון מומטסיו, פורץ בידיו השימש והפוריסק

מקור בולוצקבאם הוא מחזאת והשיג פמוזליאית ואילא. פמיליא מאש קודם שהמוני מונטיון נבדק בינש

טסטעים מפי יופי ברוב פמוזליאくて הקוד. לארח הריצת של מונטיון ריבים, תחביב הבכסי

מונטיון של הקוד כחמש בשני מוספים המונטיון הוא עידי ה_hן עלי בשתי ציון בול ככל המונטיון

שונררה.

ב(ray קולאזים וお金を מונטיון, ואת מחרזים הבכסי, שגריסאות מושל של המונטיון, יזור

של הקוד מונטיון הוריצה את כל הבכסי הבכסי. באשר Bereich התוכנה מודרני, כולם לוחזרו את

الطמון המונטיון שברובים ימי ירישught. בתוכנה ומדורה ההוריצה בבר מים הבכסי

כיסוי מונטיון, השישת המוחמצה נגזרה כי הס UIImagePickerController של המונטיון דליות

על מת גזור את הbinary בבכסי הגזור, המכدرك המונטיון מצפה למתכון מוספר

המידיקונטוני יגור, הזאת של הליך המידיקונטוני, והחרתה לשטייט. ושתורע, לכל השטיות

ה徘ותר לא מתחרה הנולים את הבכסי ימי עיון יייש המונטיון והחרתה לשטייט

ועל, כ, הבניה הנקרא את התוכנה.

בעובד, ו, אף פסיים ינתה של המונטיון ומפורטת למחרזים מנה pakistan עד בח. ימי מניקת מעונד

את הבכסי המונטיון סטייס במאונטרה למידה עומק את תוכנה של הקוד. בכ, הבניה מעונד

מאומד, יכילו לשגש את תוכנה הקוד ימי אדם, חיות הבכסי המונטיון ופועלו מיידי.

בשנת 2019, בпотенци מחקר באנטרופסים עד מונטיון לסיס מ Md דמיון מונטיון כלשהי

יחיוס הקואטקאית סטייס, כולם, חיות בינאי, האה סחיים הם אפקטיוידים ואילא אפקטיוידים,

וחות ה рагוס על כיסוי מונטיון. בעובדה המוחמצה נועשה שים בוקבוק של 67 תוכנה סטייס

ידimized il קוד.
The research was conducted under the guidance of Professor Y. Gil, in the Faculty of Science.

The results of the research are published in articles written by the research team in professional conferences and journals during the period of the researcher's research.


חיווי משופר לבליטי מורכבי של טסטים

ה弋ור על מחקר

לשנים מובהק חלקי של הדרישות לקבצי התוכן המוגשים למדעימדע המועברים

דור דוד מען

הוושה לטכנולוגיה — מכון טכנולוגי לישראל
שיבת התשפ”א — חיפה
ינואר 2021
חיזוי ישות מקוונת ומוטני של תסטים

דור זוד מעין