TU WIEN Informatics

Predicting Mutation-Scores with a Cheap Quality Model

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Software Engineering & Internet Computing

by

Stefan Heigl, Bsc
Registration Number 01126290

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.-Prof. Mag. Dr. Manuel Wimmer
Assistance: Univ. Ass. DI Daniel Lehner

Vienna, 1st May, 2023

Stefan Heigl	Manuel Wimmer

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Stefan Heigl, Bsc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 1. Mai 2023

Stefan Heigl

Acknowledgements

I would like to thank Daniel Lehner, for his support, guidance, and patience throughout my research. His insights, feedback, and advice have been essential in shaping the direction this thesis.

I would also like to thank Nicole Heiden. Without her support and encouragement during the last years this thesis wouldn't have been possible.

Finally, I would like to thank my family and friends for their unwavering support, encouragement, and understanding throughout my academic journey.

Kurzfassung

Eine gute Test-Suite zu halten ist wichtig, da ansonsten ein Software-Projekt unwartbar werden kann. Es gibt mehrere Techniken, um die Qualität einer Test-Suite zu bestimmen, wobei Mutation Testing die bedeutendste ist. Mutation Testing hat aber Nachteile, der größte davon ist die Laufzeit, die es für große Software-Projekte unpraktisch macht. Diese Arbeit versucht dieses Problem zu umgehen, indem ein neues Qualitätsmodell aufgebaut wird, das die Ergebnisse von Mutation Testing auf der Grundlage anderer Evaluierungsmetriken für Test-Suiten vorhersagt.

Zuerst wurde eine Systematic Mapping Study durchgeführt, um Methoden zur Evaluierung von Test-Suiten zu ermitteln. Die gefundenen Methoden wurden hinsichtlich ihrer Fähigkeit, Mutation Scores vorherzusagen, bewertet. Anschließend wurden sie verwendet, um ein neues Qualitätsmodell mithilfe von Machine Learning aufzubauen. Mehrere Regressionsalgorithmen wurden evaluiert.

Es gibt viele Methoden zur Bestimmung der Test-Suite-Qualität. Diese Arbeit fand 56 Test-Suite Evaluierungsmetriken. Die meisten dieser Veröffentlichungen verwenden entweder Mutation Testing oder eine Art von Code Coverage. Breitere Forschung zum Thema Test-Suite-Evaluation begann 2013. Mehrere Coverage Metriken und Test Smells wurden hinsichtlich ihrer Fähigkeit, Mutation Scores vorherzusagen, evaluiert. Die meisten von ihnen sind nicht dazu in der er Lage Mutation Scores vorherzusagen. Nur Branch Coverage zeigte mit einem R^2 von ~0.65 ein vielversprechendes Ergebnis. Die Verwendung von Random Forest zur Kombination von Test-Suite Evaluierungsmetriken führte zum besten Qualitätsmodell mit einem R^2. Diese Modell ist um etwa 50% besser als die beste Test-Suite Evaluierungsmetrik allein. Die Verwendung weiterer Test-Suite Evaluierungsmetrik im Random Forest Modell erhöht die Qualität nur geringfügig.

Das neue Qualitätsmodell, das in dieser Arbeit vorgestellt wird, bietet eine effektive Möglichkeit, Mutation Testing Ergebnisse vorherzusagen. In der Praxis kann dieses Modell genutzt werden, um die Testzeit von Software-Projekten zu verbessern, währenddessen eine qualitative Aussagen zur Test-Suite-Qualität immer noch gegeben ist.

Abstract

Keeping a good test-suite is important, otherwise a software project may get unmaintainable. There are several techniques to determine the quality of a test-suite, with mutation testing being the most significant one. But mutation testing has its disadvantages, the major one being runtime which prevents it from being practical for large software projects. This thesis aims to circumvent this problem by building a novel quality model that predicts mutation testing results based on other test-suite evaluation metrics.

First a systematic mapping study was conducted to determine methods to evaluate test-suites. The found methods were evaluated, in terms of ability to predict mutation scores. Then they were used to build a novel quality framework using machine learning. Several regression algorithms were evaluated.

There are a lot of methods to determine test-suite quality. This study found 56 metrics that do that. Most of these publications either use mutation testing or some kind of coverage. More intense research into the topic of test-suite evaluation started in 2013. Several coverages and test smells haven been evaluated, in terms of how well they predict mutation scores. Most of them have no meaningful ability to predict mutation score. Only branch coverage showed some promise with a R^2 of ~0.65. Using random forest to combine test-suite evaluation methods lead to the best quality model with a R^2. It is about 50% better than the best test-suite evaluation method on its own. Using more test-suite evaluation methods increases the random forest model only slightly.

The novel quality model presented in this thesis provides an effective way to predict mutation testing results. Practitioners can use this model to improve the test time of their projects, while still having qualitative feedback about their test-suite quality.

Contents

CHAPTER 1

Introduction

Keeping the quality of a test-suite, which is maintainable, but also tests efficiently, during a software project is essential to avoid bugs and to keep up with changing requirements [1]. However, achieving this goal is getting more and more difficult the larger and older a project gets, because we as humans can only deal with a reasonable amount of complexity or change [22]. This pushes software developers to keep test-suites compact. Testing itself comes with a hurdle. It cannot show, that software is defect-free. It can only show the presence of failures [5]. But the more developers test their software, the more confidence in their software they have. This leaves software developers in a dilemma. On the one side they should test as much as possible to make sure, that their software is working as expected. On the other side they have to keep the test-suite compact. Otherwise, the software project may get unmaintainable. Additionally, they can never be absolutely sure, that their software is absolute defect-free.

Methodologies can be a guide out of this dilemma. One known and applied for decades is code coverage [30]. Here one measures how much of the production code has been covered by a test-suite. There are several variations of coverage criteria. Most notably are statement and branch coverage, where someone measures how many statements, or branches respectively, have been covered [47]. These metrics are easy to compute, they can be computed alongside a single execution of the test-suite. But in terms of test-suite quality, coverages are a rather weak criteria. They do not have any fault revealing ability [14]. Therefore, they do not reveal if the code is tested enough.

Mutation Testing, another method for evaluating test-suite quality, has shown that it is a better method to evaluate test-suites compared to coverage criteria [14, 28]. During mutation testing artificial defects get inserted into production code. The more defects a test-suite detects, the better the test-suite quality is. Mutation testing requires a lot more time to execute than coverage analysis. For each mutation the whole test-suite has to be run to evaluate if it can detect the mutation.

Consider the example shown in algorithm 1.1. To fulfill 100% statement coverage one test with $x = 2$ is enough. Two test cases with $x = 2$ and $x = 0$ are enough to fulfill 100% branch coverage. This shows, that, at least in this case, branch coverage is a stronger criterion for test-suite evaluation. Let's take the two test cases from branch coverage and use those for mutation testing. A *Conditionals Boundary Mutator* ([97]) generates one mutant for this program. The mutant is changed in line 2. $x > 1$ is replaced by $x >= 1$. Mutation testing will execute the test-suite twice, one time for original program and a second time for the mutant. The two test cases from branch coverage succeed for the original program, but they cannot identify the mutation. This shows that mutation testing is an even stronger criterion than branch coverage. But to get this result, twice the runtime was required.

Algorithm 1.1: Example program

```
   Input: An integer x
   Output: True iff x is greater than 1
1  isBigger ← false;
2  if x > 1 then
3  |  isBigger ← true;
4  return isBigger;
```

There have been a lot attempts to minimize the amount of mutants and therefore decrease the runtime for mutation testing, but even with a reduced set the test-suite has to be run several times, keeping it always more expensive than coverage analysis or other test-suite evaluation methods. There has been a lot of research in the last ten years, but software developers haven't adapted the practice of mutation testing yet [52]

Even with this disadvantage, mutation testing is the best method to evaluate test-suites and therefore is often mentioned as a "gold standard" for experimental evaluations of test methods [43]. Additionally, it has been shown that developers working on projects with mutation testing write more tests on average over longer periods of time, compared to projects that only consider code coverage [57].

1.1 Problem Statement

Finding a similar good method to mutation testing, but less computationally expensive, would be important in making test-suite evaluation more applicable. And with already a bunch of faster but less accurate methods of test-suite evaluation out there, the question arises if those can be used to find a method similar to mutation testing. So, is there a way to get a quality model based on existing test-suite evaluation methods, that is comparable to mutation testing with respect to its defect detection capabilities, but easier to compute?

In order to investigate this idea, this thesis deals with the following questions:

RQ1 What are existing metrics for test-suite evaluation?

RQ2 How can existing metrics be combined to get a cheap quality model similar to mutation score?

To answer these questions, a framework is proposed in this thesis. Implementing the framework contains two parts, each of them tackles one research question. The first part of the framework contains a systematic literature review to find existing metrics. In the second part the found metrics are used to build a quality model built upon the found metrics. The quality model uses machine learning to accomplish this.

1.2 Contribution

In this thesis a framework is developed. The framework is a novel quality model for test-suite evaluation, which should be as similar as possible to mutation testing in respect to test-suite evaluation, but is significantly better in terms of runtime. The framework is developed based on Design Science Methodology as proposed by Wieringa [74] for Information Systems and Software Engineering. More precisely, the following four steps are taken:

1. Define the framework architecture
2. Finding existing metrics for test-suite evaluation
3. Build a novel quality model for test-suite evaluation
4. Evaluate the novel quality model for test-suite evaluation

In Step 1 the requirements for the framework are identified, based on the research questions. An architecture to satisfy these requirements is developed. The implementation of the framework is performed in two parts. In the first part existing methods for test-suite evaluation are collected. In the second part the found test-suite evaluation methods are combined to finish the new framework. Each part is covered in step 2 and step 3 respectively.

Step 2 contains a systematic mapping study as proposed by Petersen et al. [56]. This process leads to a list of test-suite evaluation methods that can be applied to existing software projects. The list is ordered by frequency of appearances in literature.

In Step 3 the novel quality model is built using different machine learning models. To develop the machine learning models open source software projects are collected and adapted, so that each project produces the test-suite evaluation methods found in the previous step. Additionally, mutation testing is applied to each project. This provides a data basis for supervised machine learning. The output is a regression model, that takes several test-suite evaluations and predicts the mutation-score from them.

The evaluation of the new model is done in Step 4. To do so two experiments are conducted. The experiments have the structure as proposed by Wohlin et al. [75]. In the first experiment existing methods for test-suite evaluation are compared to mutation testing to establish a baseline. The second experiment investigates how the novel quality model compares to mutation testing. Comparing the results from the first experiment with the second one gives insights if the novel quality model improves upon existing test-suite evaluation methods.

1.3 Structure of the Thesis

The remainder of this thesis is structured as following. Chapter 2 provides some background information about the theory and techniques used in this thesis. The novel quality model is described in Chapter 3. In Chapter 4 a systematic mapping study is performed to answer *RQ1* and Chapter 5 evaluates the novel quality model to answer *RQ2*. Chapter 6 presents related work and Chapter 7 concludes this thesis.

CHAPTER 2

Background

This chapter provides some background information, on which this thesis relies on. We dive deeper into software testing, software quality, test-suite quality and machine learning.

2.1 Software Testing

When talking about software testing we should first think about what it exactly is. Graham et al. [22] dwelled deeper into the topic of defining software testing. They took the International Software Testing Qualifications Board (ISTQB) definition [95] and split it up into two parts. The first part looks at testing from a process view. The second part looks at some goals testing has.

Graham et al. [22] mentioned the following properties for the process view:

- *Process*: Testing is a process rather than a single activity. There are a series of activities involved.
- *All life cycle activities*: Testing happens during all the software development lifecycle. Not only the finished software product needs to be tested, every document produced during the development process can be tested as well.
- *Both static and dynamic*: Testing doesn't require, that a piece of code has to be executed. Testing can happen statically as well, for example by looking at code or requirements.
- *Planning*: Testing requires planning. There are things to do before and after test execution. The plan should include, but not limited to, documentation of what the test should check and definitions of how the test results are reported.

- *Preparation*: Testing needs test cases, that can be executed. These need to be prepared before the test execution.
- *Evaluation*: During test execution, the tester must check the test results and determine if the software under test has met the completion criteria, or if more testing is required.
- *Software products and related work products*: As mentioned before, testing doesn't only include the software product. The requirements and design specification need to be tested too. Static and dynamic testing are required to determine if a software product is fit for delivery.

Testing has the following characteristics or goals as mentioned by Graham et al. [22] and Ammann et al. [5]:

- *Verification*: Check whether a software product satisfies the specified requirements.
- *Validation* Evaluate if the software product is fit for purpose. Or in other words, does the software product solve the problem the user/customer wants to solve.
- *Detect Defects*: Check whether the software product contains other unexpected behavior, that may not be covered by any requirement. These are called implicit requirements.

If software testing isn't successful, then a bug has been identified. A bugs lifecycle has three phases [22]:

1. *Mistake/Error*: The origin of every bug is mistake or error a developer makes. This error may occur during coding, but it can occur much earlier in the software development phase. Errors can happen during writing of requirements too.
2. *Fault*: A fault is the manifestation of a mistake. It's, for example, the wrongly written requirement or the software, which has been developed incorrectly.
3. *Failure*: A fault leads to failure during runtime. This may manifest itself as an error a user encounters, or it may be part of the software which does not behave as expected.

We already mentioned that testing is part of any software development phase and that it not only checks whether the requirements are fulfilled, but also checks if the users' needs are met. This leads to different kinds of testing. Most prominently there are the four following kinds of testing [22, 5]:

- *Component/Unit Testing*: When doing unit testing, one tests small independent parts of a software. Such parts may be modules, programs, objects, classes etc.

- *Integration Testing*: Integration testing checks if different parts of a software product work correctly together. These tests may include the operating system, file system and hardware or interfaces between systems.
- *System Testing*: During system testing a tester is concerned with the behavior of the whole system product. It should work as defined by the requirements. The main focus here is verification of the system.
- *Acceptance Testing*: The main purpose here isn't verification, but validation of the whole product. The tester checks if the product meets the requirements, the user's needs and the business processes. Acceptance testing is often the last step before a system is delivered to a user/customer.

When testing a piece of software a tester must be clear about the objective of his endeavor. Broadly, there are four different test types [22]:

- *Functional testing*: Here testers have full focus on what the software does. Mostly the expected behavior can be derived from the requirements. Functional testing is also called black-box testing.
- *Non-functional testing*: All software contains some non-functional requirements. For example, the software should be "fast", or it should be "usable". Such qualities are tested here.
- *Structural testing*: Structural testing looks the different parts or code of a software product. It is most often used as a way of measuring the thoroughness of testing. A goal here could be to measure how much of the code is tested (coverage). Structural testing is also called white-box testing.
- *Confirmation and regression testing*: The final target of testing is the testing of changes. Confirmation testing checks if a previously detected bug has been fixed and regression testing checks is previously passed test cases still succeed.

2.2 Software Quality

Software Quality, or Quality on its own, is an ambiguous term ([34]) and has been interpreted differently over the years. [71] Still to this day, there is no single definition of quality. The ISO, IEC and IEEE [94] has collected six different ways to define quality, respectively software quality:

1. Degree to which the system satisfies the stated and implied needs of its various stakeholders, and thus provides value
2. Ability of a product, service, system, component, or process to meet customer or user needs, expectations, or requirements

3. The degree to which a set of inherent characteristics fulfills requirements
4. Capability of software product to satisfy stated and implied needs when used under specified conditions
5. Degree to which a software product satisfies stated and implied needs when used under specified conditions
6. Degree to which a software product meets established requirements

Although there are many definitions, most of them have something in common. Many talk about requirements and that those requirements should be fulfilled. Furthermore, the mention that quality can be measured on a system, component, process, product or service. [71]

The requirements to be satisfied can come from the user or the customer. So it is not clear if high quality means we satisfy what the person using the system or what the person paying for it wants it to do. We can't even assume, that all requirements are explicitly stated. So high quality may be very hard to achieve. It may be only be possible to determine if a software product has high quality after the product has been delivered to the user. [71]

There are two ways to think about software quality. Those are in-process quality, which focuses on the process that produces a high quality product, and product quality, which looks on the product itself in terms of quality.

2.2.1 In-Process Quality

In-process quality works on the assumption that good processes lead to good products. Several quality standards are solely about the processes. A widely used standard with this manufacturing and process view is ISO 9000 [92]. It contains the idea of establishing a quality management system in a company so that the resulting quality of the product will also be high. It does not concern itself with the quality of the products. [71]

This approach does have some problems. One is that this process orientation leads to quality assessment that is virtually independent of the product itself. There isn't any proof, that good processes lead necessarily to high quality products. [71]

Kan [34] identified some metrics, which can be used for in-process quality. Those are:

- *Defect Density During Machine Testing:* Here the defects during testing are collected and presented as some density metric. A specific example for this could be defects per KLOC.
- *Defect Arrival Pattern During Machine Testing:* During testing not only the amount of defects is important, but also how the discovery of defects changes over time. A specific example here could be defects per KLOC per week.

- *Phase-Based Defect Removal Pattern:* This is an extension of the test defect density metric. Here not only the defects during testing are counted, but throughout the whole development process, including the design reviews, code inspections, and formal verifications before testing. The defect counts are then grouped per development phase.
- *Defect Removal Effectiveness:* The defect removal effectiveness is an estimation on how many defects are removed during testing. This is calculated by dividing the number of defects found during testing through the number of defects found during and after testing.

2.2.2 Product Quality

Contrary to in-process quality, product quality looks at the product itself and tries to measure the quality there. There are several ways to look at product quality, one of them is user expectation. So product quality can be thought of as understanding the user expectations, translating them into clear product attributes and ensure that these attributes are then implemented in the product. [71]

ISO/IEC 25010 [93] splits up product quality several quality factors. The standard identified the following factors:

- *Functional suitability*: degree to which a product or system provides functions that meet stated and implied needs when used under specified conditions
- *Reliability*: degree to which a system, product or component performs specified functions under specified conditions for a specified period of time
- *Performance efficiency*: performance relative to the amount of resources used under stated conditions
- *Usability*: degree to which a product or system can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use
- *Security*: degree to which a product or system protects information and data so that persons or other products or systems have the degree of data access appropriate to their types and levels of authorization
- *Maintainability*: degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers
- *Portability*: degree of effectiveness and efficiency with which a system, product or component can be transferred from one hardware, software or other operational or usage environment to another

Kan [34] identified some metrics for product quality too. Those are:

- *Defect Density:* Defect density can also be seen as a product quality metric. The general concept of defect rate here is the number of defects over the opportunities for error during a specific time frame.
- *Customer Problems Metric:* Another product quality metric used in the software industry measures the problems customers encounter when using the product. This includes all problems, not only those which were identified as defects, as any problem the user encounters matters to them.
- *Customer Satisfaction:* Customer satisfaction is often measured by customer survey data via the five-point scale: Very satisfied, Satisfied, Neutral, Dissatisfied, Very dissatisfied. This is easy to survey but gives important feedback about the software product.

2.2.3 Quality Models

We have already discussed quality factors of the ISO/IEC 25010 [93]. In general there are models, which try to split up quality in several factors, ISO/IEC 25010 being one of them. In literature, these models are called quality models.

Wagner [71] proposed several definitions in the context of quality models, to give a better idea what quality models are.

1. **Quality model:** A model with the objective to describe, assess and/or predict quality. Many quality models describe a decomposition of the general product quality into sub-qualities to make them better understandable and manageable.
2. **Quality Factor:** A management-oriented attribute of software that contributes to its quality. In the context of quality models, we are also confronted with the terms quality requirements and quality goals describing demands to the quality of a software system. We distinguish them by seeing quality goals as more abstract while quality requirements should be concrete and measurable.
3. **Quality Goal:** An abstract demand onto a quality factor of a software product.
4. **Quality Requirement:** A concrete and measurable demand on a specific product factor that has an impact onto a quality goal or quality factor of a software product.

Quality models have been a research topic for several decades and many of them have been proposed [71]. The first quality models were published in 1978 by Boehm et al. [10] and McCall et al. [45]. The two models are similar. They deconstruct quality into quality factors such as maintainability or reliability.

In the 1990s, researchers have been proposing more elaborate ways of decomposing quality characteristics. The quality models got thereby better. Implicitly they introduced quality model metamodels [71]. Contemporary examples for quality metamodels are ISO/IEC 25010 [93] and Quamoco Quality Models [72].

2.3 Test-Suite Quality

Testing can give confidence in the quality of software [22]. A high quality test-suite can provide this confidence. Specific metrics are available that help quantify the quality of a given test-suite. However, measuring the quality is still a challenge. [26]

If a tester could test everything (all combinations of inputs and preconditions), the confidence would be the highest. But this is not feasible except for trivial cases. Take, for example, a simple program with two input variables, both variables representing a number with 8 bit. Testing everything would require writing test cases for each possible combination of inputs. In our example this would mean that we need 2^{16} test cases. We need risks and priorities to focus testing efforts. [22]

The commonly used type of test adequacy criterion is called code coverage [26]. Code coverage measures the percentage of code, which was executed by test cases. There are many ways to measure code coverage, e.g. statement coverage and branch coverage. In statement coverage the percentage of statements of a program is measured, and in branch coverage the amount of branches is measured [26]. Code coverage can be used to quickly check what parts of the code are executed by the test-suite. However, code coverage alone is not very efficient in determining the quality of a test-suite, and therefore the confidence in this metric is low [28, 26].

2.3.1 Mutation Testing

Besides code coverage, mutation testing has emerged as another way to measure test-suite quality. Mutation testing is considered a better way to measure test-suite quality, as it's fault revealing capability is higher [14]. Mutation testing was first described in 1971 by Lipton [41].

During mutation testing several program variations, so-called mutants, are generated. These mutations are rather simple and are created by mutant operators. A simple example of such mutations can be demonstrated with the expression $x = a + b$. A simple mutant operator could change this expression to $x = a - b$. [15]

For each mutant the test-suite is run. If at least one test case fails because of the mutant, we say the mutant is killed. If all test cases still succeed, we say the mutant is alive. In this case tests can be modified, or new test cases can be added, so that the mutant gets killed. [15]

Mutation testing produces a test adequacy criteria called mutation score. This can be calculated as described in Equation 2.1. The mutation score MS depends on a program P and a test-suite T testing P. A higher mutation score indicates a test-suite with higher quality. Or in other words, the test-suite is better in revealing faults. [15]

$$MS(P,T) = \frac{|killed\ mutants|}{|all\ mutants|} \tag{2.1}$$

Lin et al. [40] lists several conditions for a test-suite execution on a mutant:

1. A test must reach the mutated statement.
2. Test input data should infect the program state by causing different program states for the mutant and the original program.
3. The incorrect program state must propagate to the program's output and be checked by the test.

If the first two conditions hold, we call the mutation analysis level weak. If all conditions hold we call it strong mutation testing. Strong mutation testing has a stronger fault revealing capability than weak mutation testing, because it requires that the test-suite can identify the fault. Weak mutation testing is more similar to code coverage methods.

The results of mutation analysis depend heavily on the mutant operators used [15]. There is no distinct list of mutant operators, as the possible operators depend on the used programming language. But Gutiérrez-Madronal et al. [24] introduced a classification of operators:

- *Traditional mutation operators:* They could be applied to any programming language no matter its nature.
- *Nature mutation operators:* They just could be applied according to the nature of the programming language. These operators are defined according to syntactic-like language faults.
- *Specific mutation operators:* They can not be applied to any other programming language. These operators are defined according to non syntactic-like language faults. That means that the mutations are done in the part of the language which differs from the rest of the languages.

Papadakis et al. [52] dwells deeper into the advantages of mutation testing. They emphasize, that mutation testing is unique as it provides a mechanism by which assertions concerning test effectiveness become falsifiable. Failure to detect certain kinds of mutants suggest failure to detect certain kinds of faults. Mutation testing is the best method to evaluate test-suites, and therefore is often mentioned as a "gold standard" for experimental evaluations of test methods [43].

Despite its usefulness, mutation testing has some disadvantages. The following list describes the most mentioned ones in literature:

- *Equivalent mutant problem:* Mutation testing relies on the assumption, that each mutated program is functionally different from the original program. But this is not always the case. Equivalent mutants can never be killed and therefore distort the mutation testing results. Detecting such mutants is still a problem, that remains to be solved. [52, 51]

- *Redundant mutant problem:* Mutants cannot only be equivalent to the original problem, they may also be functionally equivalent to each other. Redundant mutants distort the mutation testing results too, as they always get killed together. Detecting redundant mutants is still a problem, that remains to be solved. [52]

- *Large number of mutants:* Programs have many possible places where faults can be injected. This means that are also many possible mutants that can be generated, even for simple programs. Consider the program return a+b, where a and b are integers. Even for this one line, there are many possible mutants, e.g. a-b, a*b, a/b, a+b++, -a+b, a+-b, 0+b, a+0. The large number of mutants lead to very high execution costs, because for each mutant the whole test-suite must be executed. Consider a program with 150 mutants and 200 test cases. It requires (1+150)*200 = 30200 executions with their corresponding results. [51, 78]

This section describes two options to evaluate test-suite quality. But each comes with its downsides. Code coverage is fast to compute, but the results don't say much about the test-suite quality [14]. Mutation testing, on the other side, gives meaningful information about the quality of a test-suite [14], but to perform mutation testing is computationally expensive [51, 78]. To the knowledge of the author of this thesis a test-suite evaluation metric, that combines the benefits of code coverage and mutation testing without the downsides does not exist.

2.4 Machine Learning

Machine learning is a very broad field and has been applied to many fields, for example predictive maintenance [13], self-driving cars [16] or credit card fraud detection [8]. Also, software testing has seen been improved through machine learning. Machine learning was used for test case generation, refinement, and evaluation. Also, machine learning has been used to evaluate test oracle construction and to predict the cost of testing-related activities [19]. The term machine learning was first coined by Samuel [62] in 1959. It is a subfield of artificial intelligence [4].

In machine learning the programmer doesn't implement an algorithm to solve a problem, but the machine learns itself to solve a problem by previous examples. For this the computer has to reason itself about existing data. This data is called training data. The process of solving the real problems after learning the training examples is called generalization [32]. Machine learning has many applications, such as spam filtering, optical character recognition and detection of malicious behavior [32].

Machine learning itself can be split up into four different types [32]:

- *Supervised Learning:* A supervised learning algorithm uses known input-out pairs to train to get the desired output. The output of the training model is an inferred model, which can be used on new data.

- *Unsupervised Learning:* Unsupervised learning algorithms don't have input-out pairs available. Here the training data is not labeled, which means the output information is not given. Cluster analysis is the most common unsupervised learning application.

- *Semi-supervised Learning:* Semi-supervised learning algorithms uses a mixture of labeled and unlabeled data sets. The output models have learned the structure of the data as well can make predictions. Regressions and classifications are typical problems solved through semi-supervised machine learning.

- *Reinforcement learning:* Reinforcement learning is different from supervised and unsupervised learning as it doesn't rely on a predefined training data. It rather improves itself by exploration of a problem. It is used in many disciplines such as game theory, simulation-optimization or statistics.

The main focus in machine learning is on the training data. Reinforcement learning is the exception here as it doesn't require any training data. It is very important that the training data is unbiased, otherwise the produced model may not generalize [79]. To evaluate how much bias a model has, is to hold out some data from the data. In other words we split the training data into a training set and a test set [79]. The test set is used to evaluate how well the model performs. The evaluation gains more meaning if the test set is independent of the training set. To make an assumption about generalization it is important, that the two sets do not overlap. Ideally the test set and the training set don't even contain similar samples [79]. A possible split of the training data is to use 80% for the training and 20% for the test set. Another option is cross validation. In this case the data is not split into two sets, but into N sets. N-1 parts are used for training and one part is used for test. This process is repeated N times until all parts have been used for testing. N is often set to 10, this is called 10-fold cross validation [4].

A problem, which may occur during machine learning, is overfitting [79]. It occurs when a model performs much better on the training set, than on the test set. In this case the model does not generalize and is unable to handle new data. The reason is usually that the model has to many parameters that are over-optimized to predict the training samples.

We have already mentioned some machine learning applications. Broadly the applications can be split into four different groups [32]:

- *Classification*: Classification is defined as assigned one or some predefined categories to an item. Binary classification is the simplest form, which can be extended to multiple classification. Multiple classification assigns an item to one of three or more classes.

- *Regression*: Different to classification, regression does not assign a value from a predefined category, but it assigns a continuous value. There are two types

of regression. In univariate regression only one output value is estimated. In multivariate regression more than one output value are predicted.

- *Clustering*: Clustering is the process of segmenting a group of items into subgroups, whereby each group should contain similar items. Other than classification and regression, clustering is a classic unsupervised machine learning application. An integral part of clustering is the similarity function. It determines how the data is split up. Clustering may be used to automatically label data, which itself can be used as training data for classification or regression.

- *Hybrid Tasks*: If any of the previous applications are combined, then we talk about hybrid tasks. In the section about clustering we already mentioned, that the clustering result may be the input for some classification or regression. Such a scenario is a hybrid task. Another classic example is the dynamic document organization, where text classification and text clustering are combined.

CHAPTER 3

Novel Test-Suite Evaluation Framework

This chapter proposes a novel quality model framework, which uses existing test-suite evaluation metrics to predict mutation scores. Section 3.1 uses the research questions defined in Chapter 1 to derive some requirements for this framework. The architecture of the framework is described in Section 3.2 and Section 3.3 provides some insights into the implementation process.

3.1 Requirements

The research questions proposed in Section 1.1 imply requirements on the framework. Besides that, there are some implicit requirements that ensure that the framework will be applicable in an industrial setting.

Each requirement contains a goal, which can be derived directly from the corresponding research question. If the research questions are fulfilled, then the goals are fulfilled too. Each goal can be refined into a specification. How a specification is fulfilled is explained in the realization section.

The following requirements have been identified:

3.1.1 Requirement 1: Efficiency

Goal

From *RQ2* we can imply that the new model should efficient. Efficient in this context means, that the new model should use significantly less runtime to calculate the result, than mutation testing.

Specification

Each calculation of a test-suite evaluation metric must be done during a single test-suite execution. Additionally, the use of the combination algorithm cannot be computationally expensive and put the overall cost over mutation testing. In the end, the overall framework must be faster than mutation testing.

Realization

To realize the first part of the specification, only test-suite evaluation metrics are taken into account, that can be calculated in a single execution of the test-suite of a software project. For the second part of the specification only established machine learning algorithms are used, which should not add much complexity to the runtime behavior of the framework.

3.1.2 Requirement 2: Effectiveness

Goal

RQ2 does state, that its resulting evaluation score is as similar as possible to mutation score. Thus, the new evaluation score does not have to be more effective than mutation testing. Effective in this context means, that the new models predicts the mutation score for a given project as precise as possible.

Specification

The evaluation score resulting from the novel framework should predict the result of mutation testing as close as possible.

Realization

To realize the specification supervised machine learning algorithms are used, that are trained on a data set containing the test-suite evaluation metrics described in Chapter 4 as well as the corresponding mutation testing results. The model, which fits mutation testing best, is chosen.

3.1.3 Requirement 3: Automation Potential

Goal

To make sure that the new framework can be used in an industrial setting, it is important, that it can be integrated into software-build pipeline. In other words it is important that the new evaluation score can be calculated without any human intervention

Specification

First each used test-suite evaluation method can be calculated without any human intervention, and it can be integrated into an automated build process. Second the new combination algorithm itself does not require any human intervention, and it can be integrated into an automated software build process.

Realization

For the realization only test-suite evaluation methods and combination algorithm which fulfill requirements 1 and 2 are chosen.

3.2 Architecture

Figure 3.1 shows the architecture of the novel quality model framework developed in this thesis. It uses existing test-suite evaluation metrics to predict mutation scores. As entry point for the framework a software project must be given, on which evaluation scores can be calculated. During the build and execution of the corresponding test-suite several evaluation engines get hooked into the build process. Each engine can produce one or more metrics. After all evaluation engines have done their job all the calculated metrics are passed to a combination algorithm. The result of which is an evaluation score.

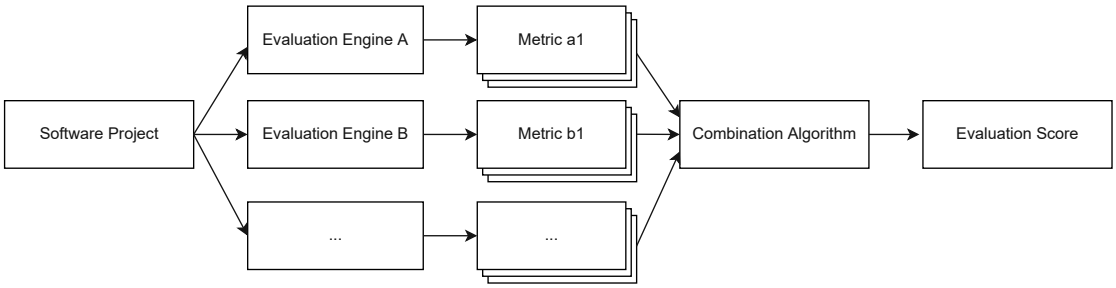

Figure 3.1: Framework Architecture.

There are several components in this overview, which are yet undefined. E.g. it is unclear which metrics should be used for the combination algorithm and neither is which evaluation engines are necessary to produce those metrics. On the other hand, the combination algorithm itself is still a black box. There is no further information on how this algorithm should work. The next two chapters of this thesis fill out these unknowns.

Chapter 4 evaluates which test-suite evaluation metrics already exist in literature. So in the end of this chapter a list of possible metrics is provided.

The subsequent chapter, Chapter 5, uses these metrics, selects some corresponding evaluation engines, combined with several machine learning regression models to build instantiations of the framework. These variations are evaluated, and the best is presented.

3.3 Implementation

In this thesis the framework proposed in Section 3.2 is implemented. Although not a fully integratable prototype is created, the necessary parts to evaluate the approach are evaluated.

An overview of the implemented parts can be seen in Figure 3.2. The implementation works under the assumption, that all evaluation engines produces a report of their collected metrics and store them in the build directory.

collect-metrics takes these reports after all engines have finished and produces an aggregated report with the metrics of all evaluation engines. Depending on the type of metric, some data processing might be necessary, e.g. if the metric only counts some occurrence from some phenomena, then it might be necessary to normalize the metric. Depending, if *collect-metrics* provides the aggregation for *analyze-metrics*, a mutation testing report is also included in the output.

The main purpose of *analyze-metrics* is to create a machine learning model, which takes the data from *collect-metrics* and calculates a new evaluation score. The new evaluation score should be as similar as possible to mutation score, therefore it needs the mutation testing results too. To build the machine learning model few data sets are not enough, but as many as possible aggregated metrics from different software projects are needed.

The novel quality model uses the aggregation without mutation testing to predict the new evaluation score.

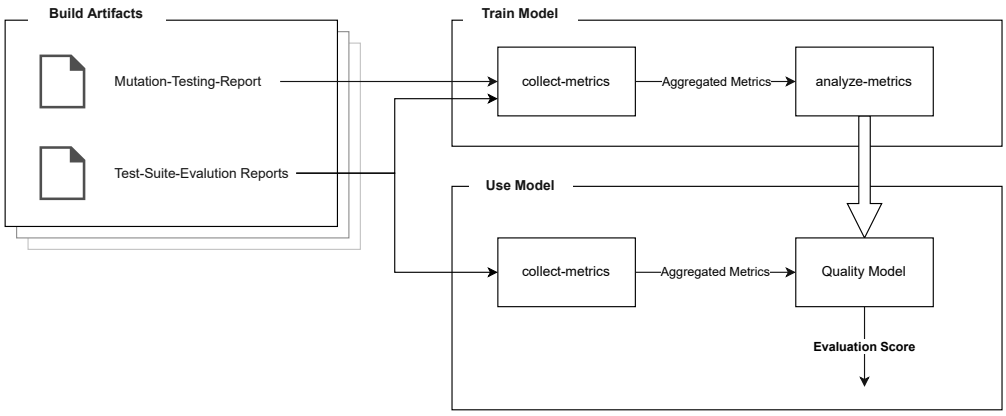

Figure 3.2: Implementation Overview.

The *collect-metrics* application is written with Java 11[1] and bundled into a Maven-Plugin[2]. So the current implementation is restricted to software projects written in Java and using

[1] https://www.java.com
[2] https://maven.apache.org

Maven. But there can easily be implementations for other programming languages and build systems. You can find the source code to *collect-metrics* online[3].

analyze-metrics is written in Python[4] using a Jupyter Notebook[5]. sklearn [53] is used as machine learning engine. You can find the source to *analyze-metrics* online[6].

sklearn is the most comprehensive and open-sourced machine learning package in Python. It is chosen for this thesis as it includes implementations of a comprehensive list of machine learning methods under unified data and modeling procedure. This allows for effortless switching of one machine-learning method to another, which is beneficial for this thesis. [25]

During training for the novel quality model we may encounter test data, which is low in variance. This may lead to overfitting. To circumvent this, features with low variance are filtered out. This is implemented with the VarianceThreshold by sklearn. VarianceThreshold is a simple baseline approach to feature selection. It removes all features whose variance doesn't meet some threshold. More precise it removes all features with a lower variance than configured. In this thesis a variance threshold of 0.05^2 is used.

3.4 Combination Algorithms

As mentioned before, *analyze-metrics* uses several machine learning algorithms to create a novel quality model. The following sections describe the algorithms used in this thesis. All the algorithms listed are supervised machine learning algorithms, as the data used for training contains the input values and the corresponding output value. More precisely they are regression algorithms, as the required output should be a real number in the range from 0 to 1. This list of algorithms is chosen, because they are supported by sklearn [53]. Consequently, the sklearn [53] implementations are used.

3.4.1 Linear Regression

Linear Regression is the perhaps one of the most common and comprehensive statistical and machine learning algorithms [44]. In this thesis the implementation by sklearn is used which is an ordinary least squares linear regression.

The implementation fits a linear model with coefficients $w = (w_1, ..., w_p)$ to minimize the residual sum of squares between the observed targets in the data set, and the targets predicted by the linear approximation [79]. It tries to solve Equation 3.1 [96].

$$\min_w ||Xw - y||^2 \tag{3.1}$$

[3] https://github.com/steeeve1510-master-thesis/collect-metrics
[4] https://www.python.org
[5] https://jupyter.org
[6] https://github.com/steeeve1510-master-thesis/analyze-metrics

3.4.2 Ridge regression

Ridge regression is similar to linear regression, but it addresses some problems of ordinary least squares by imposing a penalty on the size of the coefficients [27]. It tries to solve the Equation 3.2 [96].

$$\min_w(||y - Xw||^2 + \alpha * ||w||^2) \tag{3.2}$$

The hyperparameter α controls the penalty term, and therefore controls the amount of shrinkage. The larger the value of α, the greater the amount of shrinkage and thus the coefficients become more robust to collinearity [27]. In this thesis α is set to the sklearn default value, which is 1 [96].

3.4.3 Lasso

Lasso is a linear model which uses shrinkage, similar to Ridge regression. But here the penalty term uses the absolute value, and not the square. It is useful in some contexts due to its tendency to prefer solutions with fewer non-zero coefficients, effectively reducing the number of features upon which the given solution is dependent. [70]

The sklearn implementation of Lasso solves Equation 3.3 [96].

$$\min_w(\frac{1}{2n_{samples}}||y - Xw||^2 + \alpha * ||w||) \tag{3.3}$$

As mentioned before, we can see, that the equation looks similar to Ridge regression, but the penalty term uses the absolute value. α is a hyperparameter, which allows to change the weight of the penalty term. The sklearn default for *alpha* is 1 and is used in this thesis [96].

3.4.4 Elastic Net

Elastic Net is a hybrid of Lasso and Ridge regression, where both the absolute, and the squared penalty term are used. This combination allows for learning a sparse model where few of the weights are non-zero like Lasso, while still maintaining the regularization properties of Ridge. [81]

A practical advantage of trading-off between Lasso and Ridge is that it allows Elastic net to inherit some of Ridge's stability under rotation. [81]

The sklearn implementation tries to minimize objective function defined in Equation 3.4.

$$\min_w(\frac{1}{2n_{samples}}||Xw - y||^2 + \alpha\rho||w|| + \frac{\alpha(1-\rho)}{2}||w||^2) \tag{3.4}$$

Again the default values for α and ρ are used, which are 1 and 0.5. [96].

3.4.5 Random Forest

A random forest is an ensemble algorithm. It combines several decision trees. It works on the assumption, that on average the combined decision trees perform better than a single one and the combined model isn't as much prone to overfitting than a single tree. Each tree isn't trained with the whole data set. Each tree only is trained with a sub set, which is unique for the tree. Again, the sklearn implementation is used here. In contrast to the original publication [12], the sklearn implementation combines classifiers by averaging their probabilistic prediction, instead of letting each classifier vote for a single class.

3.4.6 Support Vector Machine

Support Vector Machine (SVM) is a machine learning algorithm, which has mainly been used for classification, but can also be used for regression. A support vector is a data point, which is nearest to a hyperplane. If removed the hyperplane would change. A SVMs job is to find an optimal hyperplane, such that the margin between the classes, the hyperplane separates, is maximal. That hyperplane can then be used to determine the most probable label for unseen data. The features used to infer the hyperplane are not typically raw data. Rather, they are most often derivative data resulting from some kind of interpolation during the feature selection stage. [59]

CHAPTER 4

Finding Metrics for Test-Suite Evaluation

In a first step we look at existing methods for evaluating tests suites. We perform a mapping study to search for literature as proposed by Petersen et al. [56].

In Section 4.1 introduces mapping studies as proposed by Petersen et al. [56]. In the subsequent Section 4.2 the knowledge questions are defined and the mapping study for this thesis is planned. Section 4.3 presents the results and in Section 4.4 the results are taken and used to answer the knowledge questions defined in Section 4.2. Finally, *RQ1* is answered.

4.1 Mapping Studies

It becomes more and more difficult to grasp the current state of the art as literature in specific research areas grows. Methodologies that generate an overview are key to solve this problem. One such methodology is the systematic mapping process [56]. We will call the process simply mapping study.

In broad terms mapping studies search the available literature by terms and then categorizes the found publications by keywords. Petersen et al. [56] describes the five main steps for performing a mapping study.

4.1.1 Definition of Research Questions

In the beginning of each mapping study researchers has to decide what they want to find and consequent define appropriate research questions, which the study should answer. When defining research questions you have to consider that mapping studies cannot answer all kinds of questions. The main focus of them is to provide an overview of a

research area. But they can also identify the frequencies of publications over time or identify the venues in which research has been published [56].

A number of typical research goals for mapping studies were identified by Arksey et al. [6], namely:

- To examine the extent, range and nature of research activity
- To determine the value of undertaking a full systematic review
- To summarize and disseminate research finding
- To identify research gaps in the existing literature

4.1.2 Conduct Search for Primary Studies

The next step after defining the research questions is to search for some primary studies. There are several search strategies how primary studies can be found. Most notably there is database search, manual search and snowballing. For database search a search string is used to query scientific databases, relevant conference proceedings or journal publications. Doing manual search is fairly similar to database search, the only difference being that the search is done manually, e.g. look through relevant journals or conferences papers relevant to the research topic. Snowballing is different. Here one takes a set of initial publications and gathers all references mentioned in the initial set. This is called backward snowballing. Additionally, one looks for publications which references one of the initial set. This is called forward snowballing. Often multiple search strategies are combined in one study. For example use database search to get a set of publications and then apply snowballing them.

When using database search or manual search one has to define which databases are of interest for the study. You can find those by talking to experts and gather the most essential databases for a given research topic. In context of software engineering it is sufficient to use IEEE and ACM as well as two indexing databases (e.g. Inspec/Compendex and Scopus) [20, 37].

Database search and, to some extent, manual search require search terms. One good method to create search terms is by using the four groups population, intervention, comparison, and outcome (PICO) developed by Keele et al. [36].

- *Population:* In software engineering, population may refer to specific software engineering role, category of software engineer, an application area or an industry group.
- *Intervention:* In software engineering, intervention refers to a software methodology, tool, technology, or procedure.

- *Comparison:* This is the software engineering methodology/tool/technology/procedure with which the intervention is being compared.
- *Outcome:* Outcomes should relate to factors of importance to practitioners such as improved reliability, reduced production costs, and reduced time to market.

You can build the full search term by taking each term identified in a group and join them with *OR*. You will get a complex term for each group. Finally, combine these four terms with an *AND* to get the full search term.

The outcome of the primary studies should not be too few nor too many publications. Wohlin et al. [76] state that the amount of papers is sufficient if they are a good representation of the population.

4.1.3 Screening of Papers for Inclusion and Exclusion

The initial set of publications will most likely contain many papers which are not relevant for answering the research questions. Here inclusion and exclusion criteria come into play. Inclusion and exclusion criteria are used to filter out studies which are not relevant [56].

There are many attributes the criteria may refer to. These may relate to the relevance of the topic of the article, the venue of publication, the time period considered, requirements on evaluation and restrictions with respect to language [55].

For example, Petersen et al. [55] mentioned the study by Laguna et al. [39]. They conducted a mapping of literature related to product line evolution. They defined the following inclusion criteria:

- *"English peer-reviewed articles in conferences or journals published until Dec. 2011"*
- *"Articles that focus on software product lines"*
- *"Articles that provide some type of evolution of existing software artifacts, included among the terms selected"*
- *"Context, objectives, and research method are reasonably present"*

Laguna et al. [39] defined exclusion criteria too. These are:

- *"Articles that are not related to software product lines"*
- *"Articles that do not imply evolution of any software artifacts"*
- *"Non-peer reviewed publications"*
- *"Articles that are not written in English"*
- *"Context, objectives, or research method are manifestly missing"*

4.1.4 Evaluate the Search

Having a set of relevant publications, one should evaluate if the set does not miss any important ones. There are several ways to do this, Petersen et al. [55] proposed two.

The first one utilizes a test-set of known papers. This set of known papers can be compiled using scholars, which are already familiar with the topic of the study. If no experts are at hand their webpages may contain the papers for the test-set.

Another way of evaluating the search is to use a complementary search strategy. For example, if the initial search was done with databases, someone can use e.g. snowballing for evaluation. Petticrew et al. [58] proposed doing this evaluation step multiple times until a stoppage criterion has been met. The stoppage criterion can relate to the number of papers found per iteration, e.g. stop if less than x papers are added during an iteration. Time-budget may be another stoppage criterion.

4.1.5 Data Extraction and Classification

The final step after compiling a relevant and evaluated set of papers is to extract the data. To avoid errors the extraction process should be done by at least two researchers. Either each researcher extracts the data for all the papers and the results are compared, or one can check the outcome of the other one [55].

The classifications can be divided into two separate kinds. There are topic-independent classifications and topic-specific classifications. The first kind contains properties which every publication has. This may be the venue of publication, the published year, the research type or the research methods. Topic-specific classifications on the other hand are, as the name suggests, bound to the topic of research. These topics may arise during the study, or they may already exist in literature. [55]

4.2 Methodological Approach

In this section the procedure described in Section 4.1 is applied to context of finding existing quality models for test-suite evaluation. Petticrew et al. [58] mentioned that two or more researchers are required to perform a mapping study. This was not possible for this study.

4.2.1 Research Questions

As mentioned before, the goal of this part of the thesis is to find quality models for test-suite evaluation, which are already discussed in literature. Specifically, this mapping study should produce a list of metrics that are used to evaluate the quality of test-suites. The list should be ordered by occurrence count in literature. Each metric should be described in such subtlety, so that it can be used in the next part of this thesis. To better distinguish these research questions from the research questions mentioned in Chapter 3, the research questions are called here knowledge questions.

Out of these requirements, the following knowledge question arise:

KQ1 What metrics have been investigated by researchers for test-suite evaluation?

KQ2 How intensive have these metrics been investigated?

KQ3 Where and when have studies about metrics for test-suite evaluation been published?

KQ1 addresses the main goal of the study. *KQ2* and *KQ3* give us information about how important each found metric is and when and where research to those metrics has been done.

4.2.2 Search for Primary Studies

As mentioned in section 4.1.2, conducting a mapping study requires deciding which search strategy to use first. In this study database search is used. This search strategy can easily be done by one researcher, but at the same time leads to a representative result. As described in the section before mapping studies in the context of software engineering only need to use IEEE, ACM and two indexing databases, such as Inspec/Compendex and Scopus.

The following databases are used:

- IEEE Xplore [1]
- ACM Digital Library [2]
- Scopus Preview [3]

Inspec/Compendex [88] is currently part of Elsevier and can be searched via Scopus. Therefore, no second indexing database has been chosen.

IEEE Xplore

IEEE Xplore digital library is a research database by IEEE (Institute of Electrical and Electronics Engineers) and its partners. IEEE is a professional association for electrical engineering, computer science, and electronics. IEEE Xplore provides web access to more than 250 journals and more than 3 million conference papers. All in all the database contains more than five million full-text documents [83].

[1] https://ieeexplore.ieee.org
[2] https://dl.acm.org
[3] https://www.scopus.com

ACM Digital Library

The ACM Digital Library is a research, discovery and networking platform containing all ACM publications, including journals, conference proceedings, technical magazines, newsletters and books. The ACM (Association for Computing Machinery) is an umbrella organization for academic and scholarly interests in computer science. The ACM Digital Library has indexed than 3 million publications [82].

Scopus Preview

Scopus is the largest abstract and citation database of peer-reviewed literature by Elsevier. Delivering a comprehensive overview of the world's research output in the fields of science, technology, medicine, social sciences, and arts and humanities, Scopus features tools to track, analyze and visualize research [89] Scopus offers free features to non-subscribed users and is available through Scopus Preview [90]. Scopus covers more than 82 million documents [87].

4.2.3 Search Terms

Each of the database mentioned above supports full-text search via search terms. We define those terms via the PICO methodology mentioned in 4.1.2.

- *Population:* This study is in the general research area of software testing. So this is our population.
- *Intervention:* In this study we look for methods to evaluate test-suites. So "test suite evaluation" is the intervention term.
- *Comparison:* We do not limit our search to any specific comparison methodology/-tool/technology/procedure. So there won't be any comparison term.
- *Outcome:* The outcome of the studies we look for should be a metric or score. Therefore, those two are our outcome terms.

Figure 4.1 shows the full search terms. Each term from the PICO analysis has been joined together with an *AND* expression. An *OR* expression connects synonyms. The "test suite evaluation" terms has been expanded by some synonyms.

("software testing")
AND
("test suite evaluation" OR "test suite assessment" OR "test suite quality")
AND
("metric" OR "score")

Figure 4.1: Search terms

Each database has a different query language and therefore the search terms have to be adapted for each database. The result of this adaption can be seen in Table 4.1.

Database	Search term	Link
Scopus Preview	("software testing") AND ("test suite evaluation" OR "test suite assessment" OR "test suite quality") AND ("metric" OR "score")	[98]
ACM Digital Library	AllField:("software testing") AND (AllField:("test suite evaluation") OR AllField:("test suite assessment") OR AllField:("test suite quality")) AND (AllField:("metric") OR AllField:("score"))	[84]
IEEE Xplore	("All Metadata":"software testing") AND ("All Metadata":"test suite evaluation" OR "All Metadata":"test suite assessment" OR "All Metadata":"test suite quality") AND ("All Metadata":"metric" OR "All Metadata":"score")	[91]

Table 4.1: Adapted search terms per database.

4.2.4 Screening of Papers for Inclusion and Exclusion

Inclusion and Exclusion play a vital part in this study. Because there is only one researcher conducting this study close attention has been paid on automating as much as possible. The following steps are the algorithm performed by software and researcher for inclusion and exclusion. You can find an implementation of this algorithm online[4].

1. Delete all duplicates found in the publication set.
2. Check the title for keywords
   a) Include a publication if it contains at least one of the following keywords:
   test: It is highly likely, that a publication contains *test* in its title if it covers software testing.
   mutation or *mutate*: Mutation testing is an established metric to validate test-suites.
   coverage: Measuring coverages are an established metric to validate test-suites.
   b) If it does not meet the requirement mentioned above, then it should be checked manually. The publication is included if it is about software testing in general.
3. Check the abstract for keywords

[4] https://github.com/steeeve1510-master-thesis/MappingStudy

a) Include a publication if it contains the phrase *"test suite"*: The publication should focus on test-suites, therefore it is plausible that this term would be mentioned in the abstract.

b) If it does not meet the requirement mentioned above, then it should be checked manually. Again the publication is included if it is about software testing in general.

4. Manual review: All remaining publications are checked manually, if they are peer-reviewed and if they introduce some new quality model to determine test-suite-quality. Alternatively a publication is also included if it uses a quality model for test-suite-evaluation in its analysis or evaluation.

4.2.5 Evaluate the Search

Snowballing is used for evaluation. To circumvent the limitation, that only one researcher is conducting the snowballing, an algorithm which automates as much as possible has been defined. For the evaluation it is sufficient to only consider venues, which have been determined as A or A* by the CORE Conference Ranking [86]. This further reduces the effort for the evaluation. Again, you can the find an implementation online[5].

The software implements the following algorithm:

1. Remove publication without DOI
2. For all publications get the forward- and backward-references
3. Check the title for keywords. Include a publication if it contains at least on of the following keywords: *test*, *mutation* or *mutate*, *coverage*.
4. Check the abstract for keywords. Include a publication if it contains the phrase *"test suite"*.
5. Check if publication was published in an A or A* venue
6. Manual Review
   a) Include if the full-text for the publication is available AND
   b) Include if the publication introduces a new way to measure test-suite quality.
7. Remove publications which were already in the initial set.

When more than 20% new publications relative to the database search set has been found, then another snowballing iteration has to be performed. Otherwise, the snowballing is done.

[5] https://github.com/steeeve1510-master-thesis/MappingStudy

4.2.6 Data Extraction and Mapping of Studies

For data extraction we scan the remaining publications for quality models. To make things easier we search for the quality models in a hierarchical manner: First we check the abstract. If the quality model is not apparent in the abstract we scan the conclusion. If there are any ambiguities left we take a deeper look into the publication (e.g. check discussion and evaluation, read the whole paper as a last resort). We have two different types of data we want to extract, namely topic-independent and topic-dependent data.

Topic-Independent Data

Topic-Independent data is information that every publication has, e.g. what year it was published in or where it was published. We focus on the following four data-points:

1. *Year*: The year the paper was published
2. *Venue*: In which journal, conference, etc. the paper was published
3. *Venue Type*: What is the type of the venue, e.g. is it a journal or a conference?
4. *Search Strategy*: This piece of information is specific to this study, but nevertheless topic-independent. How did we find this paper? Because we only search for papers via a database search and snowballing a paper can only have one of those search strategies.

All information may not be extractable from the full-text itself. So it may be necessary to conduct an online-search again Specifically for a publication.

Topic-Dependent Data

We look for two data points that can be classified as topic dependent. The first one is called *Metric*. This is the name of the quality model used in the publication to evaluate test-suites. The second one is called *Metric Type*. Here the metric should be grouped into broader terms. Such terms could be *Mutation Score*, as it's the baseline we want to compare to, or *Coverages*, where different coverage criteria can be grouped together. As the study progresses other terms can be added. This data point is only used to give some overview over the found metrics, and therefore hasn't to be strictly grouped.

4.3 Results

In the following section the results of the mapping study described above are presented. The first section contains a deeper look into the execution of the study and see how the number of publications evolved over the study. In the subsequent sections the result relevant to the knowledge questions are presented.

4.3.1 Study Selection Process

The initial database search resulted in 137 publications. Table 4.2 shows the publications per database. 49 publications were found with Scopus Preview, 79 publications were found with ACM Digital Library and 9 additional publications were found with IEEE Xplore.

Database	Search results
Scopus Preview	49
ACM Digital Library	79
IEEE Xplore	9

Table 4.2: Number of studies per database.

Figure 4.2 depicts how the number of publications changed during the study. 20 duplicates were removed from the initial set. Checking if the all publications have full-text available resulted in one removal. Checking the title of the publications removed ten publications and the abstract review removed additional three publications. After manual inspection of all remaining papers only 73 were marked as relevant. Snowballing added twelve more publications. In the end 85 publications were identified as relevant for this study. You can find the full list of publications in the bibliography.

How the number of publications changed during snowballing can be seen in Figure 4.3. Out of the 73 publications that remained after full-text reading two had to be removed, because they had no DOI. Backwards snowballing resulted in 2145 publications, of which 1195 did not meet the inclusion/exclusion criteria for the title. 950 Publications remained. Forward snowballing resulted in 3148 publications. After applying inclusion/exclusion criteria on the titles 928 publications remained. Merging the backwards and forwards snowballing sets resulted in 1515 papers. Keep in mind that the resulting number does not contain any duplicates. This causes the result of this step to be lower than the sum of the two snowballing sets. Applying the inclusion/exclusion criteria on the abstract removes 1003 publications and applying the inclusion/exclusion criteria on the venue removes another 356 publications. This leaves us with 156 publications for manual inspection. Twelve publications were marked as relevant in the last step.

In Figure 4.4 the distribution across the different search strategies is shown. 72 publications have been found with database search and twelve publications have been found through snowballing.

4.3.2 Frequency of Publication

Most of the papers found were published from 2013 and onward, as shown in Figure 4.5. The first paper was released by Harder et al. [129] and is the only one in 2003. The following ten years only one or two papers were published each year. 2013 an onwards six to 13 publications have been made public each year.

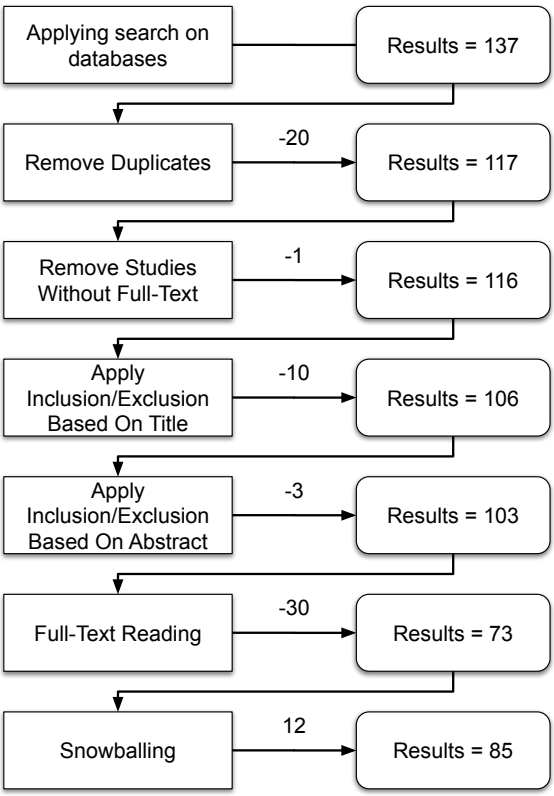

Figure 4.2: Number of included articles during the study selection process.

4.3.3 Venues

Figure 4.6 depicts the venues, were the papers have been published. The top venue is International Conference on Software Engineering (ICSE) with eleven publications followed by International Conference on Automated Software Engineering (ASE), International Conference on Software Testing, Verification and Validation Workshops (ICSTW) and Proceedings of the International Symposium on Software Testing and Analysis (ISSTA) with six publications each. The most featured journals are the Software Quality Journal (Softw. Qual. J.) and IEEE Transactions on Software Engineering (IEEE TSE) with three publications each.

When looking at the distribution of proceedings articles to journal articles, we can see that proceedings articles are the most prevalent. This is depicted in Figure 4.7. 68 papers were made public as proceedings articles and only 16 articles have been published in a journal.

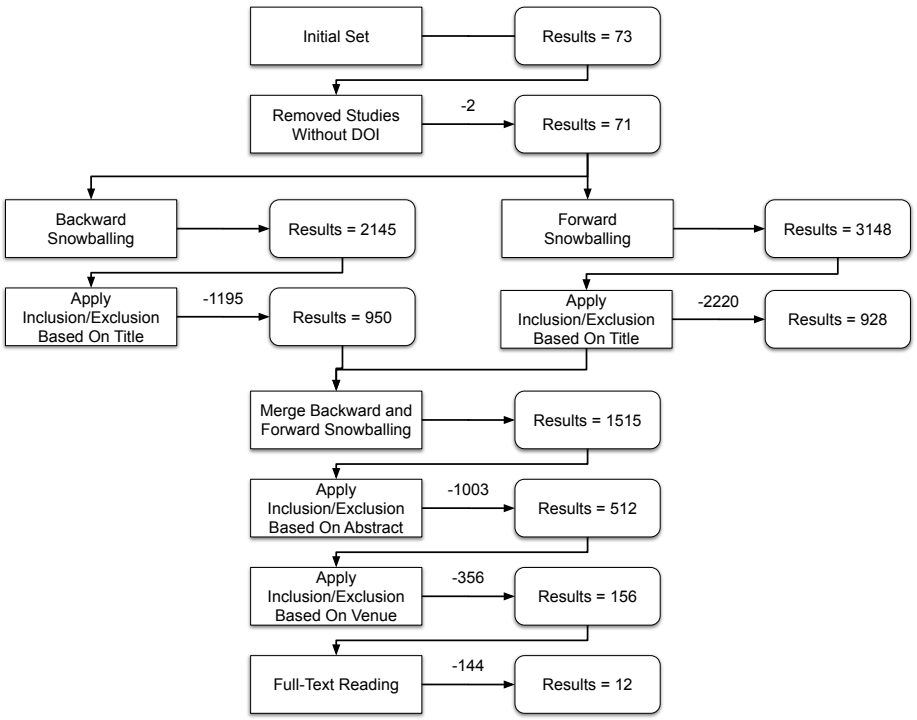

Figure 4.3: Number of included articles during the snowballing process.

4.3.4 Metrics

In total 56 metrics have been found in this study. Most of the publications introduce a new metric, but it does not get picked up by further research, as seen in Figure 4.8. 37 times a metric was only examined once and ten times a metric was used or examined twice. *Conditional Coverage* and *Modified Conditional Coverage* have been found four times each, *Test Smells* five times. Deeper investigated metrics include *Line Cover* and *Code Coverage* with eight examinations and *Method Coverage* with eight uses. The top three metrics are *Branch Coverage*, *Statement Coverage* and *Mutation Testing*, with 19 examination for the first two metrics and 54 mentions for the latter.

When grouping the metrics together we can see that most of them are using some sort of coverage. Figure 4.9 shows that coverage was used by 113 publications followed by mutation testing with 55 publications. Eight papers used a quality model based on static code analysis and three used model inference. Nine used some other method.

In Figure 4.10 we can see how many unique metrics are grouped together per type. We see that there is a wide variety of coverages. 40 different types of coverage quality models have been identified in this study. The other types are rather homogeneous. Only two different types of mutation testing, four different types of static code analysis and three different types of model inference have been identified. Seven metrics could not be

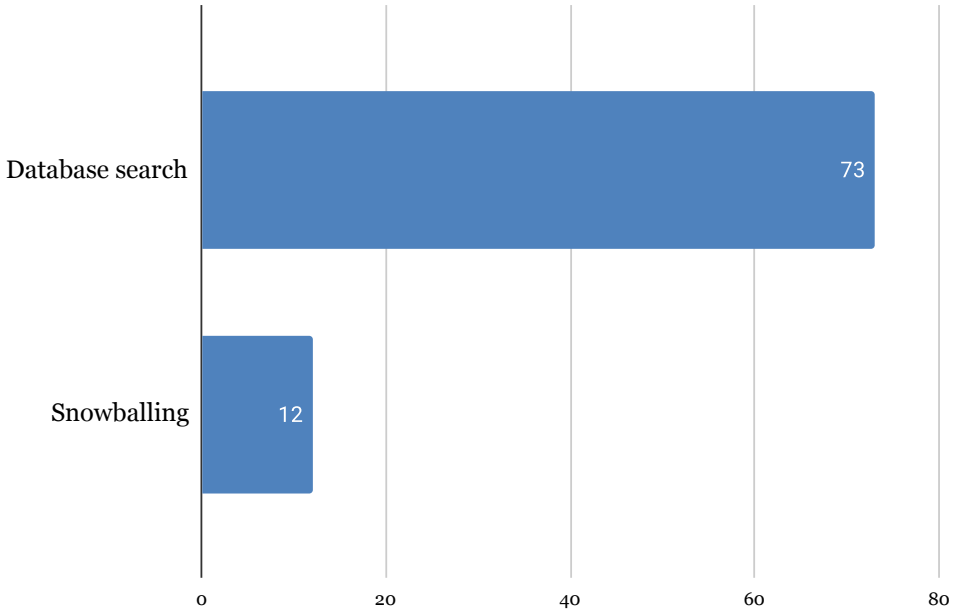

Figure 4.4: Publications per search strategy.

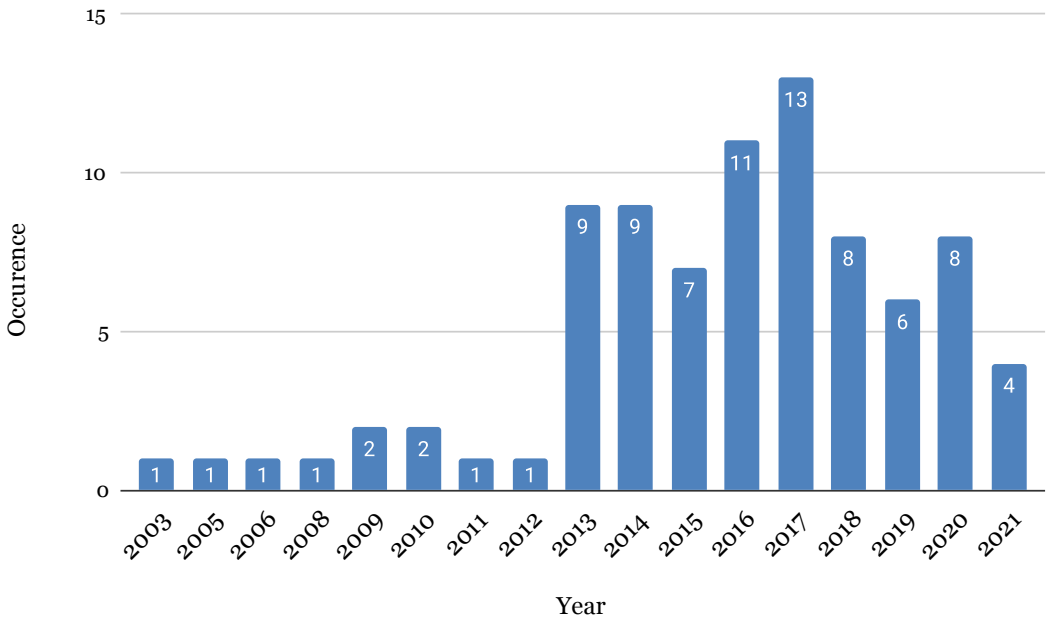

Figure 4.5: Publications per year.

matched with an appropriate type.

4.3.5 The most frequent Metrics in Detail

In this section we take a closer to look at the metrics, which have been mentioned in at least two publications in Section 4.3.4. Mutation Testing has already been covered in Section 2.3.1 and therefore is not mentioned here. The metrics are listed in order of frequency of occurrence, with the most common one listed first.

Statement Coverage

The idea of coverages is already quite old. Miller et al. [46] were the first one who published the idea in 1963. Statement coverage (or instruction coverage) is a subtype of this idea. It refers to the fraction of the statements in a program that are run by the test-suite at least once [136, 85]. Statement coverage is usually the weakest adequacy criteria and yet the most used one in practice [22]. The concept of statement coverage is easy to understand and easy to measure, making it popular with developers [136].

Branch Coverage

Branch coverage is similar to statement coverage, but the statements are the important part here. The branches are. Branch coverage examines all branches of a program. Both true and false cases of each decision point are relevant [80]. To reach 100% branch coverage, all branches must be executed. In contrast to statement coverage the implicit else-branch of if-statements has to be executed too [85]. Branch coverage subsumes statement coverage. In other words, it is a stronger adequacy criterion [80].

Method Coverage

Method coverage reports for each method or function if it was invoked. It is more suitable for software that consists of many small methods rather than a few large methods [77].

Line Coverage

Line coverage is very similar to statement coverage, but it differs if more than one statement is in one line. Line coverage is fulfilled, if one of the statements in a line is covered. Statement coverage looks at each statement individually [77].

Code Coverage

Code coverage itself is an umbrella term, that is used for all coverage metrics covered here. The reason this in this list is, that some papers in the mapping study don't need to distinguish between different coverage types and only use code coverage. We won't be able to use this metric directly in the further steps. But for the sake of completeness it shall be mentioned here.

Test Smells

Coverage criteria on its own won't determine if a test-suite is maintainable and follows good test-design-practices. Test code might contain poor design choices. Some of those bad choices can be structurally identified. These are called test smells. The presence of smells in test code may reduce the quality of test-suites and, consequently, the quality of the production code. Additionally, poorly-written tests can be difficult to understand, which make it difficult for testers to maintain the code and detect faults [167].

Aljedaani et al. [3] gathered many test-smells known to the research community. Some selected test smells are listed in Table 4.3.

Modified Condition Coverage

Each decision point in the source code can be an atomic condition or a boolean expression composed of several atomic conditions. Depending on the concrete values of the involving variables in the conditions, a final decision is evaluated to be true or false. Modified Condition Coverage covers the combinations of atomic conditions where the value of the atomic condition can affect the overall decision independently. In other words, the outcome of a compound decision changes as a result of changing each single condition [26].

Conditional Coverage

Conditional Coverage is a weaker stronger adequacy criterion than Modified Condition Coverage. The basic concept is that, when a decision is made by boolean expression, we want to make sure that each atomic condition is tested in the true and false case [47].

4.4 Discussion

In this section we look at each knowledge question defined in Section 4.2 and answer them using the results presented in Section 4.3. Finally, *RQ1* is answered.

4.4.1 *KQ1* What metrics have been investigated by researchers for test-suite evaluation?

As expected we find mutation testing in the list of metrics. This isn't surprising as we already knew beforehand, that mutation testing is a crucial method for determining test-suite quality. On the other hand we included "mutation" and "mutate" in the inclusion/exclusion criteria, in the assumption that those papers would be about test-suite quality.

Figure 4.10 shows, that a wide range of metrics are based on determining some kind of coverage. This suggests that researchers think that coverages are a promising approach to evaluate test-suite quality. A statement by Inozemtseva et al. [136] could also explain this diversity. They suggest that the idea of coverage is easy to understand and therefore

popular with developers, suggesting that it is easy to tweak existing coverage metrics and create a new one.

Other types of metrics include static code analysis, model inference and some others. But there aren't many approaches using them. This suggests that it is important that quality models for test-suite evaluation require to be easy to understand and include runtime information.

Looking at the individual metrics it can be observed that the list contains metrics which have been adapted in software engineering practice, like statement-, branch-, method-, conditional coverage and mutation testing. But metrics, which aren't typically linked to test-suite quality, like test-suite runtime or line-of-code. Finally, there are a lot of metrics which aren't widely known, like CUBA, DDU or MAP-Coverage.

The full list of metrics can bee seen in Figure 4.8.

4.4.2 *KQ2* How intensive have these metrics been investigated?

Figure 4.8 shows how often each metric used in the publications found by this study. Mutation testing has been investigated in the context of test-suite evaluation by far the most. It was found in 55 papers. This underlines the argument by Ma et al. [43] that mutation testing is the "gold standard" for experimental evaluations of test methods.

The next popular metrics after mutation testing are more common known coverage criteria, like statement coverage and branch coverage with 19 occurrences each, and method coverage with ten occurrences and line coverage with eight occurrences. In Figure 4.9 it can be observed that coverages generally are well studied in the context of test-suite evaluation. Again this can be explained by the cheap calculation and easy to understand concept [136].

Another metric with 5 occurrences are test smells. Test smells try to find bad design choices in test code and therefore should be directly linked to test-suite quality. Therefore, we would expect to find it more often.

The most remaining metrics all have been investigated once or twice, which shows that there are still new ideas how test-suite quality can be measured.

4.4.3 *KQ3* Where and when have studies about metrics for test-suite evaluation been published?

Research about test-suite evaluation happens mostly at conference proceedings, followed by journal publications. The ratio between conference proceedings and journal publications is ~1:4. Other publication forms, like books etc., could not be found. Figure 4.7 shows this in more detail. Figure 4.6 lists all the venue where the papers have been published.

When looking for the publications dates it can be observed that prior to 2013 there wasn't much research in the area of test-suite evaluation. With the first one in 2003 only

one or two publications have been found. 2013 represents a change in research curiosity in the topic. The following years around 9-10 publications per year have been found. Why this change in curiosity happened could not be established. Figure 4.5 shows the number of publications found per year.

4.4.4 *RQ1* What are existing metrics for test-suite evaluation?

KQ1 and *KQ2* show which metrics have been investigated by research and how intensive these metrics have been investigated. As expected mutation testing is on that list. The list also includes well known evaluation methods, like statement-, branch-, method- and conditional coverage. Figure 4.9 and Figure 4.10 show that mutation testing has been heavily investigated, which underlines the statement by Ma et al. [43], that mutation testing is the gold standard for test-suite evaluation. Furthermore, a wide variety of coverage metrics has been investigated. Other test-suite evaluation types, like static code analysis and model inference only play a minor role in current research. In total there have been 8 mentions of static code analysis and 3 of model inference. The most studied test-suite evaluation method from static code analysis is test smells.

KQ3 gives an overview where and when metrics have been investigated. Most of the research for test-suite evaluation metrics has been done in conference proceedings, with a ration to journal publications of ~1:4. It seems like, that test-suite evaluation is a rather new field in research. Interest in the topic rapidly increased in 2013.

4.5 Limitations and Threats to Validity

This section takes a critical look back at the mapping study and identifies possible threats to its validity and limitations.

The biggest limitation for the mapping study is the number of researchers conducting the study. Petersen et al. [56] mentioned explicitly, that at least two researchers should be available for a mapping study, to avoid errors and to verify the manual processes performed during a study. But in this thesis only one researcher was available, and therefore the risk is given that some errors have been made along the way. Further research should fulfill this requirement.

Another threat to validity are the decisions made for the mapping study. Although unlikely, as Petersen et al. [56] mentioned, the chosen databases may not be optimal, and other databases may have lead to a different result. The same argument can be made with the respect to the chosen search terms and the keywords used for inclusion and exclusion. But it is safe to assume, that different decisions may have lead to other metrics, but overall outcome would most likely be very similar.

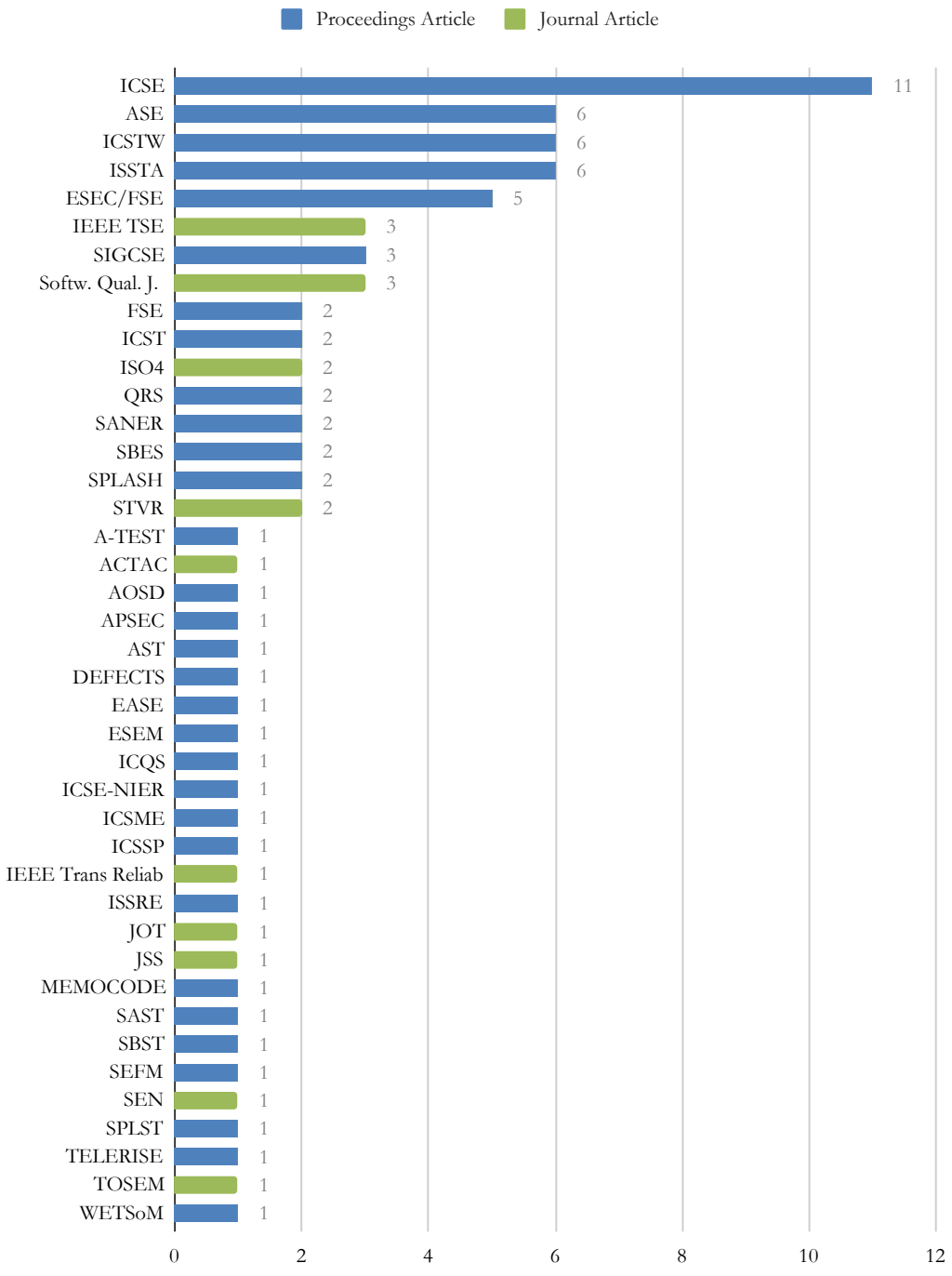

Figure 4.6: Publications per venue.

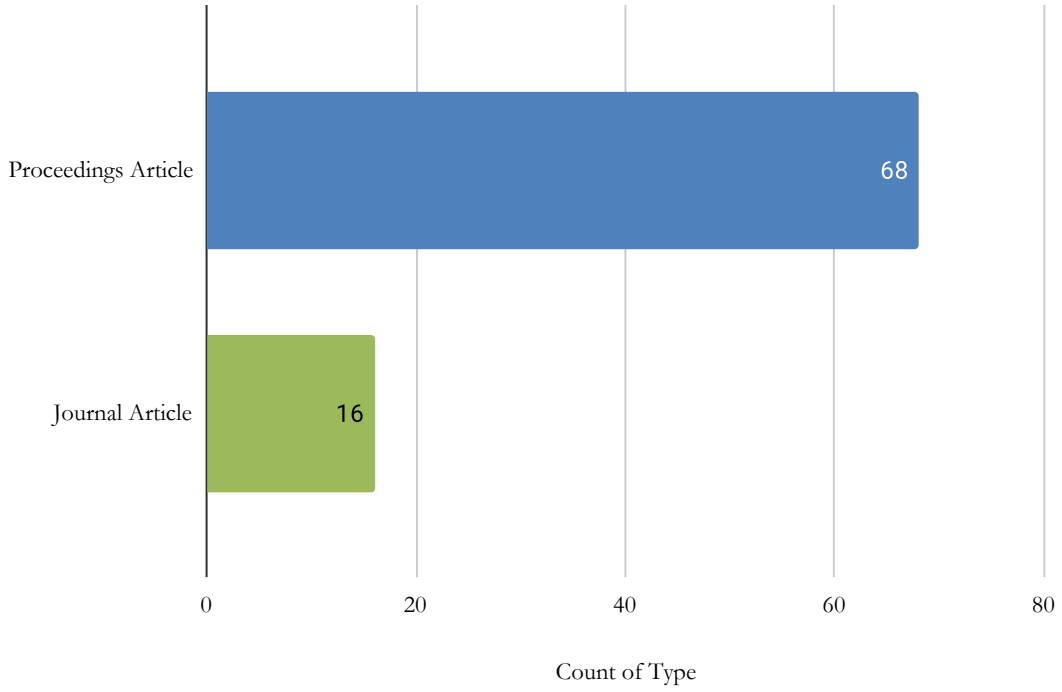

Figure 4.7: Publications per venue type.

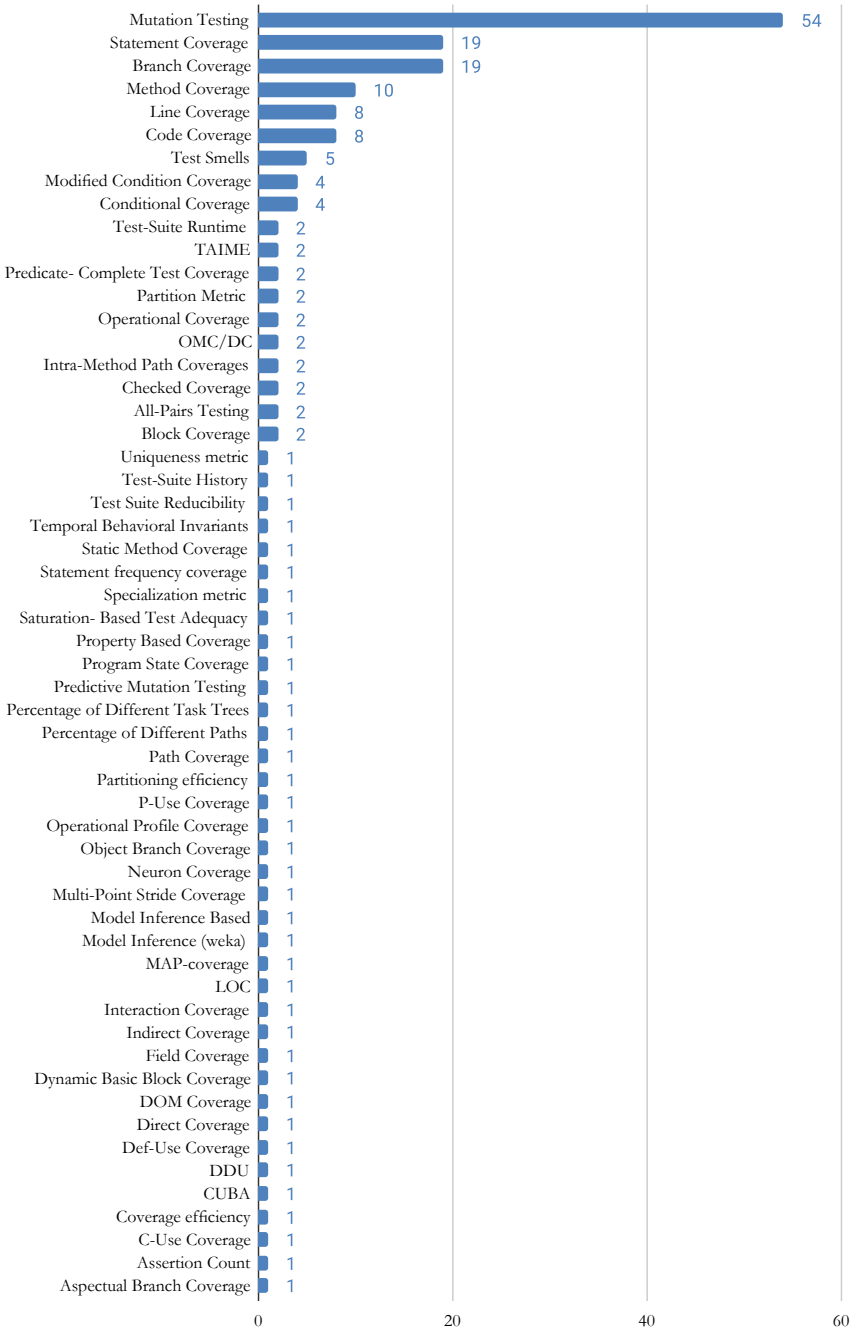

Figure 4.8: Publications per metric.

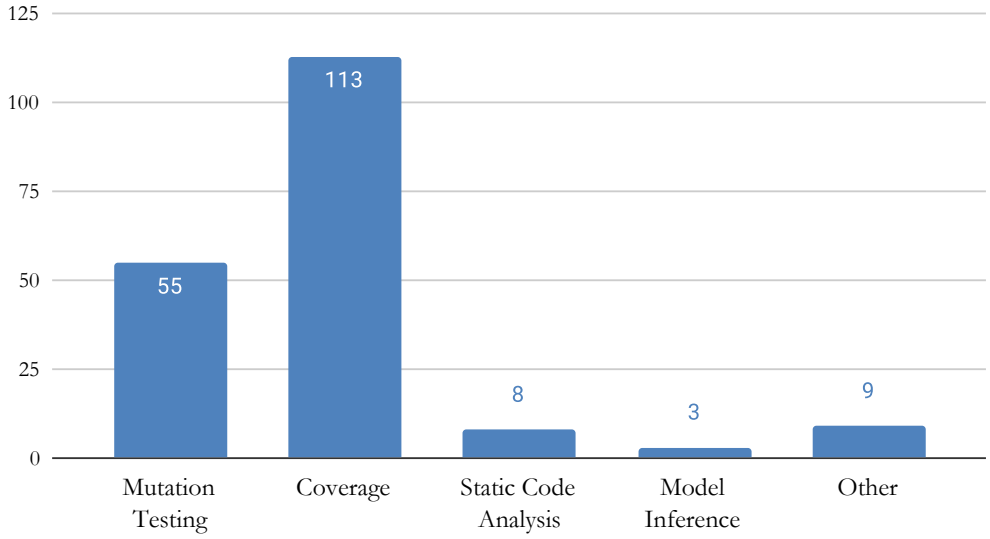

Figure 4.9: Publications per metric type.

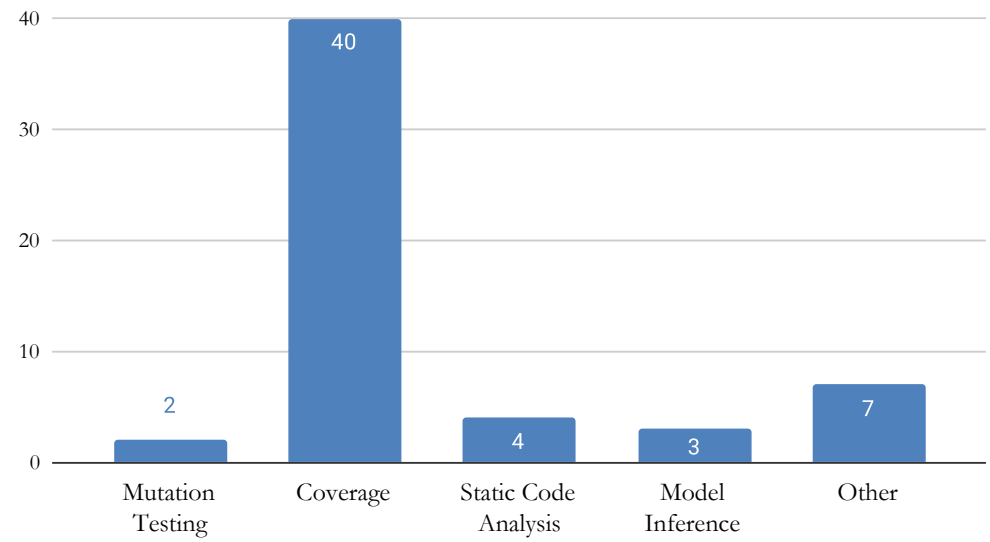

Figure 4.10: Distinct metrics per metric type.

Test Smell Name	Definition
Conditional Test Logic	A test method that contains a conditional statement as a prerequisite to executing the test statement.
Constructor Initialization	A test class that contains a constructor.
Default Test	Default or an example test-suite created by Android Studio.
Dependent Test	A test that only executes on the successful execution of other tests.
Duplicate Assert	Occurs when a test method has the exact assertion multiple times within the same test method
Empty Test	A test method that is empty or does not have executable statements.
Exception Handling	Occurs when custom exception handling is utilized instead of using JUnit's exception handling feature.
General Fixture	This smell emerges when setUp() fixture creates many objects, and test methods only use a subset.
Ignored Test	A test method that uses an ignore annotation which prevents the test method from running.
Magic Number Test	A test method that contains undocumented numerical values.
Mystery Guest	A test that uses external resources, such as a database, that contains test data.
Redundant Assertion	A test method that has an assertion statement that is permanently true or false.
Redundant Print	A test method that has print statement.
Resource Optimism	A test that make an assumption about the existence of external resources.
Sensitive Equality	Occurs when an assertion has an equality check by using the toString method.
Sleepy Test	Occurs when a test method has an explicit wait.
Unknown Test	A test method without an assertion statement and non-descriptive name.
Verbose Test	Test code that is complex and not simple or clean.

Table 4.3: Definition of diverse test smells.

CHAPTER 5

Evaluating the Novel Quality Model

After having established what test-suite evaluation methods have been studied in literature, this chapter uses the found methods and combines them in various ways using machine learning. To evaluate the variants two experiments are conducted. The first one establishes a baseline and the second one uses the baseline to identify if the variants improve on existing test-suite evaluation methods and compares the variants among themselves.

5.1 Experiment Design

This section explains the experiments for the evaluation in detail. The evaluation contains two experiments. The first experiment provides a baseline for the actual evaluation of the novel quality model. The second experiment is the evaluation of the quality model, which uses the baseline provided by the first experiment. The following two sections dive deeper into the experiments. Each experiment follows the scope and planning structure proposed by Wohlin et al. [75] with a slight deviation. Instead of hypotheses, the experiments use knowledge questions as proposed by Wieringa [74] to gain knowledge from the obtained data.

5.1.1 Experiment 1: Baseline Experiment

This experiment serves the purpose of identifying a baseline. We can establish the baseline by answering the following knowledge question:

KQ1 How do single metrics perform in predicting mutation testing results in terms of effectiveness and efficiency?

The knowledge question contains the terms effectiveness and efficiency. In this context, effectiveness describes the ability to predict mutation testing results. To answer this part of the knowledge question several regression evaluation metrics are collected. The metrics are R^2 score, Mean Squared Error (MSE), Root Mean Squared Error (RMSE) and Mean Absolute Error (MAE). A detailed description about those evaluation metrics can be found in Section 5.2.6. Efficiency here means runtime. More precisely the following condition is evaluated: Is the test-suite evaluation method faster than mutation testing in terms of runtime? This condition is evaluated for each test-suite evaluation method. No extra metric is gathered to answer this question. We reason about this property manually, by analysis of the method description.

The following sections introduce the experiment in detail in the structure proposed by Wohlin et al. [75].

Scoping

Object of Study: The objects of the experiment are the test-suite evaluation methods found in Chapter 4.

Purpose: The purpose of this experiment is to evaluate the ability of the test-suite evaluation methods to predict mutation scores in the context of a single software project.

Perspective: The experiment is conducted in the perspective of researchers, who are interested in the link between test-suite evaluation metrics and mutation testing, and software developers, who want to gain more insights from test-suite evaluations.

Quality Focus: Each test-suite evaluation method is evaluated in terms of effectiveness and efficiency to predict mutation testing results.

Context: The experiment is run in the context of open source software projects, which have to be collected for this experiment (see Section 5.2).

Scope Summary: Analyze test-suite evaluation methods for the purpose of evaluation with respect to their effectiveness and efficiency to predict mutation testing results from the point of view of researchers and software developers in the context of open source projects.

Planning

Context Selection: The experiment is run offline, which means it is run independent of any productive setting. This is also called a controlled experiment. The observation do not cause any effects in the software projects. The projects used in this experiment are no toy projects, and they are built by professionals. From this experiment we can only deduce result for the projects used. Further research is needed to generalize the results.

Variables Selection: Chapter 4 identified several test-suite evaluation method. The methods, who have been found in at least two different papers in the literature study, are used as independent variables. The independent variables are:

- Instruction Coverage
- Branch Coverage
- Line Coverage
- Method Coverage
- Test Smells

Modified Condition Coverage and Conditional Coverage are not included in this list, as no suitable tool for this experiment setup could be found. The list of independent variables contains the umbrella term test smells. Which concrete test smells are used for the experiment depends on the tool which extracts them. All test smells supported are used.

The dependent variable for this experiment is mutation testing.

Selection of Subjects: This experiment uses open source projects from GitHub. More specifically Java projects, which are built with Maven are used. How those projects are gathered is described in Section 5.2 in more detail. The way projects are collected here is called Convenience Sampling as proposed by Wohlin et al. [75].

Experiment Design: For each project all test-suite evaluation metrics are calculated. The calculation of the test-suite evaluation metrics has no effect on the artifacts, resulting from a build. So the experiment is not blocking. All the projects are evaluated with all test-suite evaluation metrics. This results in a balanced experiment.

Instrumentation: All the projects require Java and Maven for building. The mutation score is extracted with PiTest. Test smells are calculated using TestSmellDetector. JaCoCo is used to get coverage information. Tools for Modified Condition Coverage and Conditional Coverage could not be found for Java and Maven. All test-suite evaluation tools place a report in the output directory of the project. *collect-metrics* is used to aggregate the metrics into a single report. A more detailed description about the instrumentation can be found in Section 5.2.

Validity Evaluation: The validity of this experiment is discussed in Section 5.5.

5.1.2 Experiment 2: Evaluate Novel Quality Model

In the second experiment the variants of the novel quality model are evaluated. It uses the baseline, which is established in experiment 1. The evaluation answers the following three knowledge questions:

KQ2 How do the novel quality models based on different regression algorithms perform in predicting mutation testing results in terms of effectiveness and efficiency?

KQ3 How do different categories of test-suite evaluation metrics impact the effectiveness of the novel quality model?

The first knowledge question is directly related to *RQ2*, because it analyzes approaches how a novel quality model can be designed. By analyzing the effectiveness it tackles the similarity part of *RQ2* The efficiency part of the knowledge questions makes sure, that the novel quality model is cheap. The next knowledge question evaluates if the model can be improved by using different test-suite evaluation metrics. This question makes sure, that the novel quality model is as similar as possible to mutation testing. Effectiveness and efficiency is likewise interpreted as in experiment 1. Effectiveness deals with the accuracy to predict mutation testing results. This is measured with R^2 score, MSE, RMSE and MAE. Efficiency deals again with runtime. This is again determined manually be analysis of the test-suite evaluation methods description.

The following sections introduce the experiment in detail in the structure proposed by Wohlin et al. [75].

Scoping

Object of Study: The objects of the experiment are the variants of the novel quality model framework described in Chapter 3.

Purpose: The purpose of this experiment is to evaluate the novel quality model in terms of effectiveness and efficiency it predicts mutation score for a given project in reference to existing test-suite evaluation methods.

Perspective: The experiment is conducted in the perspective of researchers, who are interested in the novel quality model and its performance, and software developers, who want to use the quality model for precise and fast predictions of mutation testing results.

Quality Focus: The variants of the novel quality model framework are evaluated in terms of effectiveness and efficiency to predict mutation testing results.

Context: The experiment is run in the context of open source software projects, which have to be collected for this experiment (see Section 5.2).

Scope Summary: Analyze the variants of the novel quality model framework for the purpose of evaluation with respect to its effectiveness and efficiency in reference to existing test-suite evaluation methods to predict mutation testing results from the point of view of researchers and software developers in the context of open source projects.

Planning

Context Selection The experiment runs in the same context as experiment 1, which means that the experiment a controlled experiment. The observation do not cause any effects in the software projects. The projects are no toy projects, and they are built by professionals. From this experiment we can only deduce that the novel quality model

works, or not, with the found projects. Further research is needed to generalize the results.

Variables Selection There is only one independent variable for this experiment, namely the evaluation scores calculated by the instances of novel quality model.

The dependent variable for this experiment is, again, mutation testing.

Selection of Subjects The subjects are the same as for experiment 1. They are open source projects from GitHub, who are built using Java and Maven. How those projects are gathered is described in Section 5.2 in more detail. The way projects are collected here is called Convenience Sampling as proposed by Wohlin et al. [75].

Experiment Design For each project the evaluation score and the mutation score are calculated. These calculations itself have no effect on the artifacts, resulting from a build. So the experiment is not blocking. All the projects are evaluated with the new evaluation score and mutation testing. This results in a balanced experiment.

Instrumentation The instrumentation for this experiment is very similar to experiment 1. Again Java and Maven are required for building the projects. The mutation score is extracted with PiTest. The quality model requires Test Smells and Coverages. Therefore, TestSmellDetector is used again for Test Smell extraction and JaCoCo is used for Coverages. Tools for Modified Condition Coverage and Conditional Coverage could not be found. All those tools place a report in the output directory of the project. *collect-metrics* is used to aggregate the metrics into a single report. Additionally, *analyze-metrics* is used to build the novel quality model variations. A more detailed description about the instrumentation can be found in Section 5.2.

Validity Evaluation The validity of this experiment is discussed in Section 5.5.

5.2 Data Collection

This section describes how the data is gathered for the experiments. This section explains how the projects are found and how the metrics are extracted.

5.2.1 Project Selection

Just et al. [33] introduced a database and extensible framework providing real bugs to enable reproducible studies in software testing research, called Defects4J. These projects serve as a starting point for the project search in this thesis. Defects4J is free available on GitHub [1]. The faults in the projects are not needed here. Instead, it is more important to easily build the projects. So the projects in Defects4J were not used in the version proposed, but in the current version available on GitHub.

[1] https://github.com/rjust/defects4j

Additionally, projects are gathered through the GitHub trending page [2] and the GitHub Java topic page [3].

Only projects are considered which are built with Java[4] and Maven[5].

5.2.2 Metric Calculation Tool Selection

Both experiments need single test-suite evaluation metrics either to evaluate them against mutation testing or use them to calculate a new evaluation score. The following sections introduce the tools used to extract these metrics.

JaCoCo

JaCoCo [6] is a Java code coverage library, that runs JUnit test cases. It was created by EclEmma [7] and is also available for Maven. JaCoCo calculates the following metrics:

- Instruction Coverage (see Section 4.3.5)
- Branch Coverage (see Section 4.3.5)
- Cyclomatic Complexity Coverage
- Line Coverage (see Section 4.3.5)
- Method Coverage (see Section 4.3.5)
- Class Coverage

We ignore cyclomatic complexity coverage and class coverage, as they do not appear in the literature review performed in Chapter 4.

For the experiments the version 0.8.7 is used. The plugin definition shown in Appendix A is used to incorporate JaCoCo into Maven.

TestSmellDetector

TestSmellDetector [8], or tsDetect, is an automated test smell detection tool for Java software systems that uses a set of detection rules to locate existing test smells in test code [54]. The tool supports the following test smells:

- Assertion Roulette

[2] https://github.com/trending
[3] https://github.com/topics/java
[4] https://www.java.com
[5] https://maven.apache.org
[6] https://www.jacoco.org/jacoco/trunk/index.html
[7] https://www.eclemma.org/jacoco
[8] https://testsmells.org/index.html

- Conditional Test Logic
- Constructor Initialization
- Default Test
- Duplicate Assert
- Eager Test
- Empty Test
- Exception Handling
- General Fixture
- Ignored Test
- Lazy Test
- Magic Number Test
- Mystery Guest
- Redundant Print
- Redundant Assertion
- Resource Optimism
- Sensitive Equality
- Sleepy Test
- Unknown Test

Unfortunately, TestSmellDetector does not provide support for Maven. Therefore, a Maven Plugin for TestSmellDetector is written as part of this thesis. The code of the plugin is available online[9].

The plugin definition for the TestSmellDetector Maven plugin is shown in Appendix A.

5.2.3 Mutation Testing Tool Selection

Besides other test-suite evaluation metrics, mutation testing serves a special purpose. It is the dependent variable for both experiments, and is the desired output value for the machine learning training in experiment 2. PiTest is used to extract the mutation testing results from the gathered projects.

[9] https://github.com/steeeve1510/TestSmellDetectorMavenPlugin

PiTest

PiTest [10] is a state-of-the-art mutation testing tool, that mutates JVM bytecode. Because it operates on bytecode, it is fast, and it optimizes mutant executions. It can be called through a command line interface, Ant or Maven [15].

It supports the following mutators by default:

- Conditionals Boundary
- Increments
- Invert Negatives
- Math
- Negate Conditionals
- Return Values
- Void Method Calls
- Empty returns
- False Returns
- True returns
- Null returns
- Primitive returns

For the experiments the version 1.7.3 is used. The plugin definition shown in Appendix A is used to incorporate PiTest into Maven.

5.2.4 Reducing Builds

With the current setup we get one datapoint per Maven module. But to efficiently build a machine learning model we need at least thousands datapoints. We can increase the number of datapoints by using the process of reducing builds. Reducing builds works under the assumption, that test-suite quality gets worse, the fewer tests a test-suite has. Therefore, we can use a test-suite, remove some test cases, and should get a worse test-suite. For reducing builds we iteratively remove a single test to get a slightly worse test-suite. For each worsened test-suite the tests are run and the datapoint is collected.

An easy example for reducing builds can look as follows. Consider an example project with three test cases, $TC1$, $TC2$ and $TC3$. With the reducing build strategy we can get four datapoints for the following four test-suites:

[10] `https://pitest.org`

1. $TC1$, $TC2$, $TC3$
2. $TC2$, $TC3$
3. $TC3$
4. No test cases

We do not enforce any order, in which the test cases are removed. Any order is fine. We could even get more datapoints by using all possible permutations of tests. But this is computationally very expensive and would dampen the data diversity, because fewer projects could be included into the data set.

5.2.5 Metric Results Aggregation Tool

Each tool mentioned in Section 5.2.2 and Section 5.2.3 puts their test-suite evaluation results into the build directory of the software project. To use this data for the experiments we have to aggregate the reports, and if necessary normalize the data. *collect-metrics* is implemented with the intention of doing exactly that. The tool supports JaCoCo, TestSmellDetector and PiTest. It aggregates the data in the following way:

JaCoCo: The data is aggregated per Maven module. This means for each coverage metric the output value is the percentage covered instruction/branches/etc. in the Maven module.

TestSmellDetector: Here for each metric the values are summed up per Maven module. Additionally, the number of methods per module, as provided by TestSmellDetector, is calculated. The output value is the metric sum divided by the number of methods.

PiTest: The output value for PiTest is the amount of killed mutants per Maven module divided by the number of all mutants per Maven module.

collect-metric is incorporated into the build via a Maven plugin. You can find the source code online[11].

5.2.6 Evaluation Metrics

This section describes each metric, which is used for evaluation of the test-suite evaluation methods and quality model. There are five metrics used, namely R^2, R^2-Predicted, MSE, RMSE and MAE.

R^2

A useful statistic for checking a regression fit is the R^2 score. R^2 is the quotient of the explained variation (sum of squares due to regression) to the total variation (total sum

[11] https://github.com/steeeve1510-master-thesis/collect-metrics

of squares total) [17]. The score can have max value of 1. The value can get negative, but is usually between 0 and 1. A good regression is indicated by a higher value.

Equations 5.1, 5.2 and 5.3 show how the score is calculated, where y_i denotes the value of an observation i, $\hat{y}_i$ the predicted value for an observation i and $\overline{y}$ the mean of all observations [48].

$$R^2 = 1 - \frac{SS_{res}}{SS_{tot}} \tag{5.1}$$

$$SS_{res} = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2 = \sum_{i=1}^{n} e_i^2 \tag{5.2}$$

$$SS_{tot} = \sum_{i=1}^{n}(y_i - \overline{y})^2 \tag{5.3}$$

R^2-Predicted

The R^2-Predicted statistic gives some indication of the predictive capability of the regression. It can indicate if a regression model is overfitted to the training data [48]. The score can have max value of 1. Negative values are possible. A good predictive capability is indicated by a higher value.

The R^2-Predicted is similarly calculated than the R^2 score. But it uses the PRESS statistic. Equations 5.5, 5.4 and 5.3 show the R^2-Predicted is calculated. The PRESS statistic is just the ordinary residual weighted according to the diagonal elements of the hat matrix h_{ii} [48]. The hat matrix is defined as the matrix that converts values from the observed variable into estimations obtained with the least squares method [17].

$$PRESS = \sum_{i=1}^{n} e_{(i)}^2 = \sum_{i=1}^{n}(\frac{e_i}{1 - h_{ii}})^2 \tag{5.4}$$

$$R^2{-}Predicted = 1 - \frac{PRESS}{SS_{tot}} \tag{5.5}$$

Mean Squared Error

The Mean Squared Error (MSE) is the average of the squares of the errors. In other words, it is the average squared difference between the estimated values and the actual value.

The formula used in this thesis for MSE is shown in Equation 5.6, where y_i is the actual observation and $\hat{y}_i$ is the predicted observation. The value is always positive and lower values indicate a better model.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{5.6}$$

Root Mean Squared Error

The Root Mean Squared Error (RMSE) is similar to the MSE. The only difference is, that the RMSE calculates the square root after the errors are summed up.

Equation 5.7 shows how the RMSE is calculated. As with MSE, the value is always positive and lower values are better.

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2} \tag{5.7}$$

Mean Absolute Error

At last this thesis uses the Mean Absolute Error (MAE). It is similar, to MSE and RMSE, but it doesn't square the error, but it takes its absolute value.

The formula for the MAE is shown in Equation 5.8. y_i is the actual observation and $\hat{y}_i$ is the predicted observation. The result of this formula is always positive and lower values indicate a better model.

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \tag{5.8}$$

5.3 Results

In this section the results of the two experiments are presented. First some overview of the data collection process is shown. The other sections cover the baseline experiment and the quality model experiment.

5.3.1 Data Collection Process

In total 15 projects are used for the experiments. An overview of the projects is shown in Table 5.1. All the projects have at least 100 commits, with the most active ones over 7000. The projects were all started before 2015, with the oldest one starting in 2002. We can expect distinct programming styles in the data, because all the projects had many authors. The project with the least authors is *jackson-dataformat-xml* with 29 authors. *java-design-patterns* has the most authors with 336. The projects range from smaller ones, with around 50 files 12000 lines of code, to larger ones with over 1000 files to and 200000 lines of code. Please consider that lines of code here only counts *.java* files. The files counter include files with all file endings.

An overview of the datapoints gathered per project can be seen in Table 5.2. The column plain datapoints lists the amount of datapoints without reducing builds per project. The column expanded datapoints lists the amount of datapoints results from the reducing build per project. Without only 206 datapoints could have been calculated. Reducing builds inflated this number to 12503. The most datapoints are gathered from

Project	Commits	First Commit	Authors	Files	Lines of Code (.java Files)
commons-cli	1197	2002-06-10	60	52	11480
commons-codec	2259	2003-04-25	53	147	43404
commons-compress	3490	2003-11-23	84	550	118487
commons-csv	1770	2005-12-17	49	45	13985
commons-lang	7028	2002-07-19	215	447	166882
commons-math	7743	2003-05-12	80	1156	260262
gson	2921	2008-04-24	133	212	38436
jackson-core	2138	2011-12-22	74	273	78049
jackson-databind	6630	2011-12-22	225	1066	224455
jackson-dataformat-xml	1359	2010-12-30	29	170	23294
jfreechart	4383	2007-06-29	39	1023	283004
feign	1298	2012-03-18	210	296	44802
hutool	5609	2014-04-13	241	1756	237633
java-design-patterns	3209	2014-08-09	336	1534	103330
jsoup	2086	2009-12-19	118	133	35909

Table 5.1: Overview of Open Source Projects.

the commons-lang project, followed by jackson-databind and jfreechart. Reducing builds was not possible for the projects marked with (*).

5.3.2 Baseline Experiment

In Figure 5.1 the distributions of mutation score across all datapoints is shown. Most of the mutation scores datapoints are in the range from 0.2 to 0.3. It has around 3000. The top range from 0.9 to 1 has the fewest entries. Here only ~100 datapoints are present. The histograms for all other metrics can be found in the Appendix Section B.

Table 5.3 shows R^2, MSE, RMSE and MAE of the single metrics in reference to mutation score. As expected mutation score itself has the maximum R^2 score and no errors. Only five of the other metrics have positive R^2 values. Out of those five, four of them have R^2 values lower than 0.3. The entry after mutation score is branch coverage, with a R^2 score of ~0.65 and instruction coverage with a R^2 score of ~0.29. Magic number test is the highest ranking test smell, which can be found directly after instruction coverage. It has a R^2 score of ~0.27. Following magic number test are the remaining coverage criterion. These are followed by all the remaining test smells. Note that the remaining test smells all have a negative R^2 score. When looking at the MSE and RMSE we can see that the order of metrics stays the same. The value increases the further down you go the table. MAE mostly follows this trend, with the only exception being method coverage. From redundant assertion onwards the MAE stays roughly the same.

Project	Plain Datapoints	Expanded Datapoints
commons-cli	1	255
commons-codec	1	835
commons-compress	1	1149
commons-csv	1	426
commons-lang	1	3533
commons-math*	6	6
gson	3	33
jackson-core	1	704
jackson-databind	1	2741
jackson-dataformat-xml	1	347
jfreechart	1	2286
feign*	27	27
hutool*	19	19
java-design-patterns*	141	141
jsoup*	1	1
Total	**206**	**12503**

Table 5.2: Datapoints for Open Source Projects.

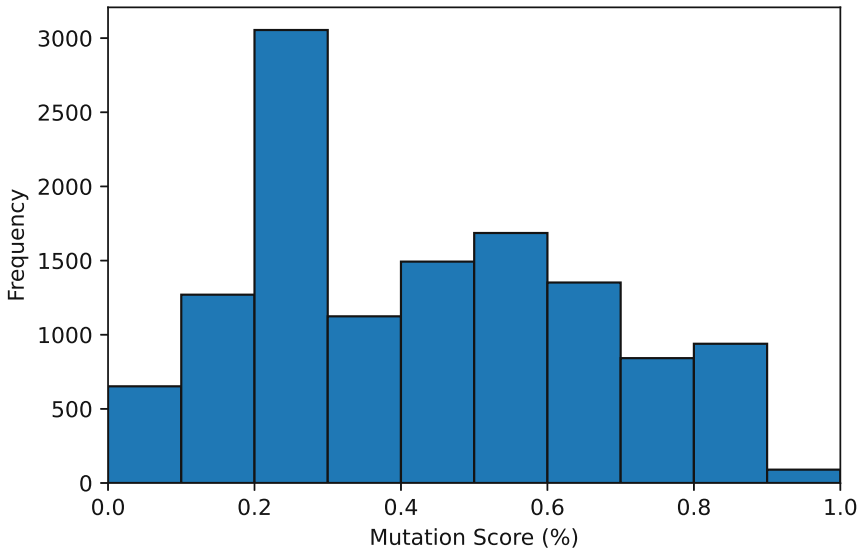

Figure 5.1: Mutation Score Histogram.

Metric	R^2	MSE	RMSE	MAE
Mutation Score	1.00000	0.00000	0.00000	0.00000
Branch Coverage	0.64805	0.01886	0.13735	0.09090
Instruction Coverage	0.29419	0.03783	0.19450	0.14498
Magic Number Test	0.27428	0.03890	0.19723	0.14683
Line Coverage	0.25726	0.03981	0.19953	0.14979
Method Coverage	0.22629	0.04147	0.20365	0.14410
Assertion Roulette	-0.32668	0.07111	0.26667	0.21587
Exception Catching Throwing	-1.16287	0.11593	0.34049	0.25774
Duplicate Assert	-2.44894	0.18487	0.42996	0.36692
Unknown Test	-2.75633	0.20134	0.44871	0.39093
Sensitive Equality	-2.84306	0.20599	0.45386	0.40445
Conditional Test Logic	-2.96686	0.21263	0.46112	0.40570
Ignored Test	-3.36921	0.23420	0.48394	0.42622
Redundant Assertion	-3.37379	0.23444	0.48419	0.42567
Mystery Guest	-3.37691	0.23461	0.48436	0.42263
Resource Optimism	-3.37710	0.23462	0.48437	0.42240
General Fixture	-3.39823	0.23575	0.48554	0.42768
Print Statement	-3.43329	0.23763	0.48747	0.42903
Constructor Initialization	-3.43356	0.23764	0.48749	0.42890
Empty Test	-3.43612	0.23778	0.48763	0.42925
Sleepy Test	-3.43954	0.23796	0.48782	0.42937
Verbose Test	-3.44273	0.23814	0.48799	0.42957
Default Test	-3.44273	0.23814	0.48799	0.42957
Dependent Test	-3.44273	0.23814	0.48799	0.42957

Table 5.3: Single Metrics Evaluation.

5.3.3 Quality Model Experiment

Figures 5.2, 5.3 and 5.4 depict the histograms of the three quality model variants. They all look very similar to the mutation score histogram, which is shown in Figure 5.1. The predicted range from 0.2 to 0.3 has the most points, with around 520 entries. The range with the fewest entries is 0.9 to 1.

The numerical evaluation of all quality model variants can be seen in Table 5.4. For each machine learning regressor there are four different data sets. The rows marked with "Overall" were evaluated with all the datapoints collected. The rows marked with "Train" were evaluated with the 80% train data set of the overall model. The "Test" rows were evaluated with the 20% test data set, which was used to test the overall model. The last entry is the result of 10-fold cross validation.

We can see that random forest produces the most accurate model. The random forest models with only coverages and only test smells both have smaller R^2 values than the

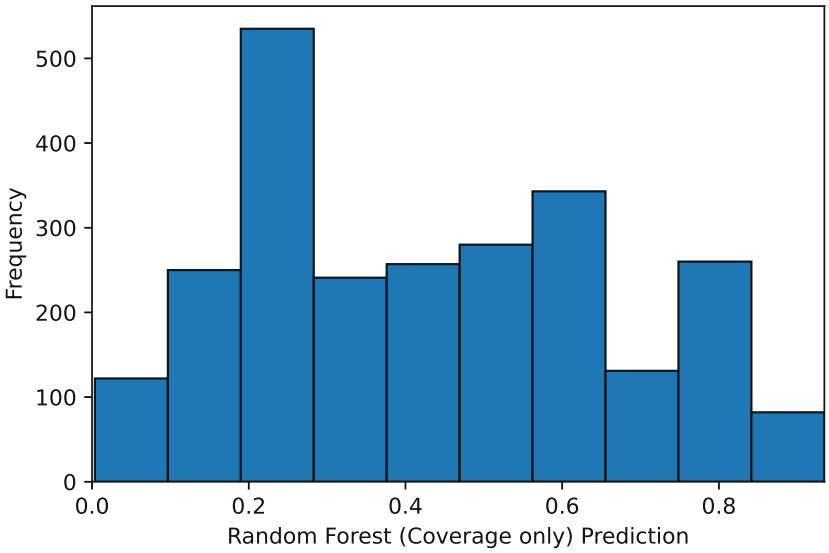

Figure 5.2: Random Forest (Only Coverages) Histogram.

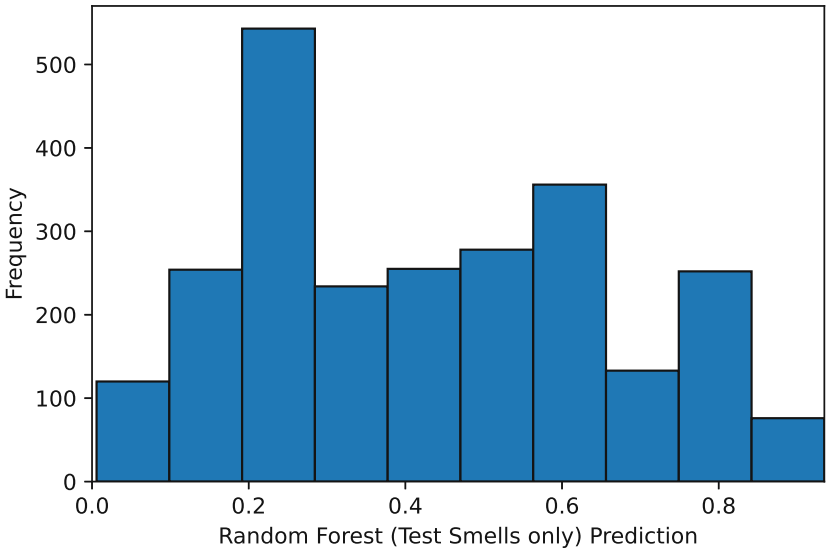

Figure 5.3: Random Forest (Only Test Smells) Histogram.

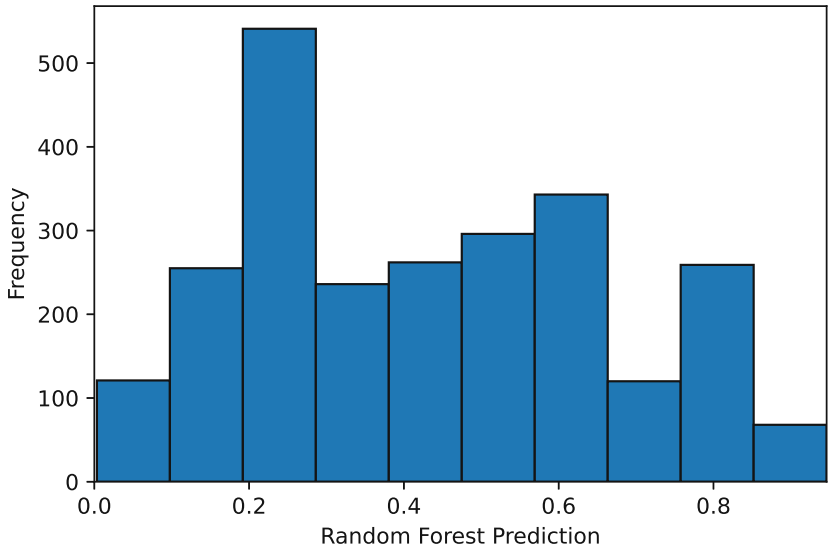

Figure 5.4: Random Forest Histogram.

variant with all the features. All the other machine learning models get lower R^2 values and higher error values as any random forest variant. All the models have similar R^2-Predicted values, which indicates that the models have no bias. MSE, RMSE and MAE get larger the lower the R^2 values get.

Table 5.5 shows all the test smell evaluation metrics used and in the data set they were part of. Some metrics could not be included in any data set, because the collected data did not have enough variance. The variance threshold is 0.05^2 for all data sets. + denotes all features which pass the filter, −− denotes that all features which got removed by the filter. As the variance threshold is the same for all data sets, a metric gets removed in all data sets if the variance is too low. All the coverage metrics pass the filter. Only assertion roulette, exception-catching-throwing, duplicate assert, unknown test and magic number test pass the variance filter.

5.4 Discussion

The following sections take the results presented in Section 5.3 and use them to answer the knowledge questions introduced in Section 5.1. Based on the knowledge gained from *KQ1*, *KQ2* and *KQ3* an answer to *RQ2* is compiled.

Regressor	Data-Set	R^2	R^2-Predicted	MSE	RMSE	MAE
Random Forest	Overall	0.98229	0.98039	0.00095	0.03088	0.00361
	Train	0.97954	0.97572	0.00112	0.03344	0.00338
	Test	0.98180	0.97593	0.00096	0.03098	0.00642
	10-fold CV	0.98574	0.97990	0.00077	0.02705	0.00309
Random Forest (Coverage only)	Overall	0.97210	0.97143	0.00150	0.03876	0.00492
	Train	0.97468	0.97348	0.00138	0.03720	0.00467
	Test	0.98326	0.98228	0.00088	0.02971	0.00774
	10-fold CV	0.98351	0.98161	0.00088	0.02909	0.00383
Random Forest (Test Smells only)	Overall	0.97306	0.97166	0.00145	0.03808	0.00561
	Train	0.96215	0.95950	0.00207	0.04548	0.00591
	Test	0.96588	0.96224	0.00180	0.04241	0.01074
	10-fold CV	0.96731	0.96359	0.00175	0.04098	0.00571
Support Vector Machine	Overall	0.89306	0.89006	0.00576	0.07588	0.06355
	Train	0.87147	0.86144	0.00702	0.08381	0.06367
	Test	0.87194	0.85592	0.00675	0.08217	0.06838
	10-fold CV	0.89332	0.88153	0.00572	0.07543	0.06258
Linear Regression	Overall	0.84263	0.83832	0.00847	0.09205	0.05406
	Train	0.84607	0.83728	0.00841	0.09172	0.05436
	Test	0.84100	0.82163	0.00838	0.09156	0.05435
	10-fold CV	0.85236	0.83109	0.00789	0.08878	0.05292
Ridge Regression	Overall	0.84191	0.83768	0.00851	0.09226	0.05429
	Train	0.84504	0.83712	0.00847	0.09203	0.05486
	Test	0.83061	0.81144	0.00893	0.09451	0.05762
	10-fold CV	0.85184	0.83420	0.00792	0.08894	0.05321
Elastic Net	Overall	-0.00007	-0.00825	0.05384	0.23204	0.20118
	Train	-0.00100	-0.01256	0.05471	0.23390	0.20300
	Test	-0.01040	-0.04939	0.05327	0.23081	0.19899
	10-fold CV	-0.00116	-0.02550	0.05355	0.23139	0.19957
Lasso	Overall	-0.00007	-0.00825	0.05384	0.23204	0.20118
	Train	-0.00100	-0.01256	0.05471	0.23390	0.20300
	Test	-0.01040	-0.04939	0.05327	0.23081	0.19899
	10-fold CV	-0.00116	-0.02550	0.05355	0.23139	0.19957

Table 5.4: Machine Learning Models Evaluation.

Feature	Overall	Coverages	Test Smells
Instruction Coverage	+	+	
Branch Coverage	+	+	
Line Coverage	+	+	
Method Coverage	+	+	
Assertion Roulette	+		+
Conditional Test Logic	–		–
Constructor Initialization	–		–
Default Test	–		–
Empty Test	–		–
Exception Catching Throwing	+		+
General Fixture	–		–
Mystery Guest	–		–
Print Statement	–		–
Redundant Assertion	–		–
Sensitive Equality	–		–
Verbose Test	–		–
Sleepy Test	–		–
Duplicate Assert	+		+
Unknown Test	+		+
Ignored Test	–		–
Resource Optimism	–		–
Magic Number Test	+		+
Dependent Test	–		–

Table 5.5: Features Passing the Variance Filter per Feature Set.

5.4.1 *KQ1* How do single metrics perform in predicting mutation testing results in terms of effectiveness and efficiency?

This question has two parts and the two parts are answered separately.

The first part is about the effectiveness of the gathered test-suite evaluation methods in their ability to predict mutation score. To answer this R^2, MSE, RMSE and MAE have been calculated for each test-suite evaluation method. Table 5.3 shows all the results of these calculations. We can see that that most of the test smells do not have any meaningful potential to predict mutation score. This is indicated by negative R^2 values and the high values for MSE, RMSE and MAE. For example many of them have a RMSE higher than 0.4 with a possible output range from 0 to 1. This means we can expect errors of more than 40%. Consider a mutation score prediction by print statement of 0.5, or 50% of all test methods have a print statement. With a RMSE of 0.4 the actual mutation score could be in the range from 0.1 to 0.9. Only magic number test has a positive R^2 value of ~0.27, which means the metric can explain around 27% of

the variance. All the coverage metrics do have a positive R^2 value with the best one being branch coverage with a R^2 value of ~0.65. Branch coverage is more than twice as good as instruction coverage with a R^2 value of ~0.29. RMSE and MAE are lower for the coverage with not exceeding values of 0.2 and 0.15 respectively. We can conclude that only branch coverage has a meaningful potential to predict mutation score. But the RMSE and MAE are still rather high. Take for example the RMSE of ~0.14. It makes a huge difference if the mutation score is 0.14 higher or lower. To rely on the predicted value, the R^2 should definitely be higher, and the errors should be as low as possible.

To answer the efficiency part of the knowledge questions we take a look at the descriptions in Section 4.3.4. We can determine if a metric is efficient we only have to clarify if the calculation of a metric can be faster done in terms of runtime. We can see that each of the metrics can be calculated in a single build of the project. Therefore, we know that each metric is calculated more effective than mutation score.

5.4.2 *KQ2* How do the novel quality models based on different regression algorithms perform in predicting mutation testing results in terms of effectiveness and efficiency?

Again this questions contains two parts. The first part is about the effectiveness and the second about the efficiency. These parts are answered separately.

Effectiveness is interpreted here as the ability to predict mutation testing results as precise as possible. To measure the effectiveness a couple of values are calculated. These are R^2, R^2-Predicted, MSE, RMSE and MAE. Table 5.4 shows these values for all the regressors. Random forest produced the best model with a R^2 value over 0.98, which means that 98% of the variance of mutation score can be explained by the random forest model. This is almost 10% better than the second-best model, which is SVM, with a R^2 value of ~0.89. The effectiveness of random forest can also be seen in Figure 5.4 and Figure 5.1. The figures show the histogram for random forest model and mutation score respectively. Visually one can see the similarities between the two histograms. RMSE and MAE suggest too very high precision for both models with 0.00095 and 0.00361 for random forest and 0.00576 and 0.06355 for SVM. Linear regression and ridge regression follow SVM with slightly lower R^2 values. These models predict mutation score very precise too. On average the absolute error for those two are just ~0.05. Only elastic net and lasso could not produce models, which predict mutation score. The MAE and RMSE there are already over 0.2. The R^2-Predicted values are similar to the R^2 values. Except for elastic net and lasso, this indicates that the models do not have a bias. 4 out of 6 regressor models perform well with really high R^2 values, with random forest being the best by 10%.

To answer the second part of this knowledge question we look into the efficiency of the regressor models. We classify a regressor model as efficient, if the required test-suite evaluation metrics and the regressor model combined do not exceed the mutation testing runtime for a project. We know from *KQ1* that each metrics used for the models are

calculated in a single run of the test-suite. Random forest models can be applied in polynomial time of the size of the forest and the depth of the trees, which is rather quick for our use case [66]. Therefore, we can conclude that all the models used here are more efficient than mutation testing.

5.4.3 *KQ3* How do different categories of test-suite evaluation metrics impact the effectiveness of the novel quality model?

The random forest model has been chosen to evaluate the different feature categories, because it most accurately predicts mutation score based on the R^2 value as described in Section 5.4.2. The first three entries of Table 5.4 show the different test-suite evaluation metric sets. All the categories produce similar good models. Random forest with all the metrics evaluates the best with an R^2 of 0.98229. Test Smells only produces a model which is slightly worse. Based on the R^2 it's 0.9% lower. The random forest model using only the coverage metrics has 1% lower R^2 value than the model produced by the set with all the features. It should be mentioned here, that many test smells had to be removed from all data sets, because the available data didn't have enough variance. The included metrics can be seen in Table 5.5. More diverse data may result in even a better quality model.

5.4.4 *RQ2* How can existing metrics be combined to get a cheap quality model similar to mutation score?

Random forest produces a precise model for predicting mutation scores, while at the same time being much faster than mutation testing. The quality model based on coverages and test smells performs the best.

KQ1 has shown, that the best test-suite evaluation metric are only mediocre in predicting mutation score. In Table 5.3 we can see the best test-suite evaluation metric is branch coverage with a R^2 of ~0.65 and a RMSE of ~0.14.

In *KQ2* we have looked at several machine learning algorithms, which combine a variety of test-suite evaluation metrics. Table 5.4 shows the evaluation of all machine learning algorithms. Random forest is the best with a R^2 of ~0.98 and a RMSE of ~0.03. The random forest model is more than 50% more precise than the best single metric. The RMSE of the novel quality model is only 24% of branch coverage.

Additionally, *KQ2* shows that the prediction can be calculated much quicker than mutation score itself. The novel quality model can be calculated in a single run of the test-suite of project. The time overhead of the random forest model itself is only polynomial in the size of the forest and the depth of the trees. This means that for a project with 100 mutants the novel quality model with random forest can be up to 100 times faster than mutation testing. The larger the amount of mutants, the better the performance improvement of the novel quality model.

KQ3 has shown that models get better if the novel quality model contains more test-suite evaluation metrics. But the increase in precision is rather slim. Table 5.4 shows the results for random forest with all the metrics, only coverage and only test smells. The model with all the metrics is only ~1% more precise than the models with only coverage and only test smells. Future work has to determine if even more metrics can increase the quality of the model even further.

5.5 Limitations and Threats to Validity

This section takes a critical look back at this chapter and identifies possible threats to its validity and limitations.

The training data collected for the novel quality model has limitations. Although the data set is rather big, it is only collected from 15 projects. Many of them come from similar backgrounds, e.g. Apache Commons and Jackson. This may bias the data towards specific programming and testing habits, especially for test smells. Furthermore, only Java projects using Maven are considered. No statements for other technologies can be made. Future research should extend this study with a more diverse data set to verify the result.

The machine learning part of this thesis contains two notable limitations. Firstly, the amount of regressors is limited. Other and more elaborate regressor algorithms may have lead to a different result. Secondly, the default configuration for all regressors is used. Hyperparameter tuning would definitely lead to more accurate results and maybe another tuned regressor model would have outperformed random forest. However, overall the model found in this study already performs very well. So this does not pose a risk for the validity of this study. Further research should consider other regression models and hyperparameter tuning as well.

The default configuration is used for PiTest, JaCoco and TestSmellDetector. It may be possible that some parts of the projects used should have been excluded from the test-suite evaluation to get a more precise information about the test-suite quality. PiTest supports many more mutant operators, but they are not activated by default. For some projects more precise data could have been extracted, if other mutant operators are used. Future studies should consider contacting the developers for a software project and let them integrate the test-suite evaluation tools. They have more knowledge about the project and therefore can better determine how the tools should be configured.

Finally, some code had to be implemented for the data extraction, model training and evaluation. There is no other researcher available to review the code, so it may be possible that the code contains bugs and biases. Further research should have multiple researchers available to review and test the code implemented.

CHAPTER 6

Related Work

The following sections list publications which are related to this thesis. We compare this thesis from four different perspectives. First we look at mapping studies in the context of software testing. The mapping study performed in Chapter 4 belongs to this category. The following section looks into applications of machine learning in the context of software testing and the subsequent section dives deeper into approaches that use machine learning to improve mutation testing. This is relevant to this thesis, as it also uses machine learning in the context of mutation testing and software testing. Finally, software quality models are investigated, as the novel quality model also fits into this category.

At the end of each section a comparison is given between this thesis and the presented literature. These comparisons also provide some information on how this thesis fits into the broader research field.

6.1 Mapping Studies in Software Testing

Scatalon et al. [64] conducted a mapping study to get deeper insights into software testing in introductory programming courses. They selected 293 publications and mapped them to categories with respect to their investigated topic (curriculum, teaching methods, programming assignments, programming process, tools, program/test quality, concept understanding, and students' perceptions and behaviors) and evaluation method (literature review, exploratory study, descriptive/persuasive study, survey, qualitative study, experimental and experience report). They found that, in terms of investigated topics, over half of the papers (51.19% - 150) are about tools. The topics course materials (1.02% - 3) and concept understanding (0.68% - 2) cover only a small amount of research performed in the area. As to evaluation methods, papers that present empirical studies (survey, qualitative and experimental) comprise 44.7% (131). Papers classified as not applicable or experience report comprise 50.51% (148), slightly above half of selected

papers. Scatalon et al. [64] also identified benefits and drawbacks of including software testing into introductory programming courses. The benefits are:

1. Improvements in students' programming performance
2. Test results can provide students useful information about their programming and testing performance before the assignment deadline
3. Test results provide an objective and consistent way to assign grades to the assignments
4. Better understanding of the programming process

Conversely, Scatalon et al. [64] identified the following drawbacks:

1. Additional workload of course staff
2. Students' testing performance
3. Students' reluctance to conduct testing
4. Programming courses are already packed

Souza et al. [67] used the mapping study methodology to investigate existing research on knowledge management initiatives in software testing. Aspects such as purposes, types of knowledge, technologies and research type were investigated. They found 15 studies relevant for the topic. Most of the studies discuss aspects related to providing automated support for managing testing knowledge. Their conclusions show that:

1. Knowledge management in software testing is a recent research topic
2. The major problem in organizations are low reuse rate of knowledge and barriers in knowledge transfer
3. Reuse of testing knowledge is the main purpose of applying knowledge management in software testing
4. Advanced technologies used to provide knowledge management in software testing include recommendation systems, ontologies and yellow pages (knowledge maps).

Shafiq et al. [65] used a mapping study to investigate how machine learning is applied in the software development process. Their study includes 227 articles. They mapped the articles the following taxonomy:

1. Quality Assurance and Analytics (119 (52%) articles)

2. Architecture and Design (39 (17%) articles)
3. Implementation (21 (9%) articles)
4. Requirements (21 (9%) articles)
5. All Stages (18 (8%) articles)
6. Maintenance (9 (4%) articles)

Quality Assurance and Analytics, being the software development stage with the most number of machine learning related articles, show that software quality is a prime focus for researchers. Shafiq et al. [65] associated each publication with a contribution (tool, approach/method, model/framework, algorithm/process, comparative analysis) and a research facet (evaluation, knowledge, solution). For the category *quality assurance and analytics*, they made the following classification:

1. Fault/Bug/Defect Prediction (50 (20%) articles)
2. Test Case/Data/Oracle Generation (7 (2%) articles)
3. Test Case Selection/Prioritization/Classification (5 (2%) articles)
4. Vulnerability/Anomaly/Malware Discovery/Analysis (19 (8%) articles)
5. Software Analysis (10 (4%) articles)
6. Technique Assessment (5 (2%) articles)
7. Software Process Assessment (3 (1%) articles)
8. Verification and Validation (16 (7%) articles)
9. Testing Effort Estimation (4 (2%) articles)

Most of the publications found in this study used decision trees (33 (15%) articles), naive bayes (31 (14%) articles) or random forest (30 (13%) articles).

Durelli et al. [19] were interested in how machine learning has been explored to automate and streamline software testing. To get deeper insights into this topic they performed a mapping study. Furthermore, they provide an overview of the research at the intersection of machine learning and software testing. They selected 48 primary studies. These studies were categorized according to study type, testing activity and machine learning algorithm employed to automate the testing activity. They concluded their study with the following findings:

1. Machine learning algorithms have been applied to tackle software testing problems since 1995, but only very recently machine learning algorithms caught the interest of researchers and practitioners.

2. The vast majority of the approaches described in the primary studies automate software testing using supervised learning algorithms. Artificial neural networks and decision trees are the most widely used algorithms.

3. Machine learning algorithms have been used mainly for oracle construction and for test-case generation, refinement, and evaluation.

4. A trend they observed is that the oracle problem tends to be tackled by employing either artificial neural network or decision tree based approaches.

5. They found that machine learning based approaches scale very well. Another benefit is that most approaches require minimal human intervention. Conversely, a drawback is that testers will not be able to truly leverage the benefits of machine learning algorithms without understanding the assumptions and implications of these algorithms. They conjecture that the adoption of machine learning algorithms might blur the roles of testers and data scientists. Another drawback is, that the available data must be in a form that facilitates the learning process.

6. There are two problems faced by researchers when trying to apply machine learning algorithms to solve software testing problems. First, most machine learning algorithms need a substantial amount of training data and second, data quality is the key for machine learning algorithms to function as intended.

7. They found that the body of empirical research available at the intersection of machine learning and software testing leaves much to be desired, especially when compared with the level of understanding and body of evidence that have been achieved in other fields.

8. There is no research group specifically dealing with machine learning and software testing.

Comparison to this Thesis

Mapping studies are often used in the context of software testing. The work presented above range from studies in the context of introductory programming courses [64] over knowledge management initiatives [65] to machine learning [65, 19]. All the studies presented are relevant to this thesis, as they share the same methodology. To the best of our knowledge no study was yet performed to investigate test suite evaluation methods. This is a novel contribution by this thesis to the field of software testing.

6.2 Machine Learning applied to Software Testing

The novel quality model presented is part of the context of software testing. Furthermore, it applies machine learning. Therefore, this section takes a deeper look at machine learning applied to software testing.

The publication by Grano et al. [23] shows how machine learning can be used to predict branch coverage. Their preliminary study is using three different types of code metrics as input-features for the machine learning algorithms. The first type contains package level metrics. These metrics contain the number of classes or the number of other packages that depend upon classes within the package. The second type contain Chidamber and Kemere object-oriented measures, like the number of dependencies a class has or the number of static methods in the software project. The last type contains the number of Java reserved keyword. Reserved keywords include *synchronized*, *import* or *instanceof*. The work by Grano et al. [23] concluded with two findings. The first finding is, that despite their simplicity, traditional code metrics give discrete cross-validation results. Support vector regression is the most accurate algorithm amongst the considered machine learning algorithms. The second finding is that machine learning algorithms are a viable option to predict the coverage in automated testing. However, further effort addressed at improving the features and tuning the algorithms needs to be done.

Walkinshaw et al. [73] used an existing test suite of a system to trains several machine learning models. These models encode the behavior of a system. These modes are then used to generated new test cases. A new test case is generated in the following way: At first random inputs for the system under test are generated. These inputs are fed into the models and each model predicts what the outcome for each input is. The predicted outcomes are then compared. The input where the predictions disagree the most is selected as the new test case. In other words, the test case with the most uncertainty is chosen. Walkinshaw et al. [73] calls this approach uncertainty-driven black-box test data generation. The authors provide a proof-of-concept implementation based on genetic programming. The initial results are encouraging. The presented approach outperforms regular and adaptive random testing.

Comparison to this Thesis

The publication by Grano et al. [23] is especially relevant to this thesis, as it uses a similar approach. But they try to predict branch coverage instead of mutation score. This thesis confirms the result by Grano et al. [23] that predicting metrics based on other metrics is a valid approach.

The work by Walkinshaw et al. [73] is relevant to this study, as it tries to encode program behavior into a machine learning model. Though the specifics are rather different. Walkinshaw et al. [73] encodes the behavior directly. Their model encodes the mapping from input to output of a program. The novel quality model presented in this thesis encodes the behavior indirectly through test suite evaluation metrics. This thesis confirms the approach by Walkinshaw et al. [73] is valid. Machine learning can be used to encode program behavior. Additionally, this work has shown that machine learning can be used to encode program behavior through means means than input-output mappings.

6.3 Machine Learning applied to Mutation Testing

The novel quality model presented in this thesis deals with a specific area of software testing, namely mutation testing. For this reason, the following section examines how machine learning is applied to mutation testing.

Zhang et al. [78] proposed predictive mutation testing. It is the first approach to predicting mutation testing results without executing mutants. They built a classification model and used the model to predict if a mutant will be killed or not. The machine learning model uses three different types of features: execution features (like how many times a mutated statement is executed by the whole test suite or how many test cases execute a mutated statement), infection feature (like the type of the mutated statement or the type of the mutant operator) and propagation features (like the complexity of the mutated method or lines of code of a mutated method). In total, they use 15 different features. Their experimental results demonstrated that predictive mutation testing improves the efficiency of mutation testing by up to 151 times while incurring only a small accuracy loss.

Strug et al. [69] tried to reduce the number mutants executed by using machine learning to predict if a mutant will be killed by a test-suite. For their approach the test suite is run on a randomly selected subset of mutants. The mutation testing result of the remaining mutants are predicted based on their similarity to the executed mutants. The similarity of the mutants is measured with help of graph representations. They use the k-nearest neighbors algorithm for their model. Their approach does not need a training phase beforehand. The models are trained on the project it is used on. Strug et al. [69] could not fully validate their approach. It still needs more experiments to fully confirm its validity, but the results obtained are encouraging.

Jalbert et al. [31] use source code and test-suite metrics to predict mutation scores. They group their metrics into four groups: Source code metrics (like lines of code or number of parameters of a method), coverage metrics (like basic blocks covered in code unit or total blocks covered in code unit), accumulated source code metrics (like average parameters of methods or average lines of code of methods) and accumulated test case metrics (like average lines of code of test methods, or average complexity per test methods) They use these metrics to train a support vector machine. They achieve an accuracy of 58.27%, but their training data only consists of one open source project.

Comparison to this Thesis

Machine learning has been applied to many problems, mutation testing being one of them. Many studies try to improve the runtime efficiency of mutation testing [60] with machine learning. Especially relevant to this thesis are the publications by Zhang et al. [78], Strug et al. [69] and Jalbert et al. [31].

The work by Zhang et al. [78] and Strug et al. [69] are similar to this thesis, as it uses machine learning to predict the mutation testing result, but the prediction is done on a

different level. This thesis tries to predict the overall mutation score. The work by Zhang et al. [78] and Strug et al. [69] predict if a single mutant gets killed. The prediction on mutation score level has the advantage that the result can be computed with a single prediction. The approaches by Zhang et al. [78] and Strug et al. [69] have to perform a prediction for each mutant. This suggests, that the novel quality model can be computed more quickly, but rigor evaluation is needed to confirm this claim.

The approach used by Jalbert et al. [31] is the very similar to this work. The only differences are the metrics and data set. The data set used by Jalbert et al. [31] is very small. They only used one program to train their model. This could explain why the accuracy is much smaller, than the accuracy found in this thesis. This thesis shows that the approach by Jalbert et al. [31] is valid. Mutation scores can be predicted with other test suite evaluation metrics. The data set used in this thesis, however, is a novel contribution. It leads to more accuracy for the novel quality model. Future work can use this data set for their own machine learning models or for other evaluations.

6.4 Software Quality Models

The novel quality model presented in this thesis belongs to the overarching area of software quality models. The following section explores this topic in more detail.

McCall et al. [45] presented one of the earliest quality models. The model is also known as McCall's Triangle of Quality. McCall et al. identified three perspectives of software quality: Product revision (the ability to undergo changes), product transition (the adaptability to new environments) and product operations (describes operation characteristics-correctness, efficiency, etc.). Under each perspective, quality attributes are defined as a hierarchy of quality factors, quality criteria, and quality metrics.

Another early quality model was introduced by Boehm et al. [11]. They developed a hierarchy of characteristics of software quality. The quality characteristics are based on three dimensions: utility, maintainability, and portability. These characteristics are broken down into multiple levels, down to primitive characteristic. The elements in the model by Boehm et al. are quality characteristic, sub-quality characteristic, primitive quality characteristic, and quality metric.

Athanasiou et al. [7] presented a test quality model based on metrics obtained through static code analysis. More precisely they used the following five metrics:

- *Code coverage*: Code coverage indicates the percentage of the tested code. But the coverage is not calculated by execution of the test suite. Instead, static call graph analysis is used.
- *Assertion-McCabe ratio*: This metric shows the tested decision points in the code. It is calculated by dividing the number of assertion statements through McCabe's cyclomatic complexity score of the system under test.

- *Assertion Density*: Assertion Density is calculated by dividing the number of assertions statements through the number of lines of test code.

- *Directness*: Directness indicates the ability to detect the location of a defect's cause when a test fails. More precisely, it is the percentage of code that is being called directly by the test code. This is again calculated by static call graph analysis.

- *Maintainability*: For maintainability the SIG quality model is adapted for test suites. The model consists of the following metrics for test code: duplication, unit size, unit complexity and unit dependency.

Nagappan [49] introduced the Software Testing and Reliability Early Warning (STREW) metric suite. It is designed to "provide an estimate of post-release field quality early in software development phases". The metric suite consists of three categories, each category containing different static source code and test code metrics. The categories are:

- *Test quantification*: The test quantification metrics are specifically intended to crosscheck each other to account for coding/testing styles. The metrics are: The number of assertions per line of code, the number of tests per line of code, the number of assertion per test and the ratio between lines of test code and production code, divided by the ratio of test and production classes.

- *Complexity and O-O metrics*: The complexity and O-O metrics examine the relative ratio of test to source code for control flow complexity and for a subset of the Chidamber and Kemere object-oriented measures. The metrics are: the cyclomatic complexity, the coupling between objects, the depth of inheritance tree for and the weighted methods per class. All the metrics are calculated separately for the source and test code.

- *Size adjustment*: The final metric is a relative size adjustment factor. To account for different project sizes they use lines of source code.

Saputra et al. [63] build a test suite quality model using white box testing metrics. Based on ten metrics, like distinct code coverage, number of mutants used in mutation testing and number of test cases, they proposed formulas for usability, efficiency, reliability, functionality, portability, and maintainability. They selected these attributes from 28 attributes found in literature. For example the formula for reliability is shown in Equation 6.1

$$Test\ Suite\ Reliability = \frac{Number\ of\ Mutants}{Number\ of\ Mutants\ killed} \tag{6.1}$$

They validated this approach with expert interviews and Cohen's kappa coefficient. They found that the proposed method is useful to measure test suite quality attributes.

Wagner et al. [72] identified the problem, that most quality models either define abstract quality characteristics or concrete quality measurements. To close this gap they developed Quamoco, a metamodel which aims to close this gap. To accomplish this goal, they introduced the concept of a product factor. Product factors have measures and instruments to operationalize quality by measurements from manual inspection and tool analysis. Additionally, Quamoco allows for modularization to create modules for specific domains. To evaluate their approach they performed empirical studies. They found that the model as well as the quality assessments were highly understandable for practitioners and considered the best that can be done with static analysis.

Comparison to this Thesis

All the presented publications define ways to quantify software quality, from the first ones in the 1970s [45, 11] to more recent ones [7, 49, 72]. Each of them use some way to combine measurable metrics, such as lines of code, in an elaborate way. The novel quality model presented in this thesis follows the procedure, but differs in the way it combines the metrics to get a quality measure. The novel quality model doesn't use a well-defined algorithm, but lets a machine learning algorithm decide how the metrics should be combined.

CHAPTER 7

Conclusion

Software testing is essential to keep the quality of software projects high. Therefore, maintaining a test-suite is an integral component of software development. But to do so is getting more and more difficult the larger and older a project gets. Our brains can only deal with a reasonable amount of complexity or change [22]. One way to determine if a test-suite is efficient is mutation testing. In fact, mutation testing is considered the gold-standard for test-suite evaluation [43]. But practical problems still prevent it to become widely used in industry, the major problem being runtime. This thesis tackles this problem by using existing test-suite evaluation methods, which can be calculated faster than mutation testing, and combining them using machine learning to get a prediction of the mutation testing result.

First a mapping study was conducted to determine existing test-suite evaluation methods from literature. The study revealed that mutation testing indeed is very popular in literature. Most of the publications either try to improve mutation testing or use it as a baseline to evaluate other test-suite evaluation methods. The most studied group of test-suite evaluation metrics is coverage. More than 2/3 of the publications used some kind of coverage. It is also the most diverse group. This study found 40 different kinds of coverage metrics. Beside mutation testing and coverages not much research has been done. The next group of methods is static code analysis, with test smells as the most prominent evaluation method in this group.

With the test-suite evaluation methods established, machine learning was used to combine them to predict mutation testing results. To obtain training data Java projects from GitHub were gathered. From these projects all the test-suite evaluation metrics were extracted. Several regression algorithms were evaluated. Random forest was determined to be the best regression algorithm. Its R^2 is 10% higher than the second best performing regression model, which is SVM. Comparing to single test-suite evaluation metrics, the random forest model is more than 50% better than the best single metric. More test-suite evaluation metrics seem to produce better quality models. With the random forest model

it was established that using all the metrics with enough variance lead to a 1% better model. This improvement is rather small and has to be validated in future work. Linear regression, Ridge regression and SVM did perform good models too, with a R^2 value of > 0.83.

7.1 Future Work

Despite the good results, scrutiny should be given. Future research should repeat this study with other training data and maybe more test-suite evaluation metrics. Other test-suite evaluation methods may lead to better results. Future research should consider using a more diverse set on training data and make sure that the data set is as big as possible.

The novel quality model has one big disadvantage. It cannot indicate how to improve the novel quality score. Mutation testing, on the other hand, produces a list of mutants. If one mutant survives one can easily add or modify a test case to improve the test-suite quality. Future work may improve this approach, by highlighting the results of the used test-suite evaluation metrics. Furthermore, the contribution of each test-suite evaluation metric to the novel quality score should be presented to the user.

As already mentioned in Section 5.4 further research should investigate if more test-suite evaluation metrics always lead to better quality models. The data presented in this thesis does not contain enough evidence to support this claim.

Finally, future work should refine the novel quality model and wrap it up into a tool that practitioners can use. Research should evaluate if the new model is as useful as mutation testing, or if it can reduce the number of full mutation testing runs required. Additionally, the new quality model can be evaluated against other test-suite evaluation methods.

APPENDIX A

Maven Plugin Configuration

Listing A.1: TestSmellDetector Plugin

```
<plugin>
    <groupId>com.heigl</groupId>
    <artifactId>test-smell-detector-maven</artifactId>
    <version>1.0-SNAPSHOT</version>
    <executions>
        <execution>
            <goals>
                <goal>detect</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

Listing A.2: JaCoCo Plugin

```
<plugin>
    <groupId>org.jacoco</groupId>
    <artifactId>jacoco-maven-plugin</artifactId>
    <version>0.8.7</version>
    <executions>
        <execution>
            <id>prepare-agent</id>
            <goals>
                <goal>prepare-agent</goal>
            </goals>
        </execution>
        <execution>
            <goals>
                <goal>prepare-agent</goal>
            </goals>
        </execution>
        <execution>
            <id>report</id>
            <phase>prepare-package</phase>
            <goals>
                <goal>report</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

Listing A.3: PiTest Plugin

```
<plugin>
    <groupId>org.pitest</groupId>
    <artifactId>pitest-maven</artifactId>
    <version>1.7.3</version>
    <configuration>
        <outputFormats>csv,xml,html</outputFormats>
        <withHistory>true</withHistory>
    </configuration>
    <executions>
        <execution>
            <id>pit-report</id>
            <phase>test</phase>
            <goals>
                <goal>mutationCoverage</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

APPENDIX B

Single Metrics Histograms

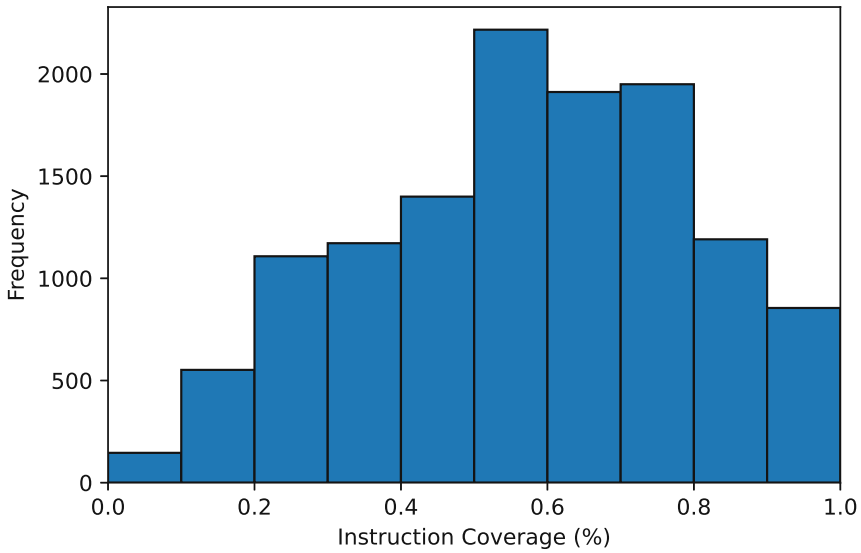

Figure B.1: Instruction Coverage Histogram.

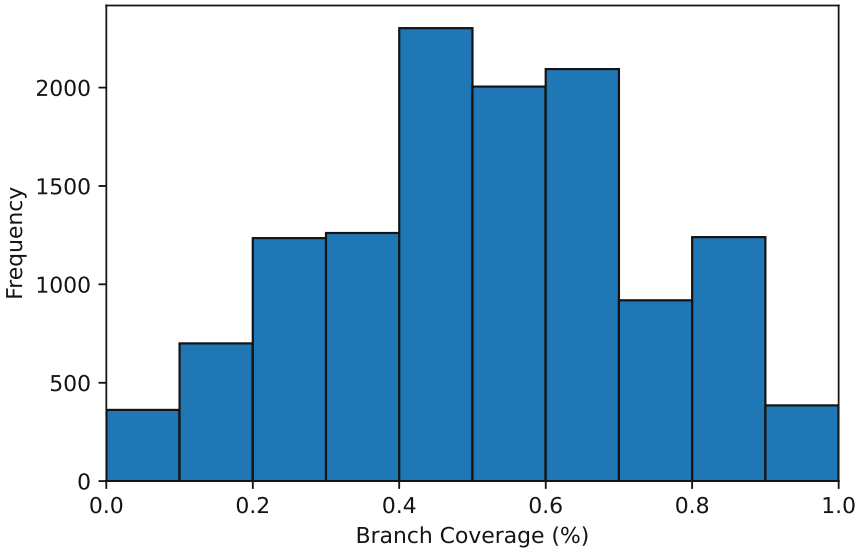

Figure B.2: Branch Coverage Histogram.

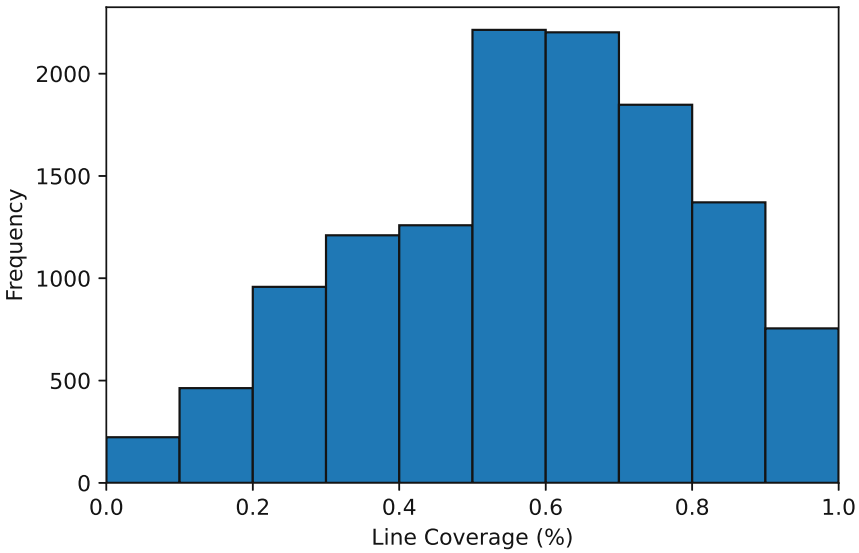

Figure B.3: Line Coverage Histogram.

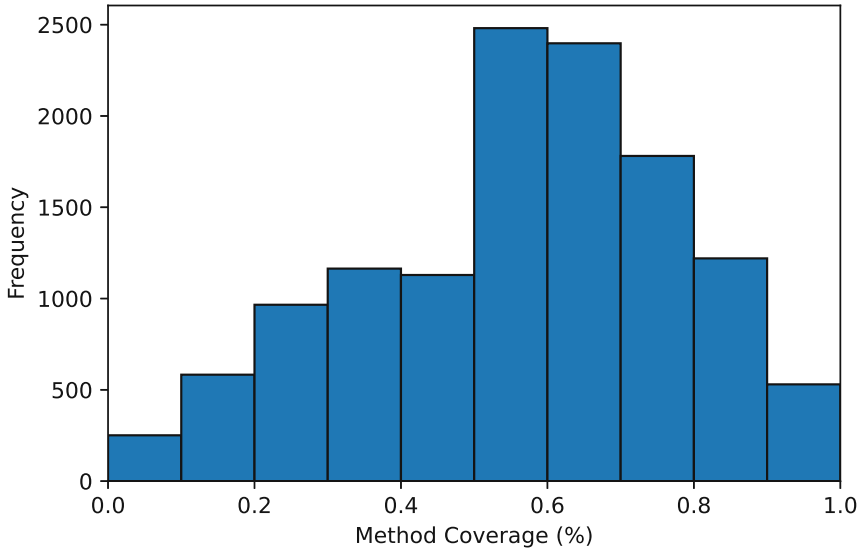

Figure B.4: Method Coverage Histogram.

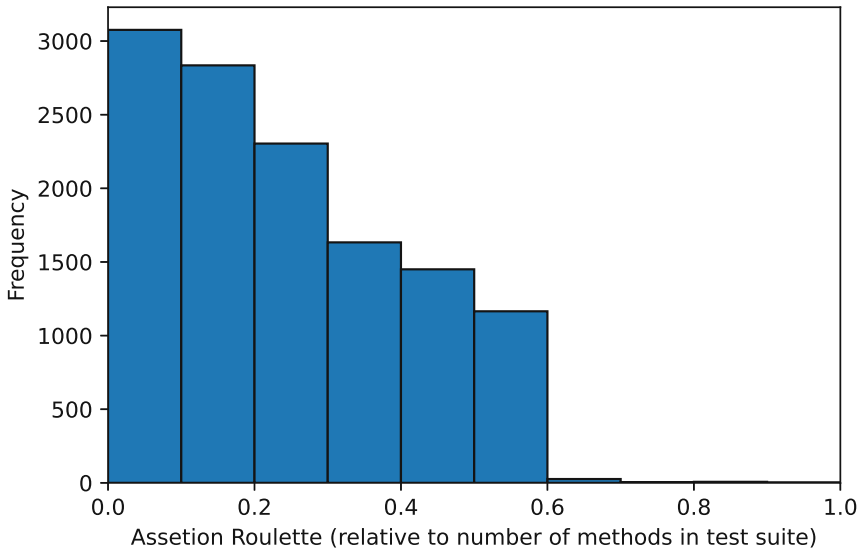

Figure B.5: Assertion Roulette Histogram.

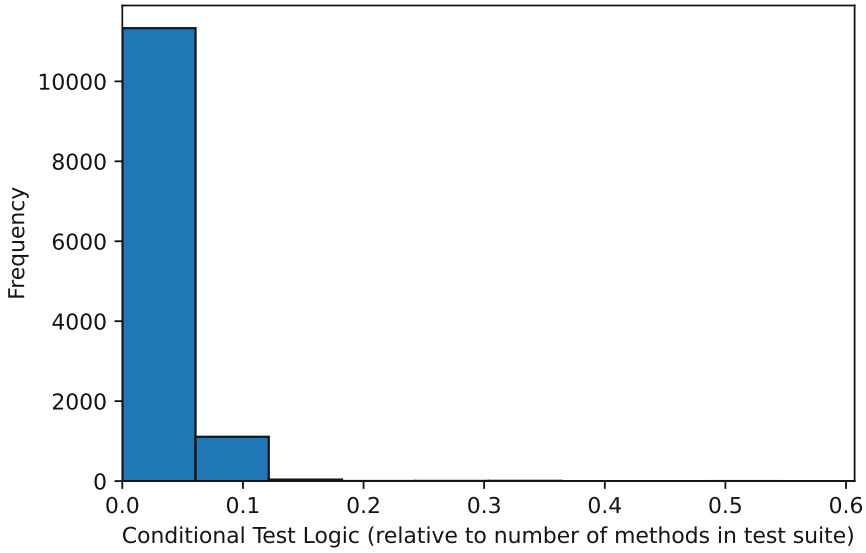

Figure B.6: Conditional Test Logic Histogram.

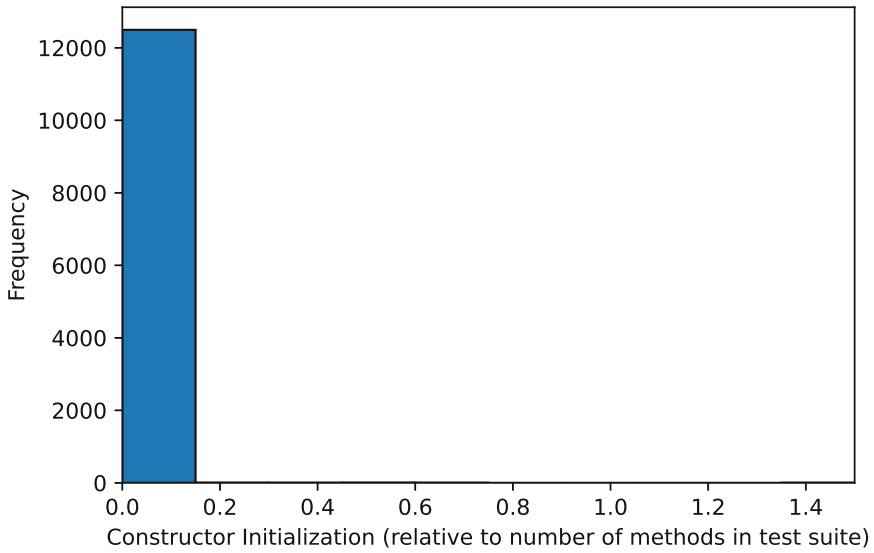

Figure B.7: Constructor Initialization Histogram.

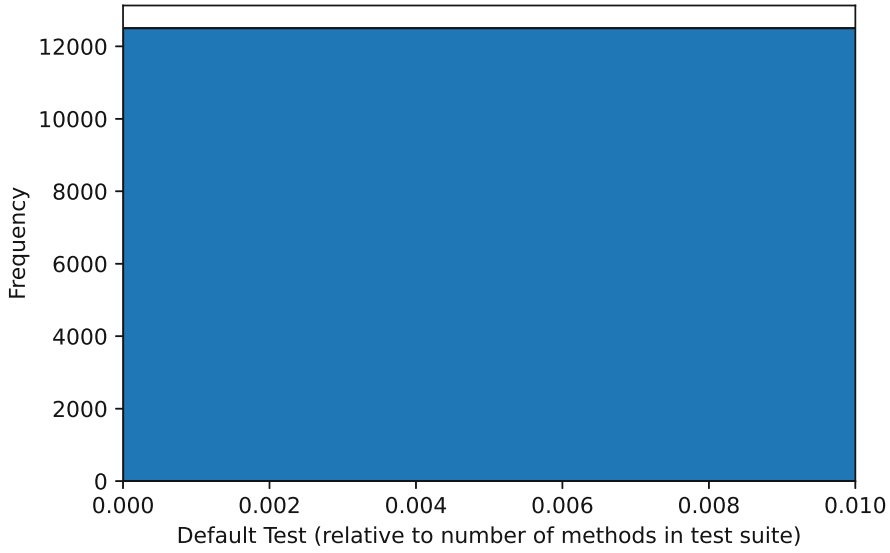

Figure B.8: Default Test Histogram.

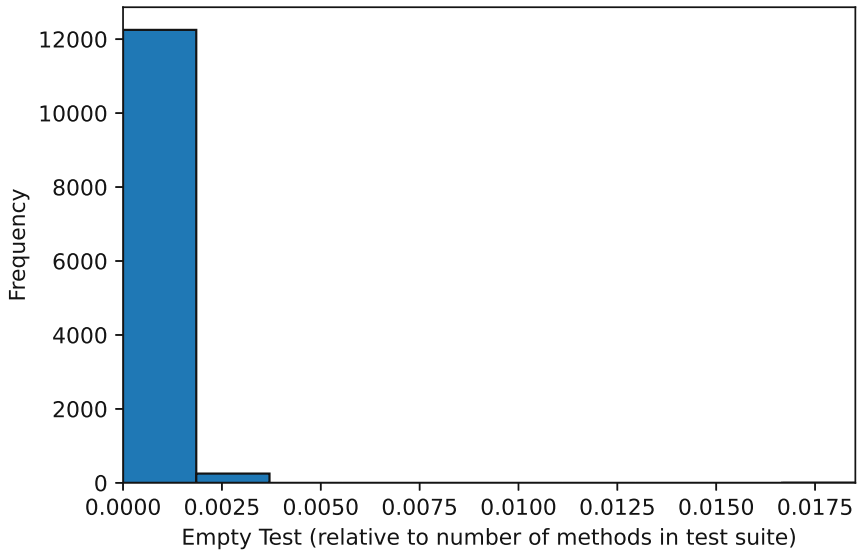

Figure B.9: Empty Test Histogram.

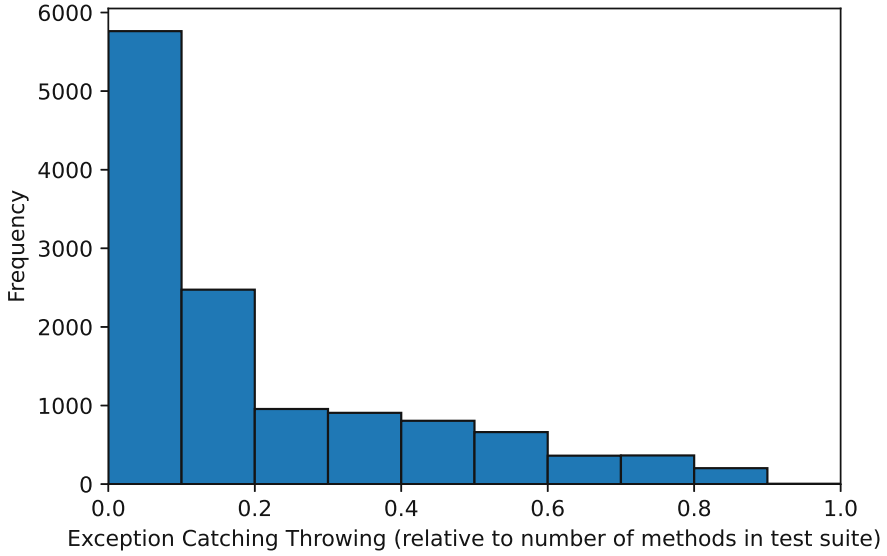

Figure B.10: Exception Catching Throwing Histogram.

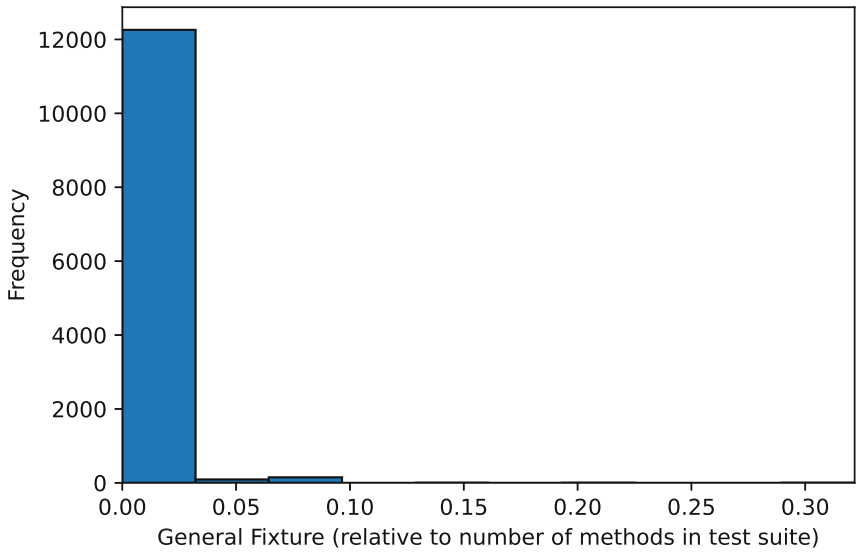

Figure B.11: General Fixture Histogram.

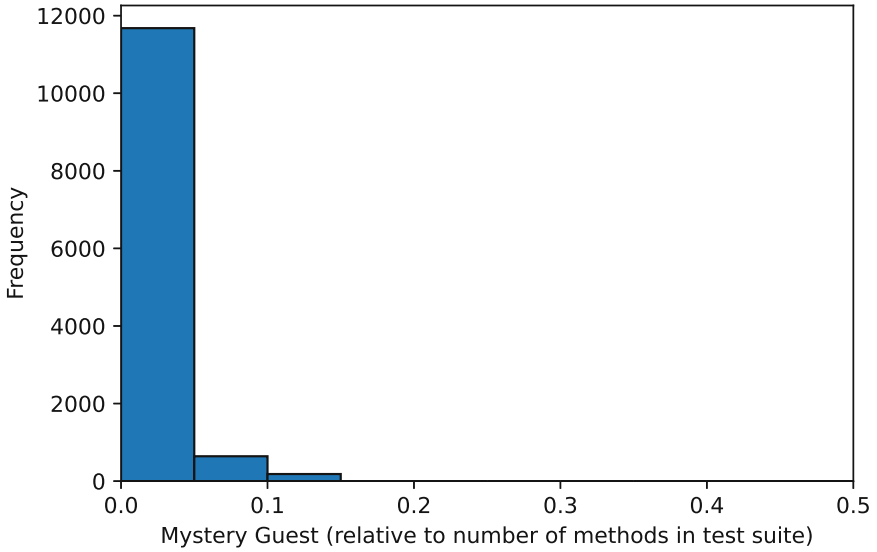

Figure B.12: Mystery Guest Histogram.

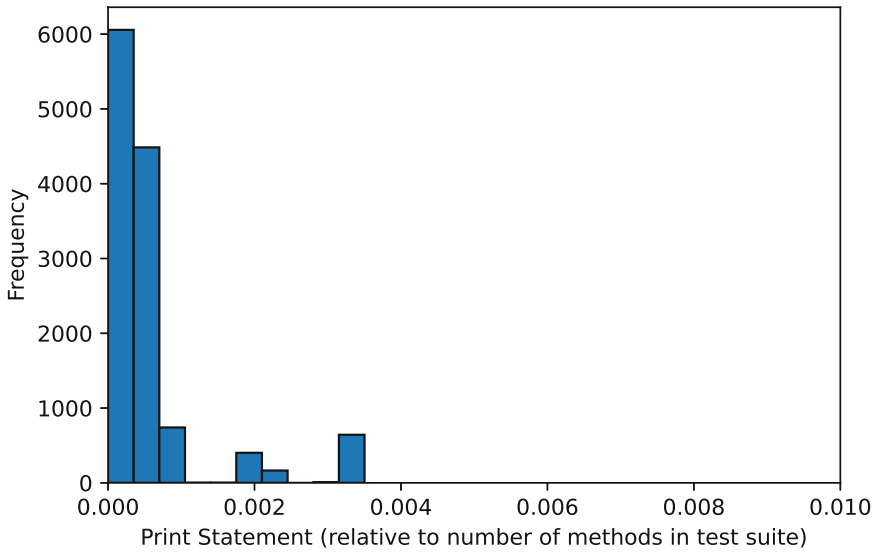

Figure B.13: Print Statement Histogram.

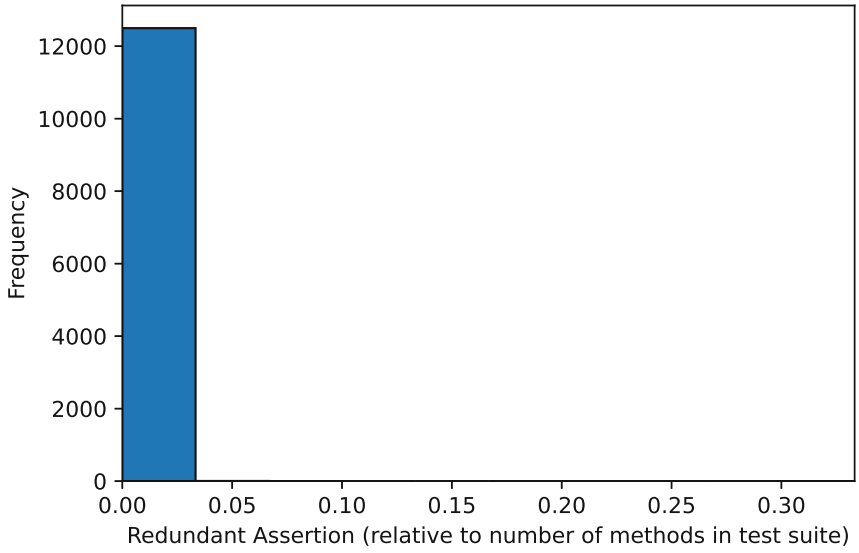

Figure B.14: Redundant Assertion Histogram.

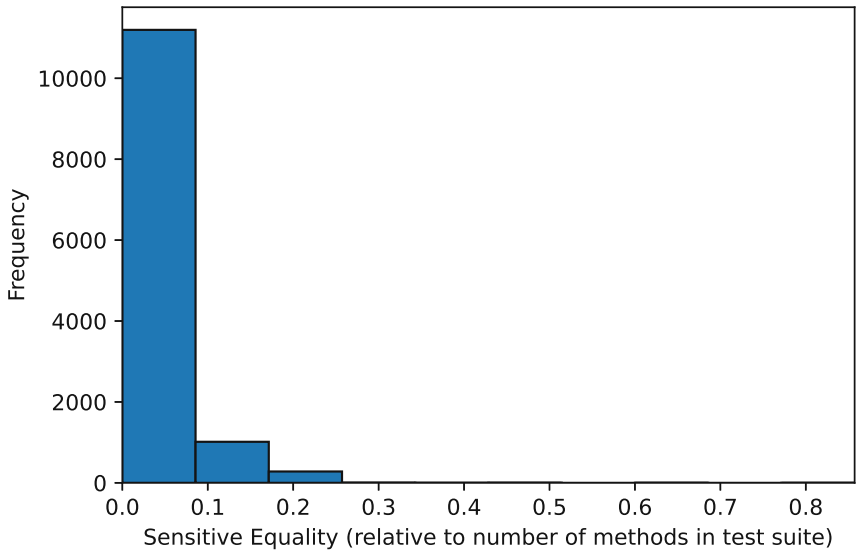

Figure B.15: Sensitive Equality Histogram.

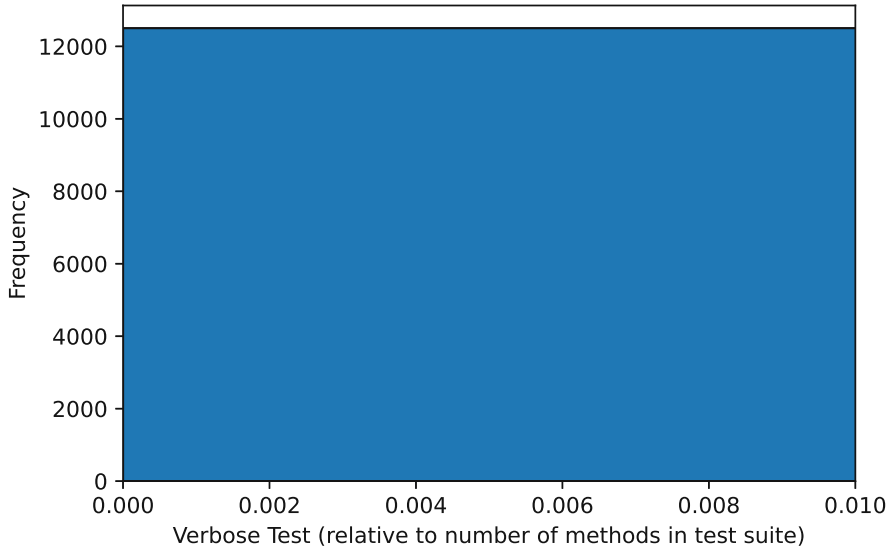

Figure B.16: Verbose Test Histogram.

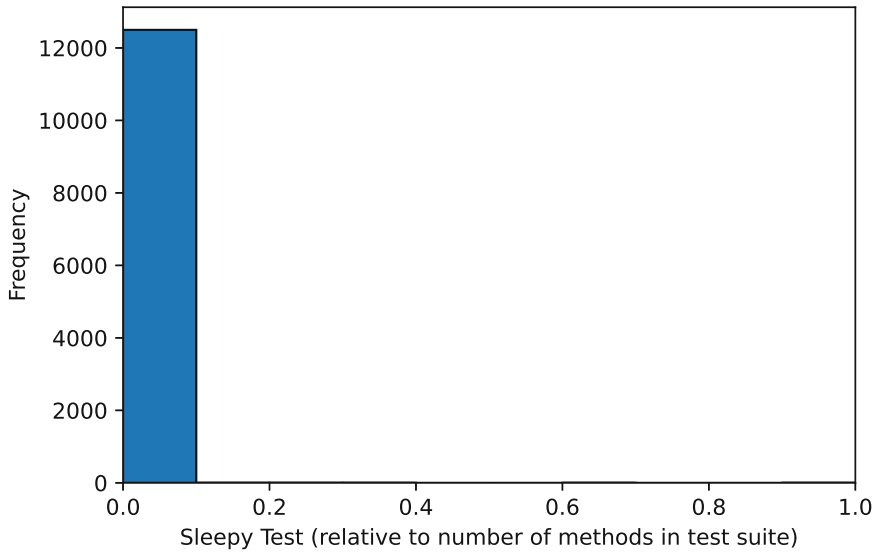

Figure B.17: Sleepy Test Histogram.

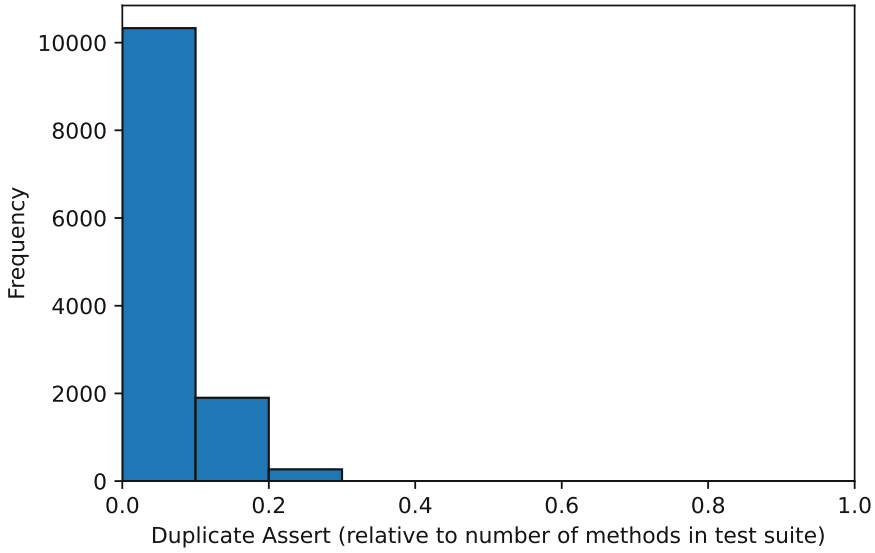

Figure B.18: Duplicate Assert Histogram.

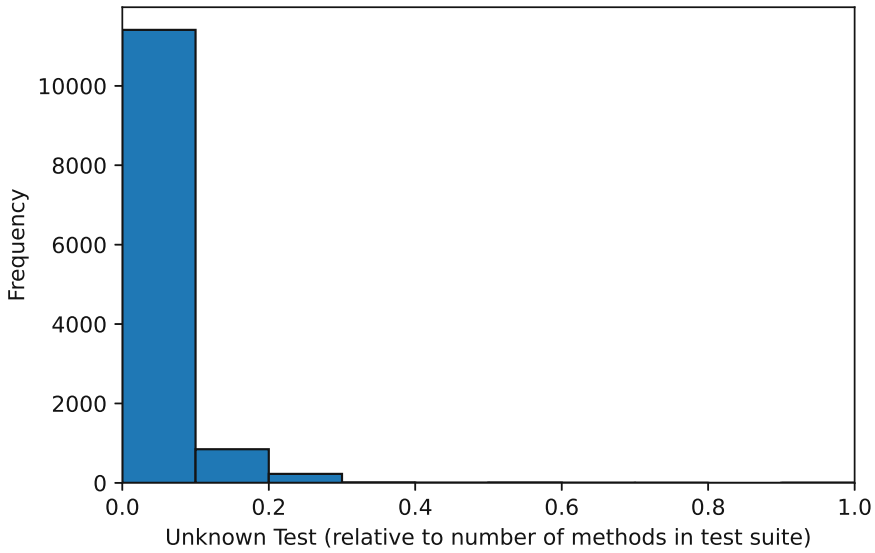

Figure B.19: Unknown Test Histogram.

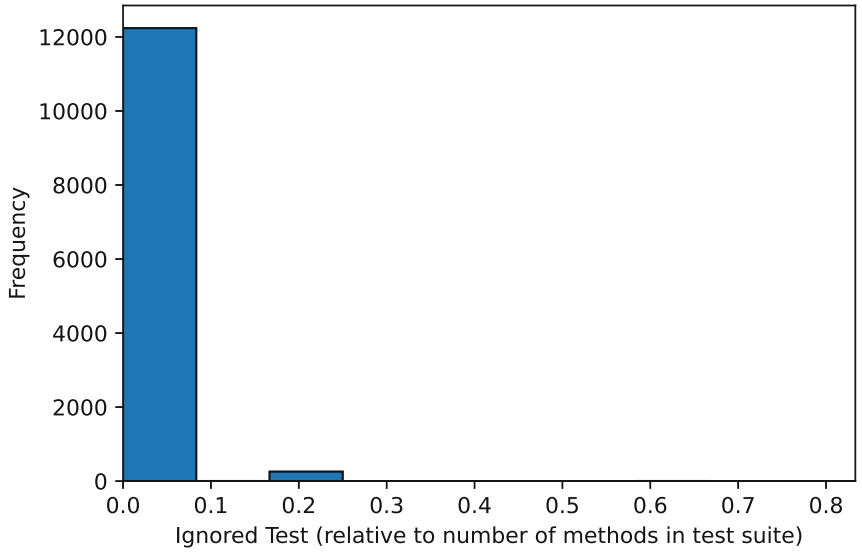

Figure B.20: Ignored Test Histogram.

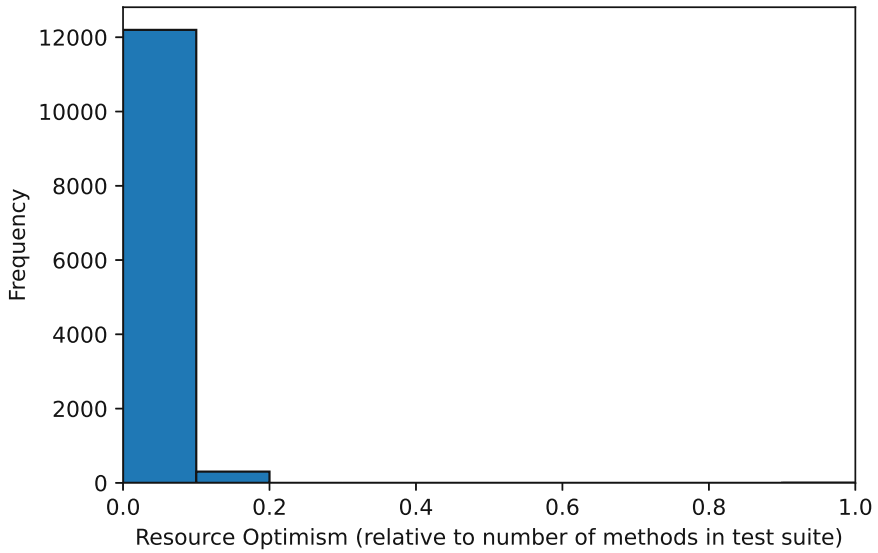

Figure B.21: Resource Optimism Histogram.

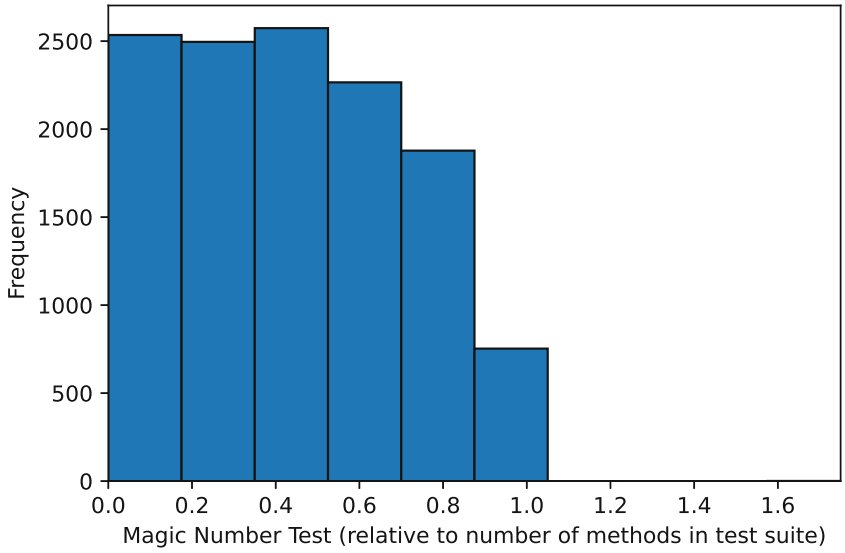

Figure B.22: Magic Number Test Histogram.

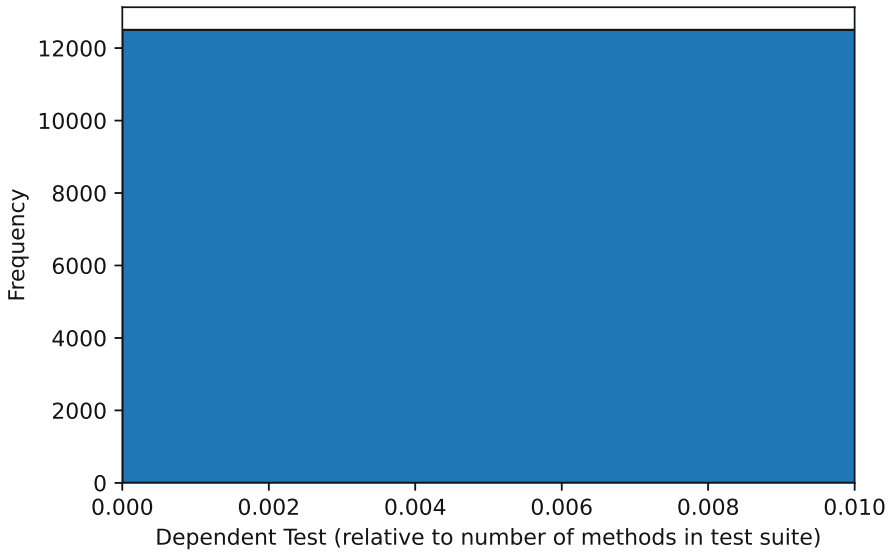

Figure B.23: Dependent Test Histogram.

List of Figures

List of Tables

Acronyms

ASE International Conference on Automated Software Engineering. 35

ICSE International Conference on Software Engineering. 35

ICSTW International Conference on Software Testing, Verification and Validation Workshops. 35

IEEE TSE IEEE Transactions on Software Engineering. 35

ISSTA Proceedings of the International Symposium on Software Testing and Analysis. 35

ISTQB International Software Testing Qualifications Board. 5

MAE Mean Absolute Error. 48, 50, 55, 57, 58, 62, 64, 65

MSE Mean Squared Error. 48, 50, 55–58, 62, 64, 65

RMSE Root Mean Squared Error. 48, 50, 55, 57, 58, 62, 64–66

Softw. Qual. J. Software Quality Journal. 35

SVM Support Vector Machine. 23, 65, 79, 80

Bibliography

References

[1] Iftekhar Ahmed et al. „Can testedness be effectively measured?“ In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 2016, pp. 547–558.

[2] Mamdouh Alenezi et al. *Test Suite Effectiveness: An Indicator for Open Source Software Quality, Open Source Software Computing (OSSCOM 2016)*. 2016.

[3] Wajdi Aljedaani et al. „Test Smell Detection Tools: A Systematic Mapping Study“. In: *arXiv e-prints* (2021), arXiv–2104.

[4] Ethem Alpaydin. *Machine learning*. MIT Press, 2014.

[5] Paul Ammann et al. *Introduction to software testing*. Cambridge University Press, 2016.

[6] Hilary Arksey et al. „Scoping studies: towards a methodological framework“. In: *International journal of social research methodology* 8.1 (2005), pp. 19–32.

[7] Dimitrios Athanasiou et al. „Test code quality and its relation to issue handling performance“. In: *IEEE Transactions on Software Engineering* 40.11 (2014), pp. 1100–1125.

[8] John O Awoyemi et al. „Credit card fraud detection using machine learning techniques: A comparative analysis“. In: *2017 international conference on computing networking and informatics (ICCNI)*. IEEE. 2017, pp. 1–9.

[9] Paco van Beckhoven et al. „Assessing Test Suite Effectiveness Using Static Metrics“. In: *CEUR Workshop Proc.* Vol. 2070. 2017, pp. 1–24.

[10] B.W. Boehm et al. *Characteristics of Software Quality*. Notas de Matematica. North-Holland Publishing Company, 1978. ISBN: 9780444851055.

[11] Barry W Boehm et al. „Quantitative evaluation of software quality“. In: *Proceedings of the 2nd international conference on Software engineering*. 1976, pp. 592–605.

[12] Leo Breiman. „Random forests“. In: *Machine learning* 45.1 (2001), pp. 5–32.

[13] Thyago P Carvalho et al. „A systematic literature review of machine learning methods applied to predictive maintenance“. In: *Computers & Industrial Engineering* 137 (2019), p. 106024.

[14] Thierry Titcheu Chekam et al. „An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption“. In: *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE. 2017, pp. 597–608.

[15] Henry Coles et al. „Pit: a practical mutation testing tool for java“. In: *Proceedings of the 25th international symposium on software testing and analysis*. 2016, pp. 449–452.

[16] S Devi et al. „A comprehensive survey on autonomous driving cars: A perspective view“. In: *Wireless Personal Communications* 114.3 (2020), pp. 2121–2133.

[17] Yadolah Dodge. *The concise encyclopedia of statistics*. Springer Science & Business Media, 2008.

[18] Norman R Draper et al. *Applied regression analysis*. Vol. 326. John Wiley & Sons, 1998.

[19] Vinicius HS Durelli et al. „Machine learning applied to software testing: A systematic mapping study“. In: *IEEE Transactions on Reliability* 68.3 (2019), pp. 1189–1212.

[20] Tore Dyba et al. „Applying systematic reviews to diverse study types: An experience report“. In: *First international symposium on empirical software engineering and measurement (ESEM 2007)*. IEEE. 2007, pp. 225–234.

[21] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[22] Dorothy Graham et al. *Foundations of software testing*. Cengage, 2019.

[23] Giovanni Grano et al. „How high will it be? using machine learning models to predict branch coverage in automated testing“. In: *2018 IEEE workshop on machine learning techniques for software quality evaluation (MaLTeSQuE)*. IEEE. 2018, pp. 19–24.

[24] Lorena Gutiérrez-Madronal et al. „Mutation testing: Guideline and mutation operator classification“. In: *ICCGI 2014* (2014), p. 184.

[25] Jiangang Hao et al. „Machine learning made easy: a review of scikit-learn package in python programming language“. In: *Journal of Educational and Behavioral Statistics* 44.3 (2019), pp. 348–361.

[26] Hadi Hemmati. „How effective are code coverage criteria?“ In: *2015 IEEE International Conference on Software Quality, Reliability and Security*. IEEE. 2015, pp. 151–156.

[27] Arthur E Hoerl et al. „Ridge regression: Biased estimation for nonorthogonal problems“. In: *Technometrics* 12.1 (1970), pp. 55–67.

[28] Laura Inozemtseva et al. „Coverage is not strongly correlated with test suite effectiveness“. In: *Proceedings of the 36th international conference on software engineering*. 2014, pp. 435–445.

[29] *ISO/IEC 9126:1991*. [Online; accessed 7. Aug. 2022]. Aug. 2022. URL: https://www.iso.org/standard/16722.html.

[30] Marko Ivanković et al. „Code coverage at Google“. In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2019, pp. 955–963.

[31] Kevin Jalbert et al. „Predicting mutation score using source code and test suite metrics“. In: *2012 First International Workshop on Realizing AI Synergies in Software Engineering (RAISE)*. IEEE. 2012, pp. 42–46.

[32] Taeho Jo. *Machine Learning Foundations*. Springer, 2021.

[33] René Just et al. „Defects4J: A database of existing faults to enable controlled testing studies for Java programs“. In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. 2014, pp. 437–440.

[34] Stephen H Kan. *Metrics and models in software quality engineering*. Addison-Wesley Professional, 2003.

[35] Amandeep Kaur. „A systematic literature review on empirical analysis of the relationship between code smells and software quality attributes“. In: *Archives of Computational Methods in Engineering* 27.4 (2020), pp. 1267–1296.

[36] Staffs Keele et al. *Guidelines for performing systematic literature reviews in software engineering*. Tech. rep. Citeseer, 2007.

[37] Barbara Kitchenham et al. „A systematic review of systematic review process research in software engineering“. In: *Information and software technology* 55.12 (2013), pp. 2049–2075.

[38] Barbara Kitchenham et al. „The educational value of mapping studies of software engineering literature“. In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. 2010, pp. 589–598.

[39] Miguel A Laguna et al. „A systematic mapping study on software product line evolution: From legacy system reengineering to product line refactoring“. In: *Science of Computer Programming* 78.8 (2013), pp. 1010–1034.

[40] Huan Lin et al. „Domain-RIP Analysis: A Technique for Analyzing Mutation Stubbornness“. In: *IEEE Access* 7 (2018), pp. 4006–4023.

[41] Richard J Lipton. *Fault diagnosis of computer programs*. 1971.

[42] Cuauhtemoc Lopez-Martin. „Machine learning techniques for software testing effort prediction“. In: *Software Quality Journal* 30.1 (2022), pp. 65–100.

[43] Yu-Seung Ma et al. „MuJava: a mutation system for Java“. In: *Proceedings of the 28th international conference on Software engineering*. 2006, pp. 827–830.

[44] Dastan Maulud et al. „A review on linear regression comprehensive in machine learning“. In: *Journal of Applied Science and Technology Trends* 1.4 (2020), pp. 140–147.

[45] Jim A McCall et al. *Factors in software quality. volume i. concepts and definitions of software quality*. Tech. rep. GENERAL ELECTRIC CO SUNNYVALE CA, 1977.

[46] Joan C Miller et al. „Systematic mistake analysis of digital computer programs“. In: *Communications of the ACM* 6.2 (1963), pp. 58–63.

[47] Jamie L Mitchell et al. *Advanced Software Testing-Vol. 3: Guide to the ISTQB Advanced Certification as an Advanced Technical Test Analyst*. Rocky Nook, Inc., 2015.

[48] Raymond H Myers et al. *Generalized linear models: with applications in engineering and the sciences*. John Wiley & Sons, 2012.

[49] Nachiappan Nagappan. *A software testing and reliability early warning (STREW) metric suite*. North Carolina State University, 2005.

[50] Maryam Navaei et al. „Machine Learning in Software Development Life Cycle: A Comprehensive Review.“ In: *ENASE* (2022), pp. 344–354.

[51] Quang Vu Nguyen et al. „Problems of mutation testing and higher order mutation testing“. In: *Advanced computational methods for knowledge engineering*. Springer, 2014, pp. 157–172.

[52] Mike Papadakis et al. „Mutation testing advances: an analysis and survey“. In: *Advances in Computers*. Vol. 112. Elsevier, 2019, pp. 275–378.

[53] F. Pedregosa et al. „Scikit-learn: Machine Learning in Python“. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[54] Anthony Peruma et al. „TsDetect: An Open Source Test Smells Detection Tool“. In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2020. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 1650–1654. ISBN: 9781450370431.

[55] Kai Petersen et al. „Guidelines for conducting systematic mapping studies in software engineering: An update“. In: *Information and Software Technology* 64 (2015), pp. 1–18.

[56] Kai Petersen et al. „Systematic mapping studies in software engineering“. In: *12th International Conference on Evaluation and Assessment in Software Engineering (EASE) 12*. 2008, pp. 1–10.

[57] Goran Petrović et al. „Does mutation testing improve testing practices?“ In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE. 2021, pp. 910–921.

[58] Mark Petticrew et al. *Systematic reviews in the social sciences: A practical guide*. John Wiley & Sons, 2008.

[59] Derek A Pisner et al. „Support vector machine“. In: *Machine learning*. Elsevier, 2020, pp. 101–121.

[60] Alessandro Viola Pizzoleto et al. „A systematic literature review of techniques and metrics to reduce the cost of mutation testing“. In: *Journal of Systems and Software* 157 (2019), p. 110388.

[61] Claude Sammut et al. *Encyclopedia of machine learning*. Springer Science & Business Media, 2011.

[62] Arthur L. Samuel. „Some studies in machine learning using the game of Checkers“. In: *IBM JOURNAL OF RESEARCH AND DEVELOPMENT* (1959), pp. 71–105.

[63] Mochamad Chandra Saputra et al. „Proposal of a Method to Measure Test Suite Quality Attributes for White-Box Testing“. In: *International Journal of Advanced Computer Science and Applications* 12.5 (2021).

[64] Lilian Passos Scatalon et al. „Software testing in introductory programming courses: A systematic mapping study“. In: *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. 2019, pp. 421–427.

[65] Saad Shafiq et al. „Machine learning for software engineering: A systematic mapping“. In: *arXiv preprint arXiv:2005.13299* (2020).

[66] Xavier Solé et al. „Evaluation of random forests on large-scale classification problems using a bag-of-visual-words representation“. In: *Artificial Intelligence Research and Development*. IOS Press, 2014, pp. 273–276.

[67] Érica Ferreira de Souza et al. „Knowledge management initiatives in software testing: A mapping study“. In: *Information and Software Technology* 57 (2015), pp. 378–391.

[68] III Standard. „Software Engineering-Product quality-Part 1: Quality model“. In: *ISO Standard* (2001), pp. 9126–1.

[69] Joanna Strug et al. „Machine learning approach in mutation testing“. In: *IFIP International Conference on Testing Software and Systems*. Springer. 2012, pp. 200–214.

[70] Robert Tibshirani. „Regression shrinkage and selection via the lasso“. In: *Journal of the Royal Statistical Society: Series B (Methodological)* 58.1 (1996), pp. 267–288.

[71] Stefan Wagner. *Software Product Quality Control*. SpringerLink : Bücher. Springer Berlin Heidelberg, 2013. ISBN: 9783642385711.

[72] Stefan Wagner et al. „Operationalised product quality models and assessment: The Quamoco approach“. In: *Information and Software Technology* 62 (2015), pp. 101–123.

[73] Neil Walkinshaw et al. „Uncertainty-driven black-box test data generation“. In: *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE. 2017, pp. 253–263.

[74] Roel J Wieringa. *Design science methodology for information systems and software engineering*. Springer, 2014.

[75] Claes Wohlin et al. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[76] Claes Wohlin et al. „On the reliability of mapping studies in software engineering“. In: *Journal of Systems and Software* 86.10 (2013), pp. 2594–2610.

[77] Qian Yang et al. „A survey of coverage-based testing tools“. In: *The Computer Journal* 52.5 (2009), pp. 589–597.

[78] Jie Zhang et al. „Predictive mutation testing“. In: *IEEE Transactions on Software Engineering* 45.9 (2018), pp. 898–918.

[79] Zhi-Hua Zhou. *Machine learning*. Springer Nature Singapore, 2021.

[80] Hong Zhu et al. „Software unit test coverage and adequacy“. In: *ACM computing surveys (CSUR)* 29.4 (1997), pp. 366–427.

[81] Hui Zou et al. „Regularization and variable selection via the elastic net“. In: *Journal of the royal statistical society: series B (statistical methodology)* 67.2 (2005), pp. 301–320.

Web Links

[82] *About ACM DL*. [Online; accessed 13. Jan. 2022]. Jan. 2022. URL: https://dl.acm.org/about.

[83] *About IEEE Xplore*. [Online; accessed 13. Jan. 2022]. Dec. 2021. URL: https://ieeexplore.ieee.org/Xplorehelp/overview-of-ieee-xplore/about-ieee-xplore.

[84] *ACM Digital Library Search Results*. URL: https://dl.acm.org/action/doSearch?AllField=AllField%3A%28%22software+testing%22%29+AND+%28AllField%3A%28%22test+suite+evaluation%22%29+OR+AllField%3A%28%22test+suite+assessment%22%29+OR+AllField%3A%28%22test+suite+quality%22%29%29+AND+%28AllField%3A%28%22metric%22%29+OR+AllField%3A%28%22score%22%29%29&expand=dl&startPage=1&sortBy=Ppub_desc&pageSize=50.

[85] *CodeCover - an open-source glass-box testing tool*. [Online; accessed 21. Apr. 2022]. Jan. 2012. URL: http://codecover.org.

[86] *CORE Rankings Portal*. [Online; accessed 25. Mar. 2023]. Mar. 2023. URL: https://www.core.edu.au/conference-portal.

[87] Elsevier. *Content - How Scopus Works - Scopus - | Elsevier solutions*. [Online; accessed 13. Jan. 2022]. Jan. 2022. URL: https://www.elsevier.com/solutions/scopus/how-scopus-works/content.

[88] Elsevier. *Engineering Websites Index & Journals Database*. [Online; accessed 25. Mar. 2023]. Mar. 2023. URL: https://www.elsevier.com/solutions/engineering-village/databases.

[89] Elsevier. *What is Scopus about? - Scopus: Access and use Support Center*. [Online; accessed 13. Jan. 2022]. Jan. 2022. URL: https://service.elsevier.com/app/answers/detail/a_id/15100/supporthub/scopus.

[90] Elsevier. *What is Scopus Preview? - Scopus: Access and use Support Center*. [Online; accessed 13. Jan. 2022]. Jan. 2022. URL: https://service.elsevier.com/app/answers/detail/a_id/15534/supporthub/scopus/#tips.

[91] *IEEE Xplore Search Results*. URL: https://ieeexplore.ieee.org/search/searchresult.jsp?action=search&matchBoolean=true&newsearch=true&queryText=((%22All%20Metadata%22:%22software%20testing%22)%20AND%20(%22All%20Metadata%22:%22test%20suite%20evaluation%22%20OR%20%22All%20Metadata%22:%22test%20suite%20assessment%22%20OR%20%22All%20Metadata%22:%22test%20suite%20quality%22)%20AND%20(%22All%20Metadata%22:%22metric%22%20OR%20%22All%20Metadata%22:%22score%22)).

[92] *ISO 9000:2015(en), Quality management systems — Fundamentals and vocabulary*. [Online; accessed 23. Jun. 2022]. June 2022. URL: https://www.iso.org/obp/ui/#iso:std:iso:9000:ed-4:v1:en.

[93] *ISO/IEC 25010:2011(en), Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models*. [Online; accessed 23. Jun. 2022]. June 2022. URL: https://www.iso.org/standard/35733.html.

[94] *ISO/IEC/IEEE 24765:2017(en), Systems and software engineering — Vocabulary*. [Online; accessed 23. Jun. 2022]. June 2022. URL: https://www.iso.org/obp/ui/#iso:std:iso-iec-ieee:24765:ed-2:v1:en.

[95] *ISTQB Glossary*. [Online; accessed 18. Jul. 2022]. July 2022. URL: https://glossary.istqb.org/.

[96] *Linear Models*. [Online; accessed 13. Jun. 2022]. June 2022. URL: https://scikit-learn.org/stable/modules/linear_model.html.

[97] *Mutation operators*. [Online; accessed 22. May 2022]. May 2022. URL: https://pitest.org/quickstart/mutators/#CONDITIONALS_BOUNDARY.

[98] *Scopus Preview Search Results*. URL: https://www.scopus.com/results/results.uri?sort=plf-f&src=s&sid=7578194155ccb01446dae27837798981&sot=a&sdt=a&sl=127&s=%28%22software+testing%22%29+AND+%28%22test+suite+evaluation%22+OR+%22test+suite+assessment%22+OR+%22test+suite+quality%22%29+AND+%28%22metric%22+OR+%22score%22%29&origin=searchadvanced&editSaveSearch=&txGid=cfca3a976b4935339e7495f15cdd2978.

Mapping Study Publications

[99] Kalle Aaltonen et al. „Mutation analysis vs. code coverage in automated assessment of students' testing skills". In: *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion - SPLASH '10*. ACM Press, 2010.

[100] Alireza Aghamohammadi et al. „Statement frequency coverage: A code coverage criterion for assessing test suite effectiveness". In: *Information and Software Technology* 129 (Jan. 2021), p. 106426.

[101] Iftekhar Ahmed et al. „Can testedness be effectively measured?" In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, Nov. 2016.

[102] Lucas Andrade et al. „Can operational profile coverage explain post-release bug detection?" In: *Software Testing, Verification and Reliability* 30.4-5 (June 2020).

[103] Thomas Bach et al. „The Impact of Coverage on Bug Density in a Large Industrial Software Project". In: *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, Nov. 2017.

[104] David Bowes et al. „How Good Are My Tests?" In: *2017 IEEE/ACM 8th Workshop on Emerging Trends in Software Metrics (WETSoM)*. IEEE, May 2017.

[105] Jeremy S. Bradbury et al. „An empirical framework for comparing effectiveness of testing and property-based formal analysis". In: *ACM SIGSOFT Software Engineering Notes* 31.1 (Jan. 2006), pp. 2–5.

[106] David Bingham Brown et al. „The care and feeding of wild-caught mutants". In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, Aug. 2017.

[107] Marc Brunink et al. „Using Branch Frequency Spectra to Evaluate Operational Coverage". In: *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, Dec. 2017.

[108] Benjamin Simon Clegg et al. „An Empirical Study to Determine if Mutants Can Effectively Simulate Students' Programming Mistakes to Increase Tutors' Confidence in Autograding". In: *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*. ACM, Mar. 2021.

[109] Lucas Cordova et al. „A Comparison of Inquiry-Based Conceptual Feedback vs. Traditional Detailed Feedback Mechanisms in Software Testing Education: An Empirical Investigation". In: *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*. ACM, Mar. 2021.

[110] Said Daoudagh et al. „Assessment of Access Control Systems Using Mutation Testing". In: *2015 IEEE/ACM 1st International Workshop on TEchnical and LEgal aspects of data pRivacy and SEcurity*. IEEE, May 2015.

[111] Stephen H. Edwards et al. „Comparing test quality measures for assessing student-written tests“. In: *Companion Proceedings of the 36th International Conference on Software Engineering*. ACM, May 2014.

[112] Camilo Escobar-Velasquez et al. „MutAPK: Source-Codeless Mutant Generation for Android Apps“. In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, Nov. 2019.

[113] Kassem Fawaz et al. „PBCOV: a property-based coverage criterion“. In: *Software Quality Journal* 23.1 (May 2014), pp. 171–202.

[114] Hermann Felbinger et al. „Empirical study of correlation between mutation score and model inference based test suite adequacy assessment“. In: *Proceedings of the 11th International Workshop on Automation of Software Test - AST '16*. ACM Press, 2016.

[115] Hermann Felbinger et al. „Mutation Score, Coverage, Model Inference: Quality Assessment for T-Way Combinatorial Test-Suites“. In: *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, Mar. 2017.

[116] Andreas Fellner et al. „Model-based, mutation-driven test case generation via heuristic-guided branching search“. In: *Proceedings of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design*. ACM, Sept. 2017.

[117] Tamás Gergely et al. „Analysis of Static and Dynamic Test-to-code Traceability Information“. In: *Acta Cybernetica* 23.3 (2018), pp. 903–919.

[118] Tamás Gergely et al. „Differences between a static and a dynamic test-to-code traceability recovery method“. In: *Software Quality Journal* 27.2 (Dec. 2018), pp. 797–822.

[119] Milos Gligoric et al. „Comparing non-adequate test suites using coverage criteria“. In: *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. ACM, July 2013.

[120] Milos Gligoric et al. „Guidelines for Coverage-Based Comparisons of Non-Adequate Test Suites“. In: *ACM Transactions on Software Engineering and Methodology* 24.4 (Sept. 2015), pp. 1–33.

[121] Milos Gligoric et al. „Mutation testing meets approximate computing“. In: *2017 IEEE/ACM 39th International Conference on Software Engineering: New Ideas and Emerging Technologies Results Track (ICSE-NIER)*. IEEE, May 2017.

[122] Ariel Godio et al. „Efficient Test Generation Guided by Field Coverage Criteria“. In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, Nov. 2019.

[123] Pablo Gómez-Abajo et al. „Systematic Engineering of Mutation Operators.“ In: *The Journal of Object Technology* 19.3 (2020), 3:1.

[124] Rahul Gopinath et al. „Code coverage for suite evaluation by developers“. In: *Proceedings of the 36th International Conference on Software Engineering*. ACM, May 2014.

[125] Rahul Gopinath et al. „Does choice of mutation tool matter?“ In: *Software Quality Journal* 25.3 (May 2016), pp. 871–920.

[126] Rahul Gopinath et al. „Mutation Reduction Strategies Considered Harmful“. In: *IEEE Transactions on Reliability* 66.3 (Sept. 2017), pp. 854–874.

[127] Alex Groce et al. „An extensible, regular-expression-based tool for multi-language mutant generation“. In: *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ACM, May 2018.

[128] Alex Groce et al. „Coverage and Its Discontents“. In: *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. ACM, Oct. 2014.

[129] M. Harder et al. „Improving test suites via operational abstraction“. In: *25th International Conference on Software Engineering, 2003. Proceedings.* IEEE, 2003.

[130] Fabrice Harel-Canada et al. „Is neuron coverage a meaningful measure for testing deep neural networks?“ In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Nov. 2020.

[131] Mohammad Mahdi Hassan et al. „Comparing Multi-Point Stride Coverage and dataflow coverage“. In: *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, May 2013.

[132] Michael Hilton et al. „A large-scale study of test coverage evolution“. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, Sept. 2018.

[133] Dominik Holling et al. „Nequivack: Assessing Mutation Score Confidence“. In: *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, Apr. 2016.

[134] Ferenc Horváth et al. „Test suite evaluation using code coverage based metrics“. In: (2015).

[135] Chen Huo et al. „Interpreting Coverage Information Using Direct and Indirect Coverage“. In: *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, Apr. 2016.

[136] Laura Inozemtseva et al. „Coverage is not strongly correlated with test suite effectiveness“. In: *Proceedings of the 36th International Conference on Software Engineering*. ACM, May 2014.

[137] Laura Inozemtseva et al. „Using fault history to improve mutation reduction“. In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013*. ACM Press, 2013.

[138] René Just et al. „Are mutants a valid substitute for real faults in software testing?“ In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, Nov. 2014.

[139] René Just et al. „Efficient mutation analysis by propagating and partitioning infected execution states“. In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis - ISSTA 2014*. ACM Press, 2014.

[140] Jeshua S. Kracht et al. „Empirically Evaluating the Quality of Automatically Generated and Manually Written Test Suites“. In: *2014 14th International Conference on Quality Software*. IEEE, Oct. 2014.

[141] Birgitta Lindstrom et al. „On strong mutation and subsuming mutants“. In: *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, Apr. 2016.

[142] Claudio Magalhães et al. „Evaluating an Automatic Text-based Test Case Selection using a Non-Instrumented Code Coverage Analysis“. In: *Proceedings of the 2nd Brazilian Symposium on Systematic and Automated Software Testing - SAST*. ACM Press, 2017.

[143] Claudio Magalhães et al. „HSP: A hybrid selection and prioritisation of regression test cases based on information retrieval and code coverage applied on an industrial case study“. In: *Journal of Systems and Software* 159 (Jan. 2020), p. 110430.

[144] Pedro Reales Mateo et al. „Validating Second-Order Mutation at System Level“. In: *IEEE Transactions on Software Engineering* 39.4 (Apr. 2013), pp. 570–587.

[145] Phil McMinn et al. „Automatic Detection and Removal of Ineffective Mutants for the Mutation Analysis of Relational Database Schemas“. In: *IEEE Transactions on Software Engineering* 45.5 (May 2019), pp. 427–463.

[146] Mehdi Mirzaaghaei et al. „DOM-based test adequacy criteria for web applications“. In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis - ISSTA 2014*. ACM Press, 2014.

[147] Ali Parsai et al. „A Model to Estimate First-Order Mutation Coverage from Higher-Order Mutation Coverage“. In: *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, Aug. 2016.

[148] Ali Parsai et al. „Evaluating random mutant selection at class-level in projects with non-adequate test suites“. In: *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*. ACM, June 2016.

[149] Alexandre Perez et al. „A Test-Suite Diagnosability Metric for Spectrum-Based Fault Localization Approaches“. In: *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, May 2017.

[150] M. Popovic et al. „Test case generation for the task tree type of architecture“. In: *Information and Software Technology* 52.6 (June 2010), pp. 697–706.

[151] Ajitha Rajan et al. „The effect of program and model structure on mc/dc test adequacy coverage“. In: *Proceedings of the 13th international conference on Software engineering - ICSE '08*. ACM Press, 2008.

[152] Brian Robinson et al. „Scaling up automated test generation: Automatically generating maintainable regression unit tests for programs“. In: *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, Nov. 2011.

[153] Jose Miguel Rojas et al. „Code Defenders: Crowdsourcing Effective Tests and Subtle Mutants with a Mutation Testing Game“. In: *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, May 2017.

[154] David Schuler et al. „Checked coverage: an indicator for oracle quality“. In: *Software Testing, Verification and Reliability* 23.7 (May 2013), pp. 531–551.

[155] Zalia Shams et al. „Checked Coverage and Object Branch Coverage“. In: *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. ACM, Feb. 2015.

[156] Elena Sherman et al. „Saturation-based testing of concurrent programs“. In: *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering on European software engineering conference and foundations of software engineering symposium - ESEC/FSE '09*. ACM Press, 2009.

[157] Khashayar Etemadi Someoliayi et al. „Program State Coverage: A Test Coverage Metric Based on Executed Program States“. In: *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, Feb. 2019.

[158] Beatriz Souza et al. „A Large Scale Study On the Effectiveness of Manual and Automatic Unit Test Generation“. In: *Proceedings of the 34th Brazilian Symposium on Software Engineering*. ACM, Oct. 2020.

[159] Francisco Carlos M. Souza et al. „Strong mutation-based test data generation using hill climbing“. In: *Proceedings of the 9th International Workshop on Search-Based Software Testing*. ACM, May 2016.

[160] Davide Spadini et al. „On the Relation of Test Smells to Software Code Quality“. In: *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, Sept. 2018.

[161] Krystof Sykora et al. „Code Coverage Aware Test Generation Using Constraint Solver“. In: *Software Engineering and Formal Methods. SEFM 2020 Collocated Workshops*. Springer International Publishing, 2021, pp. 58–66.

[162] David Tengeri et al. „Beyond code coverage – An approach for test suite assessment and improvement“. In: *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, Apr. 2015.

[163] David Tengeri et al. „Relating Code Coverage, Mutation Score and Test Suite Reducibility to Defect Density“. In: *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, Apr. 2016.

[164] Ayse Tosun et al. „On the effectiveness of unit tests in test-driven development“. In: *Proceedings of the 2018 International Conference on Software and System Process*. ACM, May 2018.

[165] Sten Vercammen et al. „Goal-oriented mutation testing with focal methods“. In: *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*. ACM, Nov. 2018.

[166] Laszlo Vidacs et al. „Assessing the Test Suite of a Large System Based on Code Coverage, Efficiency and Uniqueness“. In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, Mar. 2016.

[167] Tassio Virginio et al. „JNose“. In: *Proceedings of the 34th Brazilian Symposium on Software Engineering*. ACM, Oct. 2020.

[168] Qianqian Wang et al. „Behavioral Execution Comparison: Are Tests Representative of Field Behavior?“ In: *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, Mar. 2017.

[169] Zan Wang et al. „MAP-Coverage: A Novel Coverage Criterion for Testing Thread-Safe Classes“. In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, Nov. 2019.

[170] Michael Whalen et al. „Observable modified condition/decision coverage“. In: *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, May 2013.

[171] Chu-Pan Wong et al. „Efficiently finding higher-order mutants“. In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Nov. 2020.

[172] Tao Xie et al. „A framework and tool supports for generating test inputs of AspectJ programs“. In: *Proceedings of the 5th international conference on Aspect-oriented software development - AOSD '06*. ACM Press, 2006.

[173] Xiangjuan Yao et al. „A study of equivalent and stubborn mutation operators using human analysis of equivalence“. In: *Proceedings of the 36th International Conference on Software Engineering*. ACM, May 2014.

[174] Dongjiang You et al. „Efficient observability-based test generation by dynamic symbolic execution“. In: *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, Nov. 2015.

[175] Fadi Zaraket et al. „Property based coverage criterion“. In: *Proceedings of the 2nd International Workshop on Defects in Large Software Systems Held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2009) - DEFECTS '09*. ACM Press, 2009.

[176] Jie Zhang et al. „Predictive mutation testing“. In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, July 2016.

[177] Lingming Zhang et al. „Faster mutation testing inspired by test prioritization and reduction“. In: *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. ACM, July 2013.

[178] Lingming Zhang et al. „Operator-based and random mutant selection: Better together“. In: *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, Nov. 2013.

[179] Lingming Zhang et al. „Regression mutation testing“. In: *Proceedings of the 2012 International Symposium on Software Testing and Analysis - ISSTA 2012*. ACM Press, 2012.

[180] Peng Zhang et al. „CBUA: A probabilistic, predictive, and practical approach for evaluating test suite effectiveness“. In: *IEEE Transactions on Software Engineering* (2020), pp. 1–1.

[181] Junji Zhi et al. „On Adequacy of Assertions in Automated Test Suites: An Empirical Investigation“. In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*. IEEE, Mar. 2013.

[182] Qianqian Zhu et al. „Mutation Testing for Physical Computing“. In: *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, July 2018.