CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Exploring Heuristics for Predicting Microbenchmark Stability and Code Coverage using Static Code Analysis

Master's thesis in Computer science and engineering

Sam Salek Maghsoudi
Malte Åkvist

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2024

Master's thesis 2024

Exploring Heuristics for Predicting Microbenchmark Stability and Code Coverage using Static Code Analysis

Sam Salek Maghsoudi
Malte Åkvist

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2024

Exploring Heuristics for Predicting Microbenchmark Stability and Code Coverage using Static Code Analysis
Sam Salek Maghsoudi
Malte Åkvist

Supervisor: Philipp Leitner, Department of Computer Science and Engineering
Examiner: Gregory Gay, Department of Computer Science and Engineering

Master's Thesis 2024
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2024

Exploring Heuristics for Predicting Microbenchmark Stability and Code Coverage using Static Code Analysis
Sam Salek Maghsoudi
Malte Åkvist
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Performance testing is a method to optimize performance and identify regressions in software applications. This method can be employed through microbenchmarks, which measure the performance of a small unit of code. However, writing accurate microbenchmark tests is difficult as they have a high need for precision. Tools have been developed to automate the creation of microbenchmarks, but the method they implement results in large benchmark suites and thus also long running times when executing the suites. A solution to this problem is to only select a subset of the benchmarks to reduce the execution time of the whole suite. The goal of this thesis was thus to explore heuristics for selecting benchmarks with high stability and/or high code coverage; stability and code coverage are two important properties of benchmarks that are useful for detecting performance regressions. A laboratory experiment was conducted to explore two heuristics: firstly, a suitable heuristic for predicting the stability of microbenchmarks using only code features from static code analysis; secondly, a heuristic for a suitable approach for combining the stability of benchmarks with their code coverage. The experiment used 2250 JUnit tests from three open-source projects by converting them to benchmarks with the tool ju2jmh. Data from these benchmarks was used to design the heuristics. The first heuristic was created through regression models, where four separate candidates were explored. The model with the best performance was a Random Forest, gaining an R^2 value of 0.214 and a mean absolute error (MAE) of 3.491. This indicates that it performs better than always predicting the median value, but is still of low explanatory power. The second heuristic was designed using average rank aggregation and showed promising results. A balance was achieved where strengths in either stability or code coverage compensated for lesser performance in the other.

Keywords: Computer science, engineering, thesis, microbenchmarks, Java, JMH, stability, code coverage.

Acknowledgements

We are very grateful for Philipp Leitner as our supervisor for providing us with support, guidance, and feedback throughout our work on the whole thesis. We would also like to thank our examiner Gregory Gay for giving feedback and advice on the study.

Sam Salek Maghsoudi, Malte Åkvist, Gothenburg, June 2024

Contents

List of Figures

List of Tables

1
Introduction

In the realm of software development, the pursuit of both optimizing performance and ensuring it does not degrade is an ever-evolving challenge. This challenge can be facilitated by assessing the application's performance through performance testing, which seeks to test performance aspects such as stability, latency, speed, and more. Performance testing can be employed through microbenchmarks, a common testing technique within the field. Microbenchmarks are tests that measure the performance of a very small unit of code, such as a method, in order to make sure a solid performance baseline is held. Thus, by testing a small portion of code in isolation, its performance can be efficiently assessed.

Writing microbenchmark tests of high quality that are both accurate and meaningful is difficult. They have a high need for precision in order to detect any differences that occur in performance, and evaluating the quality of them can be tough. To help the developer in this process, the recently developed ju2jmh [1] tool can be utilized to automatically generate JMH (Java Benchmark Harness) [2] benchmarks from pre-written unit tests in Java (JUnit tests), which means the time spent manually writing benchmarks is instead automated. While this tool has proven to be effective [3], the method applied by ju2jmh involves using all of the JUnit tests from a provided selection of test classes. This leads to long running times when executing the generated microbenchmark suite.

This study will focus on exploring heuristics that can potentially be applied to microbenchmark tools such as ju2jmh, so the tool gains the capability of finding which JUnit tests are suitable candidates to generate the microbenchmarks from, with the goal of speeding up the execution time of the generated suite. In order to evaluate this, knowledge of what characterizes a valuable microbenchmark has to be assessed. Two general properties that are important to benchmarks are their stability and their code coverage. Stability refers to the reliability and consistency of the benchmark results [4]. The property is important since inconsistent performance can lead to unpredictable behavior and false results in terms of detecting performance regressions. Code coverage refers to how much of the source code is executed in terms of lines, branches, or other elements [5]. The importance of code coverage lies in the fact that the number of faults in software tends to increase as its source code grows in size [6]. Thus, if a larger area of the code is covered by tests, a larger percentage of faults and regressions can be discovered. Depending on the user's needs, project specifics, or other factors, either microbenchmark stability, code coverage, or a combination of the two could be important properties that should be taken

into account when generating benchmarks. Consequently, the overall question of interest in this study is: which subset of JUnit tests should be selected to generate microbenchmarks that achieve high stability, or both high stability and high code coverage at the same time?

1.1 Problem statement

Today, software developers have access to a wide range of tools and frameworks that allow them to carry out performance testing of various types and focuses. These include JMeter [7], LoadRunner [8], Java Microbenchmark Harness (JMH) [2], and many more. According to Khatchadourian *et al.* [9], using a testing framework is important for isolating sources of performance changes within code, and projects with tests from such a tool will provide the best indicators for improvements in performance. Despite the range of performance testing tools and their stated importance, a paper about an exploratory study of performance testing in open-source Java projects [4] from Laaber and Leitner found that most of the projects contained few performance tests. The paper also found that the performance tests which did exist were not well maintained; the tests were usually written as a one-time activity. Furthermore, the projects did not seem to get many code contributions related to performance testing after their initial implementation.

With the goal of reducing the needed effort and struggles of manually writing performance tests, Alexandersson developed the tool ju2jmh [1] which utilizes existing JUnit tests to generate microbenchmarks [10]. The method applied by ju2jmh involves generating benchmarks for every single JUnit test from a provided selection of test classes. While this is intended behavior, this method leads to execution times that can extend to weeks when running the generated microbenchmark suite, since many potentially redundant JUnit tests get included in the suite. A test case minimization during benchmark generation would make the generated suite faster to execute. Such a change could make the tool more appealing to developers and thus increase the interest in including performance tests in their software projects.

1.2 Purpose of the study

The primary purpose of this study is to use source code features from static code analysis to identify suitable heuristics for determining the Java unit tests to be selected when creating a microbenchmark suite focused on (1) stability, and (2) stability and code coverage simultaneously. Static code analysis was selected for designing the heuristics because it does not require as much overhead compared to dynamic approaches, where, for example, test cases would need to be run to gather any data for the heuristics. By avoiding dynamic implementations, the heuristics will be faster and more practical to execute in practice, in addition to being easier to implement for developers.

The broader purpose of this study is also to contribute to the larger field of performance benchmarking by providing the above-mentioned heuristics. The notion is that implementing these heuristics could potentially improve certain microbenchmarking tools, such as ju2jmh, by reducing the size of the generated microbenchmark suite and thus making the tool more appealing to use by developers. Increasing the usage of these tools would result in performance faults or bottlenecks being found more easily and rapidly, thus making software projects more stable with regard to performance.

Overall, this study can lead to faster microbenchmark suites within software projects as the number of benchmarks in the suite is minimized. Furthermore, the study can provide more efficient benchmarking options for developers and researchers. The results of the study may also spark new ideas within the field which can be used to fuel future research and studies around the topic.

1.2.1 Research questions

By finding heuristics that can identify specific unit tests based on important properties of microbenchmarks, one should be able to apply the heuristics to ju2jmh, or other similar microbenchmarking tools, so the tool only selects a specific subset of unit tests when generating the benchmarks, resulting in an overall faster execution time for the microbenchmark suite. By utilizing static code analysis for designing the heuristics, overhead generated by dynamic approaches can be avoided, which would achieve more practical heuristics that are easier to implement in practice. The first research question outlined below tackles one important performance property (stability), with the second research question adding another property (code coverage) to the resulting heuristic from the first question.

RQ1: *What is a suitable heuristic based on source code features computed through static code analysis for identifying unit tests for a microbenchmark suite focused on stability?*

Stability is an important property of microbenchmarks and can be defined as the extent of variability in execution time under repeated runs of a benchmark [4]. This property is valuable because inconsistent performance might provide unexpected behavior and inaccurate results when attempting to identify performance regressions. Thus, the stability of a benchmark indicates when a change in performance can be consistently identified. The metric utilized for variability to measure benchmark stability is Relative Median Absolute Deviation.

RQ2: *What is a suitable approach for combining the resulting heuristic from RQ1, focused on stability, with code coverage to create a new heuristic with an equal focus on both stability and code coverage?*

Code coverage is a measure that indicates the amount of source code that is executed when a test suite is run [5]. Selecting microbenchmarks with high code coverage could be important, as they test a larger portion of the program, thus the likelihood of identifying performance bugs can be increased. Both stability and code coverage can be seen as important properties of microbenchmarks, but they do not need to be mutually exclusive. Scenarios can occur where both stability and code coverage are equally important, and as such, both properties might need to be considered at the same time when creating a microbenchmark suite. The intention is therefore to explore an approach for combining the heuristic used for stability with code coverage. The metric used for code coverage is a proxy for line coverage, the lines of code that are potentially executed by a benchmark.

1.3 Significance of the study

The significance of this study is in its aim to contribute to the field of performance benchmarking and to extend the knowledge within the area by developing suitable heuristics for two specific scenarios: (1) when there's a need for a microbenchmark suite based on unit tests with a focus on stability, and (2) when there's a need for a microbenchmark suite based on unit tests with a focus on stability and code coverage simultaneously. The idea is that these heuristics will be able to provide software projects with a faster microbenchmark suite since the suites will be focused on specific properties of performance benchmarking. By incorporating the heuristics into a microbenchmarking tool, such as ju2jmh, there is a hope that the tool will become more appealing to developers in such a way that said tool is applied to existing or future projects with the intention of creating a faster microbenchmark suite for the project. This improvement would not only make it easier and faster for developers to find faults and regressions within their code, but also improve the reliability of the software if said faults are eventually fixed. This study is also aimed at being significant for researchers, as the developed heuristics can be further

analyzed for improvement or used to develop new techniques in the performance benchmarking field. The goal is to provide enough insight that future studies or experiments can be conducted based on the findings within this thesis.

1.4 Thesis outline

Relevant background theory is presented in Chapter 2 *Background*, while previous research and works related to this study are discussed in Chapter 3 *Related Work*. The method employed by the study is described and outlined in Chapter 4 *Methods*. The actual implementation efforts of what was mentioned in Chapter 4 *Methods* are then explained in the following part, Chapter 5 *Implementation*. Finally, the results of the experiment are presented in Chapter 6 *Results* and discussed in Chapter 7 *Discussion*. The thesis concludes with Chapter 8 *Conclusion*, which outlines the overall outcome of the study and the steps that can be taken for future work.

2
Background

This chapter covers essential concepts and background information relevant to this study. This includes general information on performance testing and microbenchmarks, the JMH framework, Java, benchmark generation, benchmark stability and code coverage, regression models, evaluation metrics, and more.

2.1 Performance testing

One of the most important aspects of software development is testing the software to make sure it works properly as intended to meet customer and stakeholder expectations and requirements. The testing of software applications comes in many shapes, and the ways to test software are highly dependent on the goals and purpose of said software. One way to test projects is through functional testing. Functional tests are generally seen as the baseline of software testing; this is when the functionality of the software is tested, and it is usually done with so-called unit tests. Unit tests evaluate the smallest functional unit of code, typically a code function or method, which helps in isolating potential bugs and issues in the code [11]. Another approach to testing is performance testing, which is used to test the speed, latency, or throughput of the software to identify areas for optimization. One common way to do performance testing is through the use of benchmarks, which measure the performance of a software system against a predefined baseline.

There are many ways to perform software tests, whether it be functional testing or performance testing. And as test suites grow in size, it is common for developers to integrate certain tools into their projects to help control and operate all the tests. One such type of tool is called a "test harness", which can generally be seen as a framework or toolset used to assist the developer in running and managing tests. Test harnesses provide stubs and drivers, which are small programs that interact with the software under test. These small programs create a structured environment for executing tests and collecting their results, and they can also be used to automate tests [11]. According to Khatchadourian *et al.* [9], using a test harness is important in isolating sources for performance changes within code, and projects with tests from such a harness will provide the best indicators for improvements in performance. In a paper about an exploratory study of performance testing in open-source Java projects [4], it is found by Laaber and Leitner that most of the projects contained few performance tests. The paper also found that the performance tests that did exist were not well maintained, as they were usually written as a one-time activity.

Furthermore, the projects did not seem to get many code contributions related to performance testing, which is problematic given the previously stated importance of performance tests.

2.1.1 Microbenchmarks

Typical benchmarks are usually done on larger parts of the software but they can also be done on a smaller scale. Microbenchmarks are a specific type of benchmark that does benchmarking on a smaller scale, always pertaining to a small amount of code, such as functions or methods. Essentially, microbenchmarks can be compared to unit tests; both are similar in scale but differ in what aspect of the software they test, performance versus functionality. The performance they test is often execution time, but could also be memory usage, CPU consumption, and so on. When measuring the execution time of these finely-grained software components there are many factors that can distort the results. These include just-in-time compilation, garbage collection, dynamic loading, and background operations of the operating system (OS) [12]. Fortunately, many of these factors can be mitigated by properly applying a library specifically designed for benchmarks. A common benchmarking library in Java is JMH, which is the one employed in this study.

2.1.2 Java Microbenchmark Harness (JMH)

Java Microbenchmark Harness [2], also known as JMH, is a test harness for building, running, and analyzing various types of benchmarks, including microbenchmarks, in Java and other languages that use the Java Virtual Machine (JVM). JMH provides a structured framework for conducting microbenchmarks, ensuring that measurements are accurate and reliable. It is also easy to integrate into JVM projects through the use of popular build tools such as Gradle [13] and Maven [14].

In the world of performance benchmarking, tools and frameworks allow the user multiple ways to configure and perform the execution of benchmarks, and JMH is no different. However, some aspects related to the JMH configuration are more important than others as they modify the benchmark execution in a significant way. Thus, these aspects are worthy of highlighting: forks, iterations, and warmup iterations [2]. Forks refer to the number of environment instances used for running a benchmark suite. The purpose of forks is to try and mitigate the effects of warmup and compilation in the environment, both of which can have an impact on the accuracy of benchmark results. In regards to JMH more specifically, forks are the number of JVM instances used for a benchmark. By running benchmarks in separate JVM instances, any JVM-specific optimization or warmup effects are isolated, which leads to more consistent and reliable results. Iterations, sometimes also called measurement iterations, are the number of times a benchmark is executed within each fork (and thus within each environment). A benchmark is run as many times as it can over the specified time frame of the iteration. Each iteration measures the performance of the code under test and provides data about said performance. Generally, more iterations lead to more accurate benchmark results since more measurements

are gathered. Warmup iterations are iterations executed before the measurement iterations with the purpose of reducing environment-based effects, similar to forks. Unlike regular iterations however, warmup iterations take no measurements since their main goal is to warm up the code to be run during measurement iterations.

JMH offers a wide array of command-line arguments for executing benchmarks, these include number of forks, measurement iterations, warmup iterations, and much more, all of which can be configured when running the JMH command. This allows users to tailor and customize the benchmark execution to suit their various needs. Below is a table containing a quick overview of the most common and important command line arguments used during this study. A more extensive look at all the arguments that JMH has to offer can be found at [15].

Table 2.1: Common and important JMH command line options for this study.

Argument	**Description**	**Input**
bm	Benchmark mode. The way the benchmark should be run and hence the metric used for benchmark results (e.g., average time, throughput, etc.).	Name of benchmark mode.
tu	Time unit. The unit used for time in the benchmark results.	Name of time unit.
f	Forks. The number of times to fork a single benchmark.	Integer.
wi	Warmup iterations. The number of warmup iterations to run for each benchmark.	Integer.
i	Iterations. The number of measurement iterations to run for each benchmark.	Integer.
w	Warmup iteration minimum time. The minimum time to spend on each warmup iteration.	Integer + time unit.
r	Iteration minimum time. The minimum time to spend on each measurement iteration.	Integer + time unit.
foe	Fail on error. Decides if the whole JMH run should fail if an unrecoverable error is encountered during the run.	Boolean.
o	Output to file. Outputs the results of the whole benchmark run to a file	Name of output file.

2.2 Java

Java [16] is a widely popular object-oriented programming language, built with the purpose of being platform independent. The language allows developers to write code once and run it anywhere due to its JVM, and it also features automatic memory management. Java is most commonly used for web applications, mobile apps, and enterprise systems. This language acts as the main focus of this study.

2.2.1 Java Virtual Machine (JVM)

The JVM, which stands for Java Virtual Machine, is a virtual machine that acts as a bridge between JVM languages and the underlying hardware and OS [17]. Programming languages that utilize the JVM, such as Java [16] and Kotlin [18], compile to Java bytecode, a platform-independent representation of the program's instructions. The JVM loads this bytecode and translates it to machine-specific code that the hardware can understand and execute.

Some noteworthy techniques and technologies applied by the JVM is a Just-In-Time (JIT) [19] compiler and garbage collector [20]. The JIT compiler functions by dynamically translating Java bytecode into machine code during runtime. Unlike traditional compilers that translate the whole program before execution, the JIT compiler works on portions of code as they are executed, which optimizes performance. In regards to the garbage collector, it is responsible for automatically managing the memory in JVM programs. The garbage collector identifies and reclaims memory occupied by objects that are no longer in use, thus preventing memory leaks and ensuring efficient memory utilization. These two JVM aspects can significantly influence performance testing since the performance of a JVM program is deeply dependent on both. This is because dynamically optimizing the code and memory during runtime has the potential of improving the performance over time.

2.2.2 JUnit

JUnit is a testing framework that allows developers to write and execute repeatable tests for projects that use the JVM [21]. The framework is most commonly used as a foundation for the creation and management of unit tests within JVM languages, such as Java. Thus, JUnit allows developers to evaluate the functionality of their code, helping in isolating any potential issues and ensuring correctness of the code's intended behavior. For Java, JUnit tests follow the same syntax as regular methods, but are annotated with "@Test" to identify them as unit tests for the framework [22]. Figure 2.1 displays an example of a JUnit test that tests the functionality of a method from the *Math* class.

```
1  import static org.junit.Assert.assertEquals;
2  import org.junit.Test;
3
4  public class Math {
5      // Method that returns the cube of the given number.
6      public int cube(int n) {
7          return n * n * n;
8      }
9  }
10
11 public class MathTest {
12     // JUnit test for the cube() method in the Math class.
13     // 3 cubed should equal 27.
14     @Test
15     public void testCube() {
16         Math math = new Math();
17         assertEquals(27, math.cube(3));
18     }
19 }
```

Figure 2.1: Example of a JUnit test, rows 12-18.

2.3 ju2jmh

As with any field in software engineering, tools are continuously researched and created to help software developers in their work, to make their tasks easier and more time efficient. Within the field of performance testing for Java, one such tool is ju2jmh [1], also known as junit-to-jmh. This tool takes existing JUnit tests and generates JMH performance microbenchmarks from them. This approach was developed by Alexandersson [10], and was partly created to reduce the struggle of manually creating and writing performance tests. During the evaluation of the approach, Alexandersson determined that the generated benchmarks are similar in quality compared to human-written benchmarks in terms of stability. The ju2jmh tool needs four arguments when it is launched, as shown in Table 2.2.

Table 2.2: The four needed launch arguments for ju2jmh.

Position	**Description**
1.	The root directory of the source files for the unit tests.
2.	The root directory of the compiled class files for the tests.
3.	The root directory where the resulting benchmark source files should be generated.
4.	A list of fully qualified test class names to generate benchmarks from.

When the tool is launched with correctly inputted arguments, it parses through the directory from the first argument, locates the specific test classes provided by the fourth argument, and places copies of those classes in the location given by the third argument. Additionally, it generates a new inner class with the name "__Benchmark" within each test class copy. The tool then generates individual benchmarks from all identified JUnit tests within each test class copy and puts them inside their respective inner class. The inner class extends a separate utility class provided by ju2jmh and also creates some additional utility code to allow the benchmarks to run. The tool cannot generate benchmarks for lone JUnit tests, rather it only works by utilizing a provided selection of test classes and thus generates benchmarks for **all** of their underlying JUnit tests. Figure 2.2 displays the generated class and benchmarks when ju2jmh is run on the *MathTest* class from the previous JUnit example (Figure 2.1).

```
1  import static org.junit.Assert.assertEquals;
2  import org.junit.Test;
3
4  // ju2jmh generated copy of the original MathTest class.
5  public class MathTest {
6      // JUnit test for the cube() method in the Math class.
7      @Test
8      public void testCube() {
9          Math math = new Math();
10         assertEquals(27, math.cube(3));
11     }
12
13     // Inner class for benchmarks, generated by ju2jmh.
14     @org.openjdk.jmh.annotations.State(
15     org.openjdk.jmh.annotations.Scope.Thread)
16     public static class _Benchmark extends
17     se.chalmers.ju2jmh.api.JU2JmhBenchmark {
18
19         // Generated benchmark for the testCube() JUnit test.
20         @org.openjdk.jmh.annotations.Benchmark
21         public void benchmark_testCube() throws
22         java.lang.Throwable {
23             this.createImplementation();
24             this.runBenchmark(this.implementation()::testCube,
25             this.description("testCube"));
26         }
27
28         // Below is utility code used by ju2jmh.
29
30         private MathTest implementation;
31
32         @java.lang.Override
33         public void createImplementation() throws
34         java.lang.Throwable {
35             this.implementation = new MathTest();
36         }
37
38         @java.lang.Override
39         public MathTest implementation() {
40             return this.implementation;
41         }
42 }
```

Figure 2.2: Example of the benchmark inner class for *MathTest*, generated by ju2jmh. A ju2jmh benchmark is found between rows 19-26 (benchmark for *testCube()* method).

2.4 Benchmark stability

The stability of benchmarks is one of its many important properties. Stability refers to the variability of the benchmark's results, where high stability means a stable and consistent result after many consecutive benchmark executions, and low stability indicates unpredictable results after repeated runs of the benchmark. The metric being measured for the results can vary, but it is typically one of the common performance metrics, such as response time, execution speed, throughput, etc. Stability is important because if the results of a benchmark are highly variable it may indicate unpredictable behavior from the system under test, and thus the results could report false positives or false negatives in terms of identifying performance faults or regressions. There are multiple methods in which the variability of benchmarks can be measured, such as through the standard deviation, Coefficient of Variation (CV), Relative Median Absolute Deviation (RMAD), and so on. This thesis focuses on RMAD as the metric for stability, due to its robustness against outliers in the data.

Variability measurements that assume a normal distribution, such as the standard deviation and CV, become less informative when the underlying data is not normally distributed. This is because they give equal weight to all data points, causing outliers to affect the result disproportionally. In such scenarios, a robust variability measurement that is good at handling skewed distributions should instead be utilized. One such measure is RMAD [23], a metric for measuring the variability in a set of data points in relation to their median value. RMAD is an altered version of the Median Absolute Deviation (MAD). In RMAD, the MAD value is divided by the median of the dataset, thus making RMAD relative to the median value. This makes RMAD less likely to be influenced by extreme values or outliers in the dataset in comparison to other measurements such as the standard deviation. The formula for RMAD is given by

$$\text{RMAD} = \frac{\text{MAD}}{\text{Median}(X)}$$

where MAD is the Median Absolute Deviation and is calculated from

$$\text{MAD} = \text{Median}(|X_i - \text{Median}(X)|)$$

where X in both of the formulas represents the dataset and X_i represents each value in the dataset.

2.5 Code coverage

Code coverage is one of the many measurements used in software testing to quantify how much of a project's source code is executed, and in turn *not* executed, by a test suite [5]. By highlighting code that lacks coverage from test cases, developers can more easily identify gaps in testing, and filling in those spots will help to improve the code's overall quality and reliability. In addition, by making sure that greater portions of the program are covered, the likelihood of identifying performance bugs is naturally increased. Code coverage comes in different types, each measuring in a unique way which gives various perspectives on how much of the code is covered. Some of the most common criteria for code coverage are line coverage, branch coverage, function coverage, and statement coverage. This study focuses on line coverage and utilizes lines of code, or LOC in short, as a static proxy for the criteria.

LOC is a common and popular metric within software engineering that counts the number of lines in a piece of source code. The metric's popularity is mainly due to how simple and easy it is to measure [24] while still being able to offer insightful information on a number of software characteristics; e.g., studies have indicated that as the source code of programs gets larger, so does the number of faults per line [6]. Multiple ways to count LOC exist. It is measured statically, with there being two major variants of its measurement: physical lines of code and logical lines of code. Physical LOC counts every line of the source code, which includes blank (empty) lines, executable lines, code comments, etc. Logical LOC only counts lines with executable statements, hence blank lines and code comments are not included. This study utilizes the physical LOC variant.

While the way code coverage is gathered can be quite dependent on the programming language, it is typically calculated dynamically through specialized tools, frameworks, or libraries. For example, one of the most popular tools for analyzing code coverage in Java is a library named JaCoCo [25]. The tool works by adding new code to the Java bytecode of a program when its test suite is executed, through a method called bytecode instrumentation [26], where these newly added pieces of code record which parts of the source code are executed. Using LOC as a static proxy for the dynamic metric of code coverage is quite optimistic. The reason for this is that LOC measures all the lines in the source code of the tests, instead of the actual lines that are covered. For example, LOC as a metric will count all the lines within a test, even though the actual execution path might only cover a fraction of its code. LOC as a metric thus indicates the theoretical maximum number of lines that can be covered, even though the actual code covered is always going to be lower than (or in rare instances equal to) the measured LOC. Since we chose to focus on line coverage as the criteria for code coverage we believe LOC to be a reasonable proxy for it. This is because although optimistic, it provides an estimate of the lines that are covered.

2.6 Regression models

Regression models are statistical models used to examine the relationship between one dependent variable, called the target, and one or more independent variables, called predictors. A regression model provides a function that describes the relation between the variables, with the primary goal being to understand how the predictors contribute to changes in the target. Regression models are quite important to this study as they are used during the development of the candidate heuristics for the study's research questions. There is a vast selection of regression models to utilize, all of them applying different functions and methods to map the relationship between the data. This study will focus on linear regression, Ridge regression, Random Forest, and Gradient Boosting [27].

2.6.1 Linear regression

The linear regression model is one of the most popular and simple models for regression tasks. This model tries to find a relationship between the predictor(s) and target variables by fitting a linear equation to the observed data.

2.6.2 Ridge regression

Ridge regression is a variant of the linear model that tries to handle multicollinearity, i.e., when the independent variables in the data are highly correlated with each other. The model addresses this issue by adding a penalty to the standard least squares method it uses, which helps in stabilizing the model and reducing the variance of the estimates. Ridge regression is therefore useful to mitigate multicollinearity issues that can occur in linear regression models with a large number of predictors.

2.6.3 Random Forest

Random Forest is a non-linear model used for both classification and regression tasks. It is an ensemble model, meaning it creates a more reliable and accurate prediction by combining the projections of several different models. Random Forest works by independently constructing and utilizing multiple decision tree models, which make predictions through a sequence of binary decisions that creates a hierarchical tree structure where each leaf node is a predicted outcome. For regression tasks, Random Forest models output the mean prediction of the individual decision trees.

2.6.4 Gradient Boosting

Gradient Boosting is a model similar to Random Forest. It is a non-linear ensemble model, typically constructed from decision trees, and it can handle both regression and classification tasks. Where it differs from a Random Forest model is in its training process; Random Forest creates its trees independently, while Gradient Boosting constructs the trees sequentially one at a time, thus each tree learns to correct the errors made by the previous tree. The output of a Gradient Boosting model for regression tasks is the sum of the predictions of all individual trees.

2.7 Evaluating regression models

There are multiple ways of evaluating regression models, one of which is with certain statistical metrics. There is a large suite of metrics that can be applied depending on the given situation and needs of the analysis. This section describes two metrics of note for the study.

2.7.1 R-squared (R^2)

R-squared, denoted as R^2 and also known as "coefficient of determination", is a statistical measure which indicates the proportion of variance in the dependant variable that is predictable by the independent variable(s). This metric usually ranges between 0 and 1, where a greater value suggests a more precise match between the data and the model. Essentially, a R^2 value of 1 indicates that all of the variation in the target variable is explained by the predictor variable(s), thus meaning a perfectly fit model where there is a perfect relationship between the target and predictor(s). A R^2 value of 0 indicates the opposite, that none of the variations in the target is explained by the predictor(s), resulting in a model that doesn't fit the data at all. While the standard range for R^2 is typically between 0 and 1, a negative score can be obtained in cases where the model's predicted values perform worse in comparison to using the average as the predicted value. This can happen when the model is not appropriate for the data, e.g., if the relationship between the target and predictor(s) is non-linear but a linear model is applied.

2.7.2 Mean absolute error (MAE)

Mean absolute error, shortened as MAE, is a common metric used to evaluate the performance of regression models. It is described as the average size of the errors between the model's predicted values and the dataset's actual values. For instance, if a model has an MAE of 5 it would mean that its predictions are off by 5 units on average. An MAE of 0 indicates a model without error, a perfect relationship between the predicted values generated by the model and the actual values in the data. The unit expressed by MAE is the same as the dependent variable (target) of the model, which makes the metric easy to interpret and analyze. MAE is also quite robust against outliers in the data since it treats all errors equally in terms of magnitude, which results in outliers having less of an impact on the metric overall.

3
Related Work

In addition to the background information provided, there exists academic research that is relevant to this study. This chapter aims to outline these prior works and describe their connection to this thesis.

3.1 Predicting stability of microbenchmarks

An article that closely resembles the work in this thesis is "Predicting unstable software benchmarks using static source code features" by Laaber *et al.* [28]. Using the programming language Go, the authors predicted whether benchmarks are stable or unstable by parsing source code features of the benchmarks and then training a model on these using machine learning. Laaber *et al.* conducted their experiment on 4461 Go benchmarks derived from 230 open-source software projects, and they found that a Random Forest model performed the best, having a prediction performance from 0.79 to 0.90 using the area under the curve (AUC) score. Three different variability measurements were used, two types of relative confidence interval width and RMAD (metric described in Section 2.4), with RMAD having the best prediction performance. Their experiment is similar to ours, particularly RQ1, which regards creating a heuristic able to identify the stability of a benchmark generated from a JUnit test. Some important differences to point out however are the programming language used, how the benchmarks are created, and the modeling of the problem. This thesis focuses on Java with benchmarks generated using the ju2jmh tool, compared to theirs using handwritten benchmarks created in the Go language. They also transformed the problem from regression-based into a binary classification problem. This was done by transforming the benchmarks into two classes; stable or unstable, depending on whether the benchmark variability is above or below a certain threshold value. In comparison, this study uses the regression-based approach to predict the stability of benchmarks as a continuous value instead of binary classification.

An important thing to note is that initially in their experiment, Laaber *et al.* tried a regression-based approach but without much success, as the error was especially high. By simplifying the problem to binary classification, they were able to achieve far better results. The authors mention that potential reasons for the high error could be because their selected static source code features were not precise enough, or that source code features generally do not provide enough explanatory power for regression-based predictions. Future research suggested investigating whether other

or more specific features could predict exact benchmark stability, which is explored in this thesis. While many of the source code features selected in this study are inspired by Laaber *et al.*, language-specific features such as the number of "java.io" package uses, which are exclusive to Java, make the selected source code features different by definition.

3.2 Performance regressions

Chen and Shang conducted an empirical study on performance regressions found in source code changes from releases of the two open-source projects Hadoop and RxJava [29]. They identified performance regressions by running microbenchmarks on each commit and measuring if a significant degradation was found in the response time, CPU usage, or I/O traffic. The findings of the study were that most performance regressions are created when fixing other bugs and that there are six primary causes of performance regressions. The study is relevant as it explains the process of how performance regressions are introduced and how they can be detected.

Described in a paper by Costa *et al.* [30], a study was done about bad practices when writing microbenchmarks using the JMH framework. The authors presented five bad JMH practices and found that these practices are prevalent in open-source Java projects. They also showed that fixing them tended to lead to a statistically significant difference in performance results. The findings of this study can be useful for developing the heuristics, since understanding common pitfalls and mistakes in JMH microbenchmarking can help in creating a more effective approach for generating the microbenchmark suite. Knowing bad practices can assist in designing the heuristics to prioritize unit tests that adhere to good practices, ensuring more fair and consistent benchmark results. This is especially important for RQ1's stability heuristic, since benchmark stability describes the variability of execution time under a repeated amount of runs of the benchmark [4].

In a paper by Laaber and Leitner [4], the authors investigate the quality of microbenchmark suites with a focus on suitability to provide prompt performance feedback and integration with Continuous Integration (CI). The paper also discusses in detail the use of slowdowns as a way to evaluate microbenchmark suites. The information about slowdowns and the methods they utilized may be useful to incorporate in this study, primarily during the evaluation of the microbenchmark suites that will be selected by the developed candidate heuristics.

3.3 Applications of ju2jmh

The ju2jmh tool was employed in another article by Jangali *et al.* where the quality of the microbenchmarks it generates was evaluated across three open-source projects [3]. The presented findings show that when assessing performance in terms of execution time and throughput, the microbenchmarks generated by the tool consistently outperform benchmarks generated using the AutoJMH framework and existing JUnit tests. Furthermore, the tool's performance is on par with manually written benchmark tests in terms of stability and ability to detect performance bugs. This information is promising, as it suggests that running the generated benchmarks using the ju2jmh tool is likely to yield reliable results.

A thesis on a similar topic was done by Anthony Path [31]. In the study, eight subsets of the microbenchmarks generated using the ju2jmh tool were extracted and compared with each other against a set of metrics. This experiment was performed on three open-source projects, and the subset of benchmarks extracted were: benchmarks with runtimes of 0-5ms, 5-10ms, 10-15 ms, highest branch coverage, etc. The metrics these subsets were tested against were: missed branches, standard deviation (its metric for benchmark stability), and run time, among others. In the thesis, overall qualities of different benchmarks were assessed but no conclusion of what constitutes the "best" benchmark could be made.

4
Methods

The method to answer the research questions of this thesis is presented in this chapter. The study applied a scientific method, specifically a laboratory experiment [32], with three study objects.

4.1 Experiment approach

We approached the methodology to answer the two research questions by conducting a laboratory experiment [32], where the environment is contrived and all measurable variables are well controlled. The experiment was split into two separate parts, one for RQ1 and one for RQ2. Before work began on either RQ1 or RQ2, steps were taken to prepare for the two sections of the experiment. This included selecting study objects, picking a subset of JUnit tests from these projects to utilize during the experiment, and performing a static code analysis where the collected source code features can be used as data to design the candidate heuristics for both research questions. The list below shows the initial steps taken before focus was directed toward RQ1 and RQ2.

1. Select study objects.
2. Select a random set of JUnit tests from the study objects.
3. Extract and collect source code features from the selected JUnit tests.

4.1.1 RQ1 approach

The section of the experiment for answering the first research question followed a set of steps that happened in subsequent order, an overview can be seen in the list below and in Figure 4.1. Data collection was the first major and overarching procedure. For RQ1, the data is in the form of stability values and a selection of source code features (see Section 4.4.2.1) for each microbenchmark. The code features were used in conjunction with the stability values to give insight into the aspects of the source code that affects the stability of microbenchmarks. This insight and data were then utilized to design suitable candidate heuristics that could identify unit tests that would provide a microbenchmark suite with high stability. After every candidate heuristic was evaluated, one finalized heuristic was decided for the research question.

1. Collect benchmark stability data, done in multiple successive steps:
   (a) Convert JUnit tests to benchmarks.
   (b) Run the benchmarks.
   (c) Interpret benchmark results based on selected stability metric to gain stability values.
2. Design candidate heuristics based on the stability values and the static code analysis data from each benchmark.
3. Evaluate performance of each candidate heuristic, done in multiple successive steps:
   (a) Predict stability value of benchmarks by applying the candidate heuristic.
   (b) Compare accuracy of candidate heuristic prediction to actual stability value.

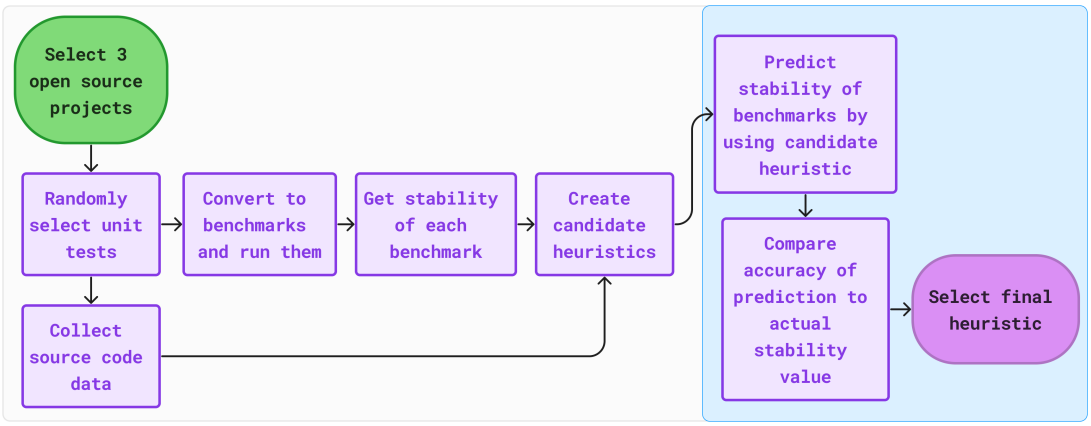

Figure 4.1: Method flowchart for RQ1, where the light blue section of the figure represents evaluation of the heuristic.

4.1.2 RQ2 approach

After a heuristic was found for RQ1, we could start the process of answering RQ2. The steps in the experiment for combining RQ1's heuristic with code coverage are seen in the list below, and an overview of this combination process can also be seen in Figure 4.2. The data that was collected for this research question is code coverage values for each microbenchmark. An approach was then explored for combining code coverage with the heuristic provided by the previous research question. Finally, the approach was evaluated in order to find how effectively it selected unit tests with a focus on stability and code coverage at the same time.

1. Rank benchmarks based on stability using heuristic from RQ1.
2. Rank benchmarks based on code coverage.
3. Design candidate heuristic by joining both rankings of each benchmark to create new combined ranking for the benchmark.
4. Evaluate combined ranking through qualitative comparisons between all rankings.

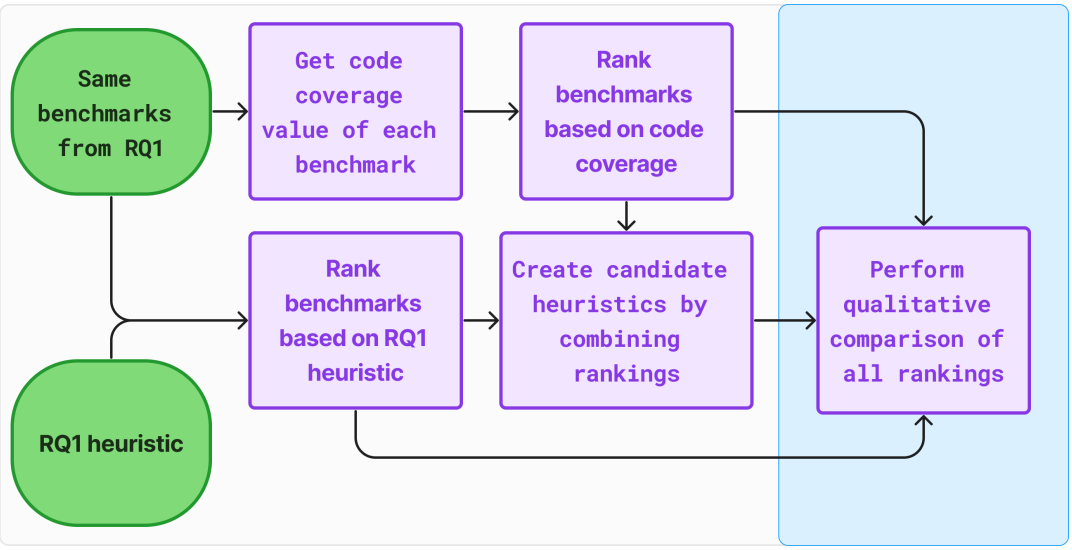

Figure 4.2: Method flowchart for RQ2, where the light blue section of the figure represents evaluation of the candidate heuristic.

4.2 Study objects

The needed data was extracted from a subset of JUnit tests from three selected open-source projects. We decided on exactly three projects due to time constraints and feasibility. The experiment had limited time to be conducted, and an increase in the number of projects would have made the management of the study objects more challenging in addition to increasing the overall time to perform said experiment. The selection criteria for the projects was to have at least 750 successfully functional and passing JUnit tests since we chose to use 750 test cases from each project for the experiment. The justification for why this amount of test cases was selected is discussed in Section 4.4.1. The projects also needed to use JUnit version 4 in order for the tests to be able to be converted into microbenchmarks with the ju2jmh tool. Finally, the projects should utilize Gradle [13] as its build tool, since ju2jmh has not been tested on projects with other build tools (such as Maven [14]). Having these criteria in mind, the selected projects were all found on the GitHub website [33]. The projects were RxJava [34], Mockito [35] and SonarQube [36], as seen in Table 4.1. While there are other projects on GitHub that fit our criteria, these projects were selected because they are either very popular within the Java ecosystem or have been utilized by other related work we drew inspiration from.

4.2.1 RxJava

The first study object, named RxJava, is a JVM implementation of "Reactive Extensions", which is a library for creating asynchronous and event-based programs by using observable sequences [34]. Version 3.1.8 was selected for the study as it was the most recent iteration of the project. This version contained roughly 12184 JUnit tests in total, which provided a lot of potential data for the experiment.

4.2.2 Mockito

The second study object was Mockito [35], a popular mocking framework for unit tests written in Java. The process of mocking involves eliminating external dependencies from a unit test and surrounding it with a controlled environment; this is usually done by creating dependency-free simulations of classes. The chosen version of the project, version 5.10.0, was the most recent when the experiment began, and had some 2316 JUnit tests.

4.2.3 SonarQube

SonarQube version 10.5.0.89998 [36] was the third and final study object we selected. SonarQube can analyze the code quality of software projects and provides the ability of continuous inspection. This project was quite large in size and scope, especially compared to RxJava and Mockito, so we could not use its whole codebase. We instead opted to utilize one of the many modules of the project, specifically the module named "sonar-scanner-engine". This module accommodates the main engine code for the tool's capability of analyzing the source code of a project, and it contained about 1188 test cases. When "SonarQube" is mentioned throughout this study, we are discussing and referring to this lone module only as no other parts of the full SonarQube codebase were made use of during the experiment.

Table 4.1: The three selected open-source projects used as study objects.

Project	**Description**	**Version**	**JUnit tests**
RxJava	Library for reactive programming.	3.1.8	12184
Mockito	Popular Java mocking framework.	5.10.0	2316
SonarQube	Code quality tool with continuous inspection.	10.5.0.89998	1188

4.3 Data description

The process of creating suitable heuristics for our research questions required a specific set of data since the heuristics are tailored for distinct properties of microbenchmarks. Specifically, the aim of this data was to provide insights into which aspects of a benchmark that affect its stability and code coverage. The goal was for the heuristics to only need statically gathered data to actually function and execute, with the reason being to reduce overhead and make the heuristics easier to apply in practice. This means data from dynamic sources, which typically produce overhead, are not to be used as input for the heuristics, but can be used to design and evaluate the heuristics. JUnit tests from three open-source projects were used as input data. We utilized this input data to generate additional data that could be applied when creating the candidate heuristics for both research questions. Each research question required a unique set of generated data.

4.3.1 Data for RQ1

For RQ1, the first set of generated data was the computed stability values of every individual benchmark. This needed to be done through a dynamic approach since actual stability can only be measured by executing the benchmarks. Relative Median Absolute Deviation (RMAD) was chosen as the metric of stability since it is robust against outliers and skewed datasets. The second generated data was a specific collection of source code features (see Section 4.4.2.1) extracted from the JUnit tests through static code analysis. The source code features contributed necessary information for developing suitable heuristics for RQ1, since the creation of our heuristics involved analyzing the relation between the benchmark's stability and its code features. These two in combination provided data that could be analyzed where correlations between the stability values and the static code analysis could be found. This insight into what code features affected benchmark stability was used to create the candidate heuristics for RQ1.

4.3.2 Data for RQ2

In regards to the second research question, the selected JUnit tests were also used to gather code coverage data for each microbenchmark. The code coverage criteria selected for this study is line coverage, as it relays a higher volume of information compared to other criteria. Code coverage is typically collected dynamically, but we did not want to use dynamically gathered data as input to the heuristics, since this would give unwanted overhead to the heuristics. We instead opted to utilize a statically gathered proxy for code coverage. Physical LOC was selected as this proxy, mainly due to its simplicity and how straightforward it is to understand. It was difficult to find alternative static features that could be used as a proxy, as there simply are not any clear options. Cyclomatic complexity is an alternative proxy that was considered but was ultimately decided against due to its complexity, time needed to implement, and to verify its measured values. Using LOC from a static code analysis is a naive approach since any statically gathered LOC value is expected to

be higher than the actual dynamic line coverage that was executed during runtime. Yet, this approach provides less overall overhead for the RQ2 heuristic, as its input data is available from a static context. The LOC values for each microbenchmark were applied to the finalized RQ1 heuristic to create the heuristic for RQ2.

4.4 Data collection

This section covers how the data was collected for this study. After deciding on three open-source projects, a subset of JUnit tests were selected. These test cases were employed during the static code analysis to gather data about source code features. The same set of test cases was also used to acquire benchmark stability.

4.4.1 Selecting JUnit tests

Due to the limited time of this study, only a subset of the total JUnit tests from the study objects could be utilized. The primary time constraint was during the gathering of stability values for the microbenchmarks, as running large benchmark suites can take multiple days, if not weeks. After examining a few different sizes of test suites, we decided on a total of 750 JUnit tests from each study object. We did not want to have too few test cases from each project as that could compromise the explanatory power of our models, but we also could not select more than 750 tests since the resulting suites would take too long to execute. We argue that this was a good balance between the time required to run the benchmarks and the size of the dataset. This amount of tests also allowed us to conduct the rest of the study in a reasonable time frame. The specific JUnit tests were selected randomly. Three separate lists were constructed, one for each study object, containing all of the project's identified JUnit tests, and 750 was randomly selected from every individual list.

4.4.2 Performing static code analysis

Both research questions needed source code data from a static code analysis of the JUnit tests: the full suite of our selected code features (see Section 4.4.2.1) for RQ1, while RQ2 only needed LOC (one of the selected features). To identify these code features, we needed a tool to aid us in the static code analysis on the selected subset of test cases. After a lot of searching, we were unable to find any public and free tool for this effort that could identify all of the code features. Thus, the analysis was performed with our own custom parser, specifically designed for our needs (see Section 5.3). This parser was utilized to go through the source code of the underlying JUnit tests of the microbenchmarks and look for our specific selection of code features, while documenting how many times they occurred. The number of occurrences of each feature were summed from each method called by each JUnit test, including methods that were called indirectly (e.g., testA calls method1, which calls method2, etc.). The gathered data of these code features were then preprocessed (see Section 5.4.5.1) and used to design the candidate heuristics.

Table 4.2: The selected source code features the parser looked for and documented.

Feature	**Description**	**Code example**
Conditionals	Number of conditionals.	if, else-if, switch.
Loops	Number of total loops.	for, for-each, while, do-while.
Nested loops	Number of total nesting levels from nested loops. Only the first nested loop on the same nesting block is counted.	loop inside a loop. E.g., *for { for { } },* *while { for { for { } } }*
LOC (total)	Total physical lines of code.	Not available.
LOC (JUnit test)	Physical lines of code for the starting method only, i.e., the initial JUnit test.	Not available.
Method calls	List of all method calls, including calls to methods in dependencies and libraries.	E.g., *SourceCodeClass.method(),* *java.util.Collections.sort(),* *junit.Assert.assertTrue()*
Source code method calls	List of method calls, excluding calls to methods in dependencies and libraries, such as Java Standard Library or JUnit.	E.g., *SourceCodeClass.method()*
Object instantiations	List of all object instantiations.	E.g., *new SourceCodeClass(),* *new java.lang.Thread()*
Package accesses	List of all accesses to packages. A package gets accessed when a class in the package gets utilized, such as method calls or object instantiations.	E.g., Access to "java.lang" package: *new java.lang.Thread()* *java.lang.Thread.sleep()*

4.4.2.1 Selected code features

A selection of source code features that our parser should document was decided on. This selection was based on the paper from Laaber *et al.* [28] where a similar study to ours was conducted with the Go programming language, see Section 3.1. We analyzed their set of code features and selected a range of them that performed the best in their study and which also could be applied to Java. The selection mainly consisted of code statements such as if-statements, loops, and method calls, but additional features that are more specific to Java and thus not present in their

paper were also added, e.g., the packages that are utilized from the methods. Certain code features, such as cyclomatic complexity [37] (software metric that attempts to measure code complexity), were also considered, but was ultimately decided against due to the time it would take to verify the parser could document the full list of metrics correctly. Table 4.2 contains the complete list of source code features that our parser searched for. The parsed data was preprocessed into usable features for the RQ1 candidate heuristics (see Section 5.4.5.1).

4.4.3 Collecting benchmark stability

The stability of the benchmarks was needed to design and evaluate the candidate heuristics for RQ1. This stability was gathered through a dynamic approach since that is the only way to collect the actual stability of benchmarks. More specifically, the ju2jmh tool was used to convert the selected JUnit tests into microbenchmarks. This allowed us to record the time it took to run each benchmark after the benchmark suite had been executed. The execution of the benchmarks happened inside multiple virtual machines (VMs) in the cloud, mainly to avoid interference from our local operative systems and the programs/processes within, and also to spread the benchmark suites over many environments in order to reduce the overall execution time. The metric for stability was decided to be RMAD because it is less likely to be influenced by extreme values in the data. The iteration times of each benchmark were calculated into individual RMAD values, and these RMAD scores acted as the metric for every benchmark's stability.

4.4.4 Collecting code coverage

Data from code coverage was needed for the second research question. The intention was to use the data to explore a heuristic that would consider code coverage along with stability. While code coverage is usually gathered dynamically, this was something we wanted to avoid due to the overhead of dynamic approaches. As previously mentioned, we sought to only utilize statically gathered input data for the heuristics. Thus, we selected physical LOC as a statically collected proxy for the dynamic code coverage, which was one of the source code features already collected by our parser (see Section 4.4.2).

4.5 Designing candidate heuristics

This section describes the method and overall strategy of designing the candidate heuristics for both research questions, including how they were evaluated.

4.5.1 RQ1: Predicting benchmark stability

The hypothesis for RQ1 was that static code features could be utilized for the prediction of benchmark stability. As there is relatively sparse research on this subject and microbenchmarks in general, there is little previous knowledge that can be drawn about the predictive power of static code features. The idea was that a feature such as concurrency, for example, could lead to worsened stability due to things such as race conditions, deadlocks, and contention of shared resources. Code smells, such as excessive nesting (nested loops) and large methods, are other features that could affect stability. These lead to worsened code execution which we think could affect benchmark stability for the worse. Based on the related work by Laaber *et al.* [28], it was also interesting to perform a similar experiment in Java to see how the results would compare to the ones they achieved in the Go language.

The candidate heuristics for this research question were designed using regression models, which are statistical models for mapping relationships between a target variable and one or more predictor variables. The goal was to create a relationship between the data of the source code features collected through the static code analysis and the RMAD values used to display stability. This allowed for heuristics where one can attempt to predict the stability of a microbenchmark by simply analyzing its source code, thus avoiding any dynamic overhead, and then using the prediction data to select a suite of the best performing benchmarks. While there are many different types of regression models for various kinds of tasks, we decided to focus on four models for this research question: linear regression, Ridge regression, Random Forest, and Gradient Boosting (read in more detail about these models in Section 2.6). We began with linear regression as a starting point because it is the most simple and basic regression model. It was observed that some of the source code features used as predictors would naturally have a high correlation with each other, a phenomenon called multicollinearity. One example of this is LOC and features such as conditionals or method calls, since the likelihood of most fundamental source code features increases as the scope of the system does. Since linear regression can not account for multicollinearity, we also decided to explore Ridge regression, which essentially is a linear regression model that can handle high correlation between its predictors. We also wanted to explore some non-linear models, which is why we added Random Forest, a non-linear model that is an ensemble of many binary decision trees. Finally, we also wanted to have at least two non-linear models to be able to compare the results between them, so we added Gradient Boosting. This model is similar to Random Forest, but handles the training of the decision trees quite differently. This resulted in four candidate heuristics for RQ1, where each regression model acted as an independent candidate heuristic.

To evaluate the performance of these models, we selected R-squared (R^2) and Mean Absolute Error (MAE) to gauge their predicting capabilities (see Section 2.7). R^2 and MAE are common metrics in statistics and regression models, with both taking a unique approach for evaluating performance. R^2 measures the proportion of variance in the target variable (benchmark stability) that is predictable by the predictor variable (source code features), and ranges between 0 and 1 where a greater score indicates a better performance of the model. MAE describes the average size of errors between the model's predicted stability values and the actual stability values of the benchmarks, and thus the unit of this metric is the same as the target variable's (RMAD). A higher MAE score means a worse prediction capability of the model. While there are many other metrics available for this process, we thought that R^2 and MAE are both easy to understand and calculate, and at the same time give enough insight from different perspectives to effectively evaluate these models.

4.5.2 RQ2: Predicting stable high coverage benchmarks

The heuristic for the second research question combined the stability of the benchmarks, gained from the RQ1 heuristic, with their respective code coverage. The benchmarks were first separately ranked in two lists based on their stability metric (RMAD) and code coverage (LOC), and each benchmark's ranking from both lists was then combined into one conjoined ranking of the benchmarks based on both properties. Intuitively, the combination of these two lists was done by taking the average rank of each benchmark from both lists and then sorting them based on these average ranks. There are many rank aggregation methods available [38], but due to both lists having the same size, and as both stability and code coverage were to have the same weighted importance in this heuristic, we decided to focus on a simple method for aggregation. We put our attention to average rank aggregation since it is both easy to understand and compute, and its non-complex nature fits well with the needs of the combination process. Such an aggregated list would be desirable as it means that those benchmarks are able to notice performance regressions with their stability, and also have an increased likelihood of detecting performance regressions due to their high code coverage, making them more sensitive to any changes affecting the code base.

The aggregated list approach was evaluated through a qualitative comparison. This involved assessing the multiple rankings (stability, code coverage, stability + code coverage) by comparing them to each other. In this process, we tried to identify the strengths and weaknesses of the rankings in order to measure if the combined ranking was better than any of the individual ranks.

5
Implementation

This chapter covers the actual implementation of our method and experiment. The sections below go into further detail about the execution of the data collection, the design stage of the candidate heuristics, and all utilized tools. The implementation can be found at the study's GitHub repository [39].

5.1 Project preparation

Before we could start the experiment, each project of the study objects needed to be set up and prepared. These preparations included making sure the projects could be successfully compiled and built without fault. We also made sure that the unit tests could run and did not encounter any issues during their execution. Finally, we added JMH as a dependency into each project (if the project had not implemented it already), so that benchmarks could successfully be generated with ju2jmh. The commands used to build the projects, run the tests, etc., were all performed with Gradle [13] since that is the common build tool used for all the study objects. Efforts were made to solve any thrown exceptions when running the test suites, but in some cases we had to fully remove a few of the unit tests that could not be resolved. For RxJava, it was the case that the project had already incorporated JMH, and microbenchmarks could be found in the source code of the project. All pre-written microbenchmarks were manually removed to avoid interfering with the experiment and the results of this study.

5.2 Selecting JUnit tests

We created a Python script to identify the unit tests from the selected study objects. The script works by looping through a given directory to find all Java files. The script then looks at each file and locates all present unit tests. This is done by checking if each method is annotated with "@Test", the primary annotation used by JUnit to identify methods as unit tests [22]. For each project, all found JUnit tests and their file location were compiled into a *.txt* file, resulting in three separate *.txt* files.

After a list of all unit tests in each project had been generated, it was time to select which specific subset of unit tests to use in the experiment. Utilizing all of the available tests from each project was not an option due to the time constraint of the study. It would have taken an extensive amount of time to run them all after they

had been converted to benchmarks. Thus, we decided to select 750 random JUnit tests from every study object. An increase in the number of tests would have reduced the overall time for other parts of the study, e.g., designing the candidate heuristics. The selection of the unit tests was done with another Python script written by us for this task specifically. The script used one of the previously generated *.txt* files (containing a project's unit tests and their location) and randomly selected 750 unit tests. The script then generates a new *.txt* containing all selected unit tests and their file location. The script also exports five additional *.txt* files that each contain a section of the JMH command needed to run the benchmarks of the unit tests; this results in each file having a portion of the total benchmarks to run, with all files together containing all the benchmarks. The reason for splitting up the JMH command into smaller sections is because multiple instances of JMH were used to run the benchmarks in parallel, in an effort to reduce the overall execution time of the benchmarks (see Section 5.4.2).

5.3 Parser

To aid us perform a static code analysis and document a specific selection of code features seen in the study object's source code, we wrote a small collection of classes with the ability to read through Java code and keep track of these features [40]. This tool was written in Java with the help of the functionality and capabilities from the JavaParser library [41]. The parser works by providing it with the path to a Java file and the name of a method (JUnit test in this case) in that file to parse. The parser will then read through the source code of the provided method and document all of the selected code features it can find, see Section 4.4.2.1 for a description of these features. In addition, the parser will recursively do the same parsing for all methods that at any time get called in the call tree from the provided method, as long as the method does not belong to an external dependency/library that is not available in the source code (due to JavaParser limitations). This causes a recursion where every method that at some point gets invoked during the execution of the original method also gets parsed, see Figure 5.1 for an overview of how the parser moves throughout the source code. Finally, the data of the documented source code features from each recursively parsed method are summed into one individual value for each future. The Gson library [42] was used to export all found source code features of the JUnit tests to files in JSON format, see Figure 5.2 for an example file. These exported JSON files were utilized to preprocess the code features into a usable state (see Section 5.4.5.1), after which the preprocessed features were used to design the candidate heuristics.

To avoid the possibility of an endless recursion, we implemented a maximum recursion depth the parser will not exceed. This depth was set to a maximum of 1000 recursions for each individual benchmark. We also added a timeout in case the parser got stuck analyzing a benchmark for too long and we set it to 1 minute. After any of these two limits are reached, the parser will continue to the next benchmark. Because of the parsing limits, it is important to note that benchmarks that reach these limits and also utilize a similar set of methods will possibly also have some

similarities in their code analysis result after the parser is done; these limits were reached for less than 1% of the total succeeding microbenchmarks. Additionally, the parser was thoroughly tested to make sure it provided the correct data, see Section 7.4 for more information.

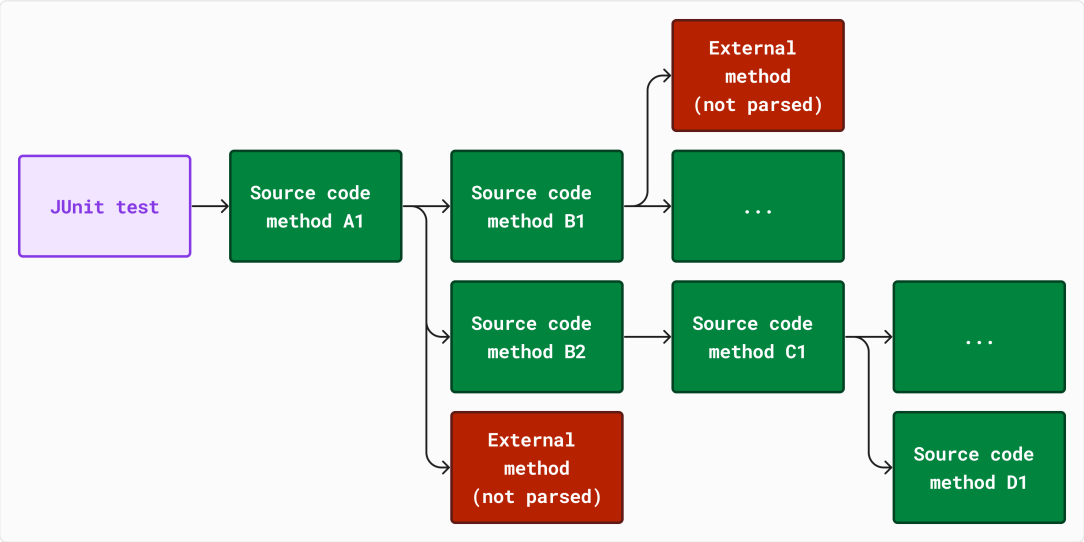

Figure 5.1: Overview of the parser's path while parsing a JUnit test. It begins parsing the JUnit test and then recursively parses through each method in the call tree of the test case. The parser does not document external methods from libraries/dependencies that are not directly available in the source code. All features parsed in the green rectangles (source code methods) are counted for the JUnit test, while the red rectangles (external methods) are not counted.

```
{
  "filePath": "path/to/test",
  "methodName": "junit_test1",
  "stats": {
    "methodCalls": {
      "packageA_method1": 1,
      "packageA_method2": 2,
      "packageB_method1": 5,
      "java.lang.Object.getClass": 5,
      "java.lang.Double.doubleToLongBits": 15,
      "java.math.BigDecimal.compareTo": 23,
      "externalPackage_externalMethod1": 4,
      "externalPackage_externalMethod2": 11
    },
    "objectInstantiations": {
      "packageA_class1": 3,
      "java.lang.Object": 5
    },
    "packageAccesses": {
      "package_A": 6
      "package_B": 5
      "java.lang": 25,
      "java.math": 23,
      "externalPackage": 15
    },
    "numConditionals": 9,
    "numLoops": 3,
    "numNestedLoops": 0,
    "numMethodCalls": 66,
    "numSourceCodeMethodCalls": 51,
    "linesOfCode": 174,
    "linesOfCodeJunitTest": 12
```


Figure 5.2: Example of a JSON output file, for a JUnit test, generated by the parser.

5.4 RQ1: Predicting benchmark stability

The initial section of our experiment was about the first research question, which focuses on exploring a heuristic for predicting stability. This part of the experiment utilized the input data (JUnit tests from study objects) to gather the stability of each JUnit test, and a multitude of steps were taken to process the input data into these stability values. The datasets of stability and the previously gathered source code features were then used in regression models to create candidate heuristics.

5.4.1 Converting JUnit tests to benchmarks

Generating benchmarks from the JUnit tests with the ju2jmh tool required four arguments. These were:

1. The root directory of the source files for the unit tests.
2. The root directory of the compiled class files for the tests.
3. The root directory where the resulting benchmark source files should be generated.
4. A list of fully-qualified test class names to generate benchmarks from.

The first three arguments are unique to every study object because of the variety of each project's structure (placement of files, directories, packages, etc.). These arguments must therefore be manually identified and selected from within each project directory. The fourth and last argument is provided using the contents from the *.txt* file generated by the test selector script. Running ju2jmh with the provided arguments generates copies of the test classes at the location specified by the third argument. These copies contain all original code of the test classes, including the unit tests, but they each also contain a newly created inner class that holds the generated JMH benchmarks (one benchmark for each unit test in the test class).

Several new JMH-specific commands become available when JMH is implemented into a project. One of these commands is called "jmhJar". This command generates a new *.jar* file containing the whole microbenchmark suite within the project, and the benchmarks can be executed by running the *.jar* using any standard JMH arguments. We utilized this command to create *.jar* files for running the benchmarks. Thus, by using the ju2jmh tool and "jmhJar" command on each study object we generated three different *.jar* files containing their respective JMH benchmarks.

5.4.2 Running the benchmarks

Even when a smaller subset of the total unit tests were selected, running these tests still took a long time. This is due to the number of forks, warmup iterations, and measurement iterations used, which together enhance the accuracy and precision of the benchmark results but also extend the total runtime.

5.4.2.1 Execution environment

Google Cloud was used as the platform to run the benchmarks. A cloud-based environment was primarily chosen for two reasons: to gain the ability to reduce the execution time of the benchmarks through multiple instances of JMH running simultaneously; and to avoid running the benchmark suite on our local machines, since the configuration of the OS and local programs could affect the benchmark results. Five identical VM instances were created. The configuration was set to run Ubuntu 20.04 as the OS, with 2 vCPUs, 1 core, 8 GB of memory, and 10 GB of persistent disk storage. Since we had a limited amount of free trial credits to use for the duration of the experiment, this configuration was ultimately decided on as

it landed on a decent price per hour.

Copies of each JMH *.jar* file that was previously generated for each study object were imported into the file system of the VMs. Additionally, the *.txt* files containing the split-up JMH command necessary for running the benchmarks were also imported, with only one section of the command being present in each VM; this was the section of the microbenchmark suite that was going to be run in that specific VM. The splitting of each project's suite into five separate VMs was mainly done to reduce the total runtime of the benchmarks. A single project's benchmark suite was run at a time.

5.4.2.2 JMH arguments

The JMH *.jar* files were run with a specific set of command line options. The goal of these arguments was to maximize the accuracy of the benchmark results while also keeping their execution time reasonably low. Each fork, warmup iteration, and measurement iteration that gets added increases the number of times a benchmark is run. When you have close to two thousand microbenchmarks to run as in this study's experiment, the execution time quickly adds up. We wanted to avoid a situation where we had to wait for many days to weeks for the microbenchmark suites to complete their runs, especially since JMH could fail a run completely despite the usage of certain command line arguments that attempt to avoid such scenarios. At the same time, we needed the benchmark results to be accurate and reliable. This required us to strike a balance between the execution time and result accuracy of the microbenchmarks. But due to the scarcity of research on JMH and benchmarking in Java, we had to find this balance and select the argument values ourselves. To judge the runtime of the benchmarks, we experimented with different value combinations for forks and both types of iterations to maximize benchmark execution time while also keeping to the time constraints of the study. We settled on two separate runs (forks) of the benchmark suites, with 10 warmup iterations and 20 measurement iterations in each run. This way, we achieved a reasonable runtime of under a couple of days to fully run all of the microbenchmarks. In addition, we got a decent amount of total executed benchmark iterations at 20 warmups (2 forks x 10 warmup iterations) and 40 measurements (2 forks x 20 measurement iterations) per benchmark, which increased the overall accuracy and precision of the benchmark results.

The benchmark mode was set to average time. This allowed us to capture the time it took to run each microbenchmark, which is a metric that can later be used to interpret the results. The unit used for the average time was set to nanoseconds, which was the most precise time unit that JMH had to offer. Every benchmark was executed for a minimum of one second per iteration, which resulted in more than a thousand executions per iteration for a majority of our selected benchmarks, since 89.3% of them had an average runtime of less than 1ms. We also used an argument to output the results of the whole run to a *.txt* file. Finally, we utilized the "foe" argument to make sure the whole JMH run should not fail and stop in case a microbenchmark encountered an exception; this was done to avoid having

to rerun the whole suite for the possible failure of one benchmark. Despite adding "foe" as an argument, there were moments where whole suites had to be rerun due to JMH failing completely at some stage of execution. After JMH successfully ran all benchmarks, we tallied the results. In total, a couple of microbenchmarks did fail and were filtered out from the final benchmark results (see Section 5.4.3 for further detail).

Table 5.1: The selected JMH arguments used for the experiment during benchmark execution.

Argument	**Description**	**Selected value**
bm	Benchmark mode.	avgt (average time)
tu	Time unit.	ns (nanosecond)
f	Number of forks.	2
wi	Number of warmup iterations.	10
i	Number of measurement iterations.	20
w	Minimum time to spend on warmup iterations.	1s
r	Minimum time to spend on measurement iterations.	1s
foe	Fail on error.	false
o	Output to file.	*FILE_NAME.txt*

Table 5.1 contains all command line arguments and their values we utilized when running the microbenchmark suites. Below we have also provided an example of a JMH command using our arguments (the command should be treated as a single-line command and be applied as such). A more detailed overview of each argument and what they do can be found in Section 2.1.2 and Table 2.1.

JMH command example (single-line command):

```
java -jar "rxjava-jmh.jar" -bm avgt -tu ns -f 2 -wi 10 -i 20 \
-r 1s -w 1s -foe false -o "rxjava-jmh-output.txt"
```

5.4.3 Interpreting benchmark stability

The results from running the microbenchmark suite can be used to compute the stability of a benchmark. As the result (time in nanoseconds to run the benchmark) of the 40 measurement iterations was gathered for each benchmark, an aggregation of these was used when computing their stability value. Plotting these iteration times can be done to get an idea of the distribution of the data. Looking at Figure 5.3 we determined that the data is not normally distributed, but instead follows a power law where a large number of occurrences (high stability) are more common and a small number of occurrences (low stability) are very rare. This type of distribution deters the usage of a stability metric based on normally distributed data, such as the standard deviation. A metric better suited for power laws is RMAD (see Section 2.4) as it is more robust to outliers, and thus this measure was decided as the metric for stability. A translation of the RMAD formula to the application of benchmarks was performed. In our experiment, two forks are used for executing the benchmarks, and both forks run the measurement of each benchmark 20 times, resulting in 2*20, 40 total iteration times for each benchmark measurement. Inputting this into the formula means that the dataset $X = \{x_1, x_2, \ldots, x_n\}$ represents the combined list of all benchmark iteration times, i.e., the size of X is 40. The computed RMAD value, being in the range $[0, +\infty)$, reflects the stability of a benchmark. A lower value indicates higher stability and a higher value indicates lower stability.

A script was written to interpret each benchmark's stability from the results of running the microbenchmarks. Firstly, the script removes any unneeded parts that are not strictly the results of a benchmark. Secondly, it removes any benchmarks that threw an exception and failed since these do not contain any usable results (see Section 5.4.4). Finally, the script calculates the RMAD value for each microbenchmark by using their respective measurement iteration times, and exports these RMADs to a JSON file. This resulted in all three study objects gaining their own JSON file containing the RMAD of each successfully run microbenchmark from that project.

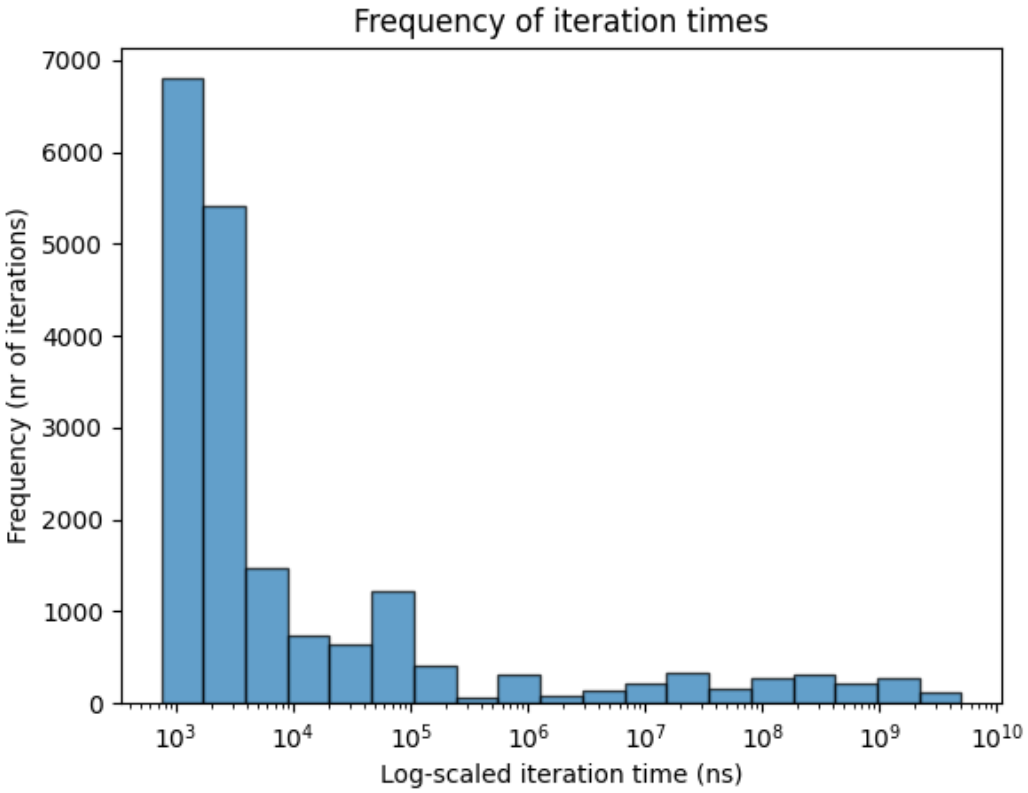

Figure 5.3: Histogram of the benchmark iteration times for the RxJava project.

5.4.4 Failing microbenchmarks

Not all of the selected JUnit tests were successfully utilized. This is because of two primary issues that could occur with these tests: (1) some of them resulted in errors when running their generated benchmarks in JMH, (2) some of them were unable to be parsed correctly during the static code analysis. This resulted in the size of our dataset used throughout the study having a total of 1935 JUnit tests, as seen in Table 5.2, which means 14% of the original test cases could not be utilized. Re-sampling the failing tests was unfortunately not a viable option due to time limitations (further discussed in Section 7.3.3).

Table 5.2: The amount of JUnit tests that were successfully parsed and executed without failure for each project.

Project	**Selected tests**	**Successful tests**	**Failed tests**
RxJava	750	722	28
Mockito	750	665	85
SonarQube	750	548	202
Summary:	**2250**	**1935**	**315**

5.4.5 Creating candidate heuristics

With the computed stability metric values and parsed source code features of 1935 JUnit tests in our dataset, the next step was to create different candidate heuristics based on regression models. Stability was the independent variable, i.e., the one to predict, and the parsed source code features were the dependent variables, i.e., predictor variables. We used various regression models from scikit-learn [43], which is a machine learning library for Python. The chosen models were: Linear regression, Ridge regression, Random Forest, and Gradient Boosting. Naturally, the first goal of the models was to achieve a performance that is better than always picking the most common stability value, corresponding to an R^2 of 0. Achieving this would mean that there exists some predictive power in the source code features. For the regression models, we did not perform any hyperparameter tuning (adjusting the parameters of the models to find optimal results) due to time limitations. Instead, the standard model configurations were used.

5.4.5.1 Preprocessing

Before inputting the data into the prediction models, the dataset was first preprocessed. This was a two-fold process: through Z-score normalization, also called standardization; and by preparing the parsed source code features. Through the Z-score normalization process, the features in the dataset were transformed to a common scale in order to eliminate potential biases caused by scale differences of the features. After this process, the values of the dataset received a mean of zero and a standard deviation of one. Not normalizing the features would limit the performance of the regression models due to their sensitivity to the input features scales.

As for the the parsed source code features (see Section 4.4.2.1), these were preprocessed before being inputted into the regression models. The list of preprocessed features can be seen in Table 5.3. While some of the parsed features could be used directly with little to no preprocessing, such as the features for the number of conditionals, LOC, etc., other features were more heavily modified. For example, the parsed feature “Package accesses” (a list of all accesses to packages) was split into the number of occurrences of several specific packages, and these occurrences acted as their own preprocessed features. The division of packages was mainly based on the importance of the package in relation to the Java language, or the tendency of the contents of the package to affect the stability of benchmarks. For the former, two examples are the packages “java.lang”, which provides fundamental classes and features to Java, and “org.junit”, which supplies the contents for creating JUnit tests. As for the latter, I/O (Input/Output) and concurrency/multithreading can cause execution time variability in the source code, which means they also could cause variability in the stability of benchmarks. We thus wanted to include certain components that relate to execution time variability, such as the package “java.io” which provides an API for reading and writing data, package “java.util.concurrent” which manages certain aspects of concurrency and multithreading in the code, and the “java.lang.Thread” class which allows one to create new threads and perform operations on threads in general.

Table 5.3: The preprocessed features that were inputted to the regression models, created from our selection of parsed source code features. The column "Origin" displays what specific parsed code feature(s) the entry was preprocessed from (see Table 4.2 for all parsed code features).

Feature	**Description**	**Origin**
numConditionals	Number of conditionals.	Conditionals
numLoops	Number of loops.	Loops
numNestedLoops	Number of total nesting levels from nested loops.	Nested loops
linesOfCode	Number of physical LOC.	Lines of code (total)
linesOfCodeJunitTest	Number of physical LOC for the starting method only, i.e., the initial JUnit test	Lines of code (JUnit test)
numMethodCalls	Number of method calls.	Method calls
numSourceCodeMethodCalls	Number of method calls from project's source code, i.e., dependencies are excluded.	Source code method calls
numObjInstantiations	Number of object instantiations.	Object instantiations
numTotalPackages	Number of packages used.	Package accesses
java.lang.Thread	Number of uses of the "java.lang.Thread" class.	Method calls, Object instantiations
java.lang	Number of uses of the "java.lang" package.	Package accesses
java.util	Number of uses of the "java.util" package.	Package accesses
java.util.concurrent	Number of uses of the "java.util.concurrent" package.	Package accesses
java.io	Number of uses of the "java.io" package.	Package accesses
org.junit	Number of uses of the "org.junit" package.	Package accesses

5.4.5.2 Evaluating heuristic models

Evaluation of the heuristic models were performed through cross-validation, using the metrics R^2 and MAE (metrics explained in Section 2.7). Repeated K-fold cross-validation was used as it allows for a robust model evaluation and efficient data utilization, where all of the data in the dataset is put to use. This is beneficial in our case, as we have a relatively small dataset. We chose to use $K = 5$ subsets and repeat the cross-validation 10 times. Repeated K-fold cross-validation works by splitting the dataset into multiple smaller subsets, specifically K subsets having the same size. The model is then trained using $K - 1$ subsets, utilizing the remaining subset as a test set for validation, which is when the performance measures (R^2 and MAE in this case) are computed. Repeating this process K times ensures each subset is used as a test set once. This K-fold cross-validation is repeated multiple times, shuffling the dataset before each repetition, resulting in a more complete evaluation of the model compared to if a single train-test split would have been used. Finally, the respective average of the R^2 and MAE values from the repeated runs are reported as the final scores.

5.5 RQ2: Predicting stable high coverage benchmarks

To recap, the goal for RQ2 was to create a heuristic that is able to find benchmarks having both good stability and high code coverage at the same time. In order to achieve this, the approach (described in Section 4.1.2) was to:

1. Rank benchmarks based on stability using the heuristic from RQ1.
2. Rank benchmarks based on code coverage.
3. Combine the two lists, resulting in a ranked list of the benchmarks having the highest stability and code coverage.

Since the heuristic model from RQ1 needed to be trained before it could make its predictions, we chose to train it on a part of the dataset (80%) and use the remaining 20% for its predictions. These 20% of benchmarks are also used to create the code coverage list and finally the RQ2 list. Visualization for this implementation is shown below in Figure 5.4.

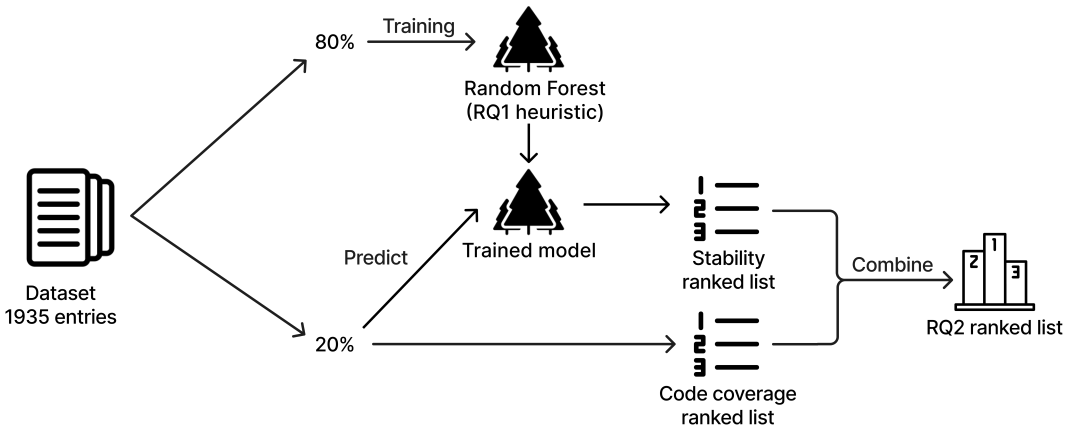

Figure 5.4: Implementation of the RQ2 heuristic. The model from RQ1 is used for creating a stability list, and extracted values of LOC is used for creating a code coverage list. The two lists are then combined, resulting in the ranked list for RQ2.

5.5.1 Rank benchmarks based on stability

The first step in creating the RQ2 heuristic was to rank benchmarks based on their stability. To do this, we trained the best heuristic model found from RQ1 on a portion of our dataset and saved it as a *.sav* file to be able to reuse it whenever needed. Since the best-performing model was a Random Forest model, we used it to make predictions on the remainder of the dataset. The predictions from the model were then turned into an ordered list of the benchmarks it thinks have the best stability, which we refer to as the "stability list".

5.5.2 Rank benchmarks based on code coverage

The second step was to find a "code coverage list", containing an ordered list of the benchmarks having the highest code coverage. The benchmarks in this list were the same 20% used previously in the "stability list". This time the benchmarks are ranked in terms of their LOC, representing the metric for code coverage. The LOC values for all the benchmarks had been previously parsed from our parser (Section 5.3), so all that was needed was to extract these values and order them from highest to lowest LOC.

5.5.3 Combining RQ1 heuristic with code coverage

With the two ranked lists ("stability list" and "code coverage list") being assembled, they could now be combined. Since both lists have the same size and contain the same benchmarks, we opted for a simple method of combining them, namely through the use of average rank aggregation. For example, a benchmark in position 2 in the first list and position 10 in the second list has an average position of 6,

(2 + 10) / 2. An issue that can arise with this method however is when separate combined rankings tie, where the resulting position is the same for two or more entries in the combined list. The way we resolved such ties was to prioritize the ranking of the benchmark with the highest rank in any of the two individual lists. If the highest rank is also of the same value, then the prioritization will randomly select one of the benchmarks.

5.6 Tools

Several tools and libraries were used in this study during the implementation phase to conduct the experiment. This includes two different programming languages with various libraries, two integrated development environments (IDE), one build tool, one microbenchmark converter, and one cloud-based platform. The tools and how they were used are described in more detail below.

5.6.1 Java

Java 17 is the main programming language and version used throughout the study, including compiling and building the study object projects, running the ju2jmh tool, and writing the parser. This specific version of the language was used to make sure all projects were compiled correctly since some of them needed a more recent version of Java to compile without errors. IntelliJ IDEA [44] is an IDE for Java and the primary IDE used in this study, mainly for inspecting and interacting with the files and source code of the study objects. This IDE was also utilized when we created the parser used for the static code analysis. In addition, Gson [42], a Java library made by Google for JSON serialization and deserialization, was utilized to generate JSON output files containing all data gathered by the parser from the static code analysis.

5.6.2 Python

The programming language Python 3.10 is utilized for some of the scripts written for this experiment, including the scripts for finding and selecting a subset of JUnit tests from the study objects to convert into benchmarks. Scripts were also written to interpret the benchmark results and to create plots of the collected data. Some libraries were used to help us write these scripts, this includes matplotlib for plot visualization [45], pandas for data analysis [46], numpy for scientific computing [47], and scikit-learn for regression models [43]. The Visual Studio Code [48] IDE was utilized for writing and running Python scripts.

5.6.3 Gradle

Gradle [13] is a build automation tool and dependency manager for multiple languages, such as Java, Kotlin, and C++. This build tool is utilized by ju2jmh to facilitate the command we used to convert unit tests into microbenchmarks. Ju2jmh recommends projects it targets to use Gradle as well, since it has not yet been tested

for projects with other build tools. Thus, all three of the selected study objects also implement Gradle to manage the execution of the various commands we ran for the projects, including building the projects and creating *.jar* files containing a project's whole microbenchmark suite (command added by JMH when implementing JMH into the project).

5.6.4 JavaParser

JavaParser [41] is a parsing library for Java that allows developers to analyze, transform and generate source code. This library can traverse Java code and look for certain patterns and features of interest; it does this by generating an Abstract Syntax Tree (AST). We utilized version 3.25 of this library to create custom classes with the capability of parsing a selection of source code features of any given method. These classes were applied to the selected study objects to fetch data about their unit test's code features.

5.6.5 junit-to-jmh

junit-to-jmh [1], also known as ju2jmh, is a tool for generating JMH benchmarks from unit tests of type JUnit version 4. This tool was utilized to create the JMH microbenchmarks that were needed for gathering data during the experiment.

5.6.6 JMH

Java Microbenchmark Harness (JMH) [2] is a test harness for Java that allows for building and running microbenchmarks. Since ju2jmh generates microbenchmarks for the JMH framework, this harness needed to be utilized for executing all the benchmarks in the experiment. Thus, JMH was implemented into each study object project if it wasn't present already. When the harness is added to a project, several new Gradle commands get appended. This includes a command that generates a *.jar* with all microbenchmarks within the project.

5.6.7 Google Cloud Platform

The generated microbenchmark suites were all executed inside VMs on Google Cloud Platform [49], mainly to speed up the time needed to run the suite. See Section 5.4.2.1 for specifics and details about the setup used for the VMs.

6
Results

This chapter presents the results of this study and the performed experiment. The data that was gathered from the experiment can be found at a Zenodo repository [50].

6.1 RQ1: Predicting benchmark stability

The point of RQ1 was to see if it was possible to predict the stability of benchmarks that had been generated from JUnit tests using the ju2jmh tool. The input data that the heuristic models used for their predictions were various statically extracted source code features from the underlying JUnit tests.

6.1.1 Stability and code features correlation

Correlation plots were created for the datasets of the projects individually and also for their combined datasets. They showed findings for whether there was any linear correlation between the source code features and the stability. Pearson's correlation coefficient was used, which is a coefficient that measures the linear correlation between two variables [51]. Analyzing Figures 6.1, 6.2, 6.3 and 6.4, which compared the correlation between the stability metric RMAD and the extracted source code features of the JUnit tests of the three study objects, shows that there was little to no correlation found for any of the features. The feature with the highest correlation was "java.io" in the RxJava project (Figure 6.1), which had a correlation of 0.21. A value of 0.21 indicates a weak positive linear relationship to RMAD which could prove at least somewhat useful to the heuristic models. Other than partly this feature no other features stand out as having any relevant correlations. This suggests that a linear model will likely be too simple and that more complex relationships would have to be found to be able to predict the stability using the source code features. One observation from the figures is that when comparing the individual project correlations to the combined projects' correlation, many of the features go from having a positive correlation to RMAD, to having a negative correlation. This phenomenon is known as Simpson's paradox [52], where a trend appears between two variables in subpopulations but then disappears or reverses when the data is aggregated.

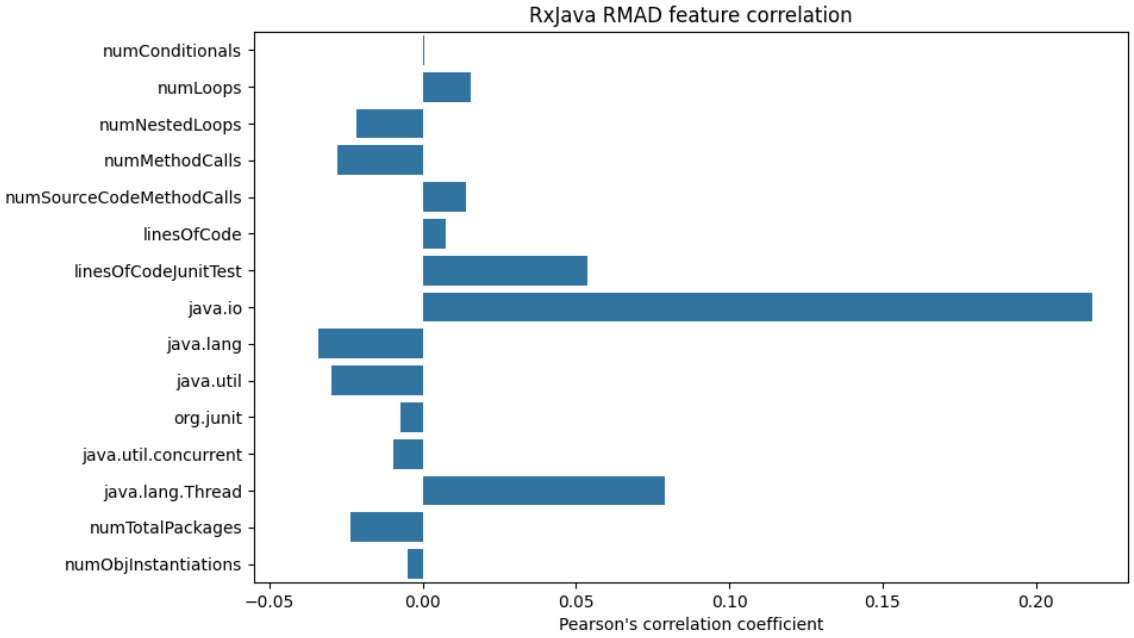

Figure 6.1: Pearson correlation for RxJava between source code features and stability metric RMAD.

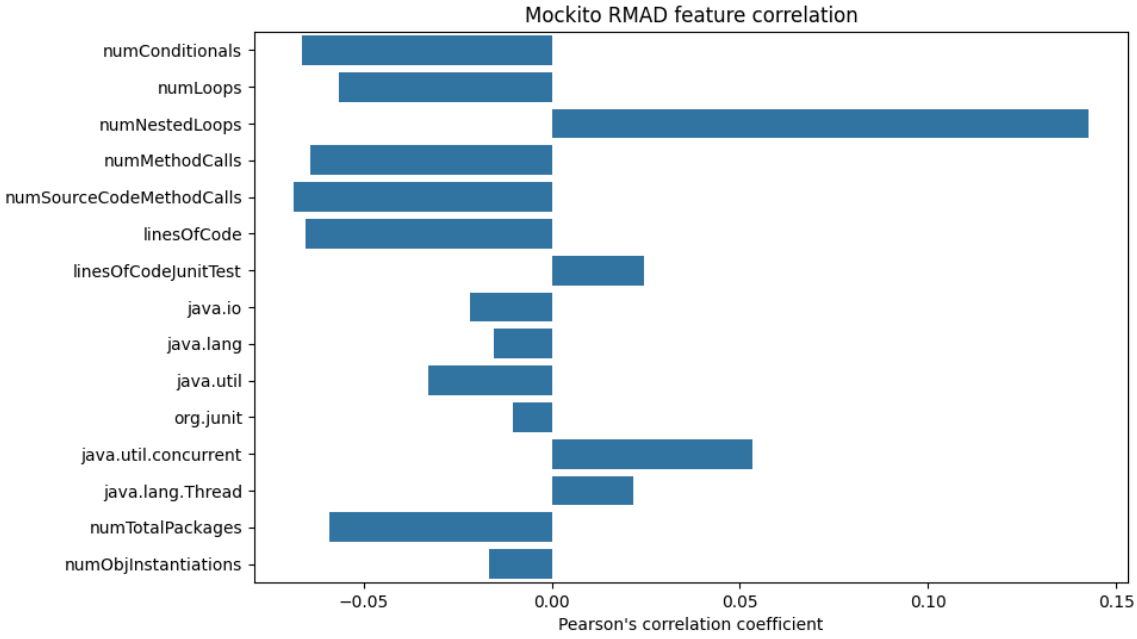

Figure 6.2: Pearson correlation for Mockito between source code features and stability metric RMAD.

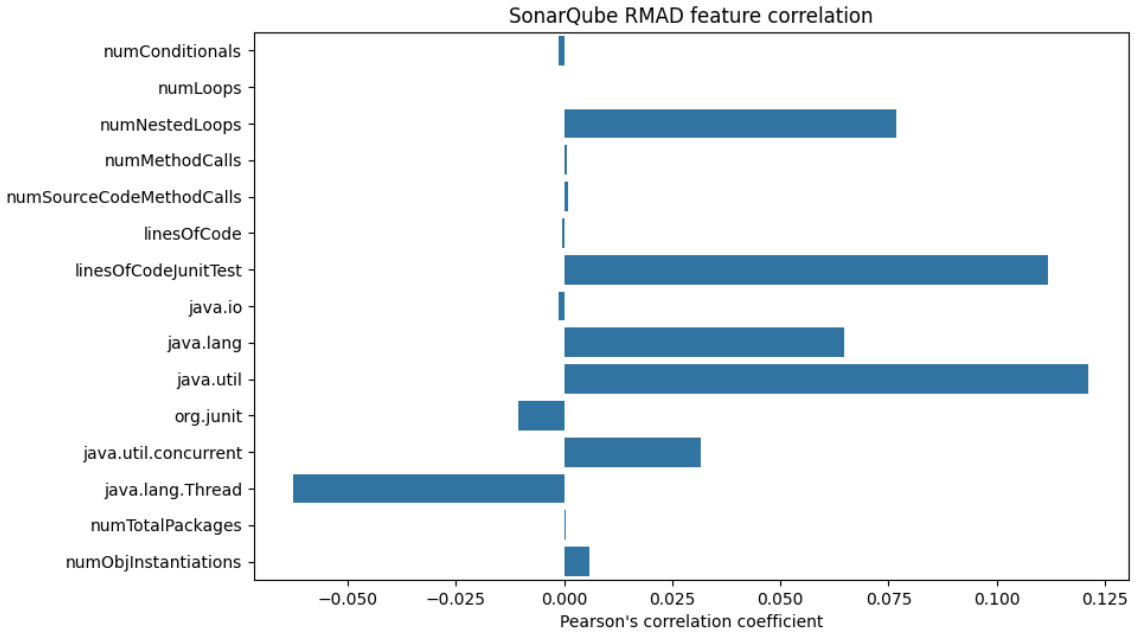

Figure 6.3: Pearson correlation for SonarQube between source code features and stability metric RMAD.

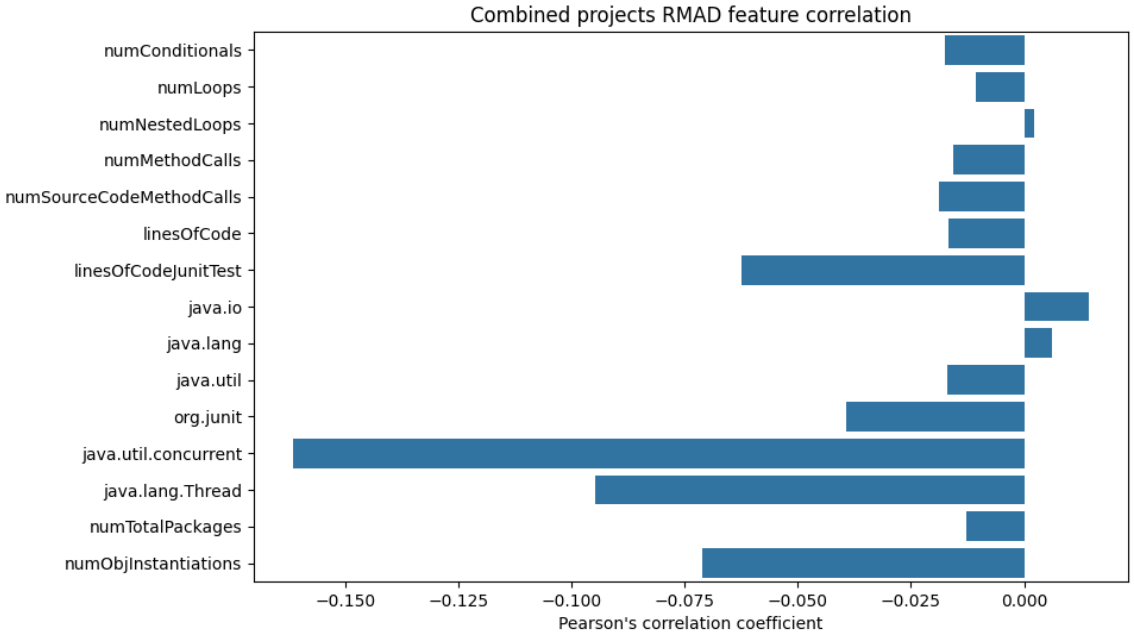

Figure 6.4: Pearson correlation for all projects between source code features and stability metric RMAD.

6.1.2 Heuristic models performance

Performance for the proposed heuristics was measured through the metrics R^2 and MAE. For context, the metric R^2 is usually in the range (0, 1) where 1 indicates a perfect fit of the model, 0 indicates a model that doesn't fit the data at all (performance is equivalent to a horizontal line always predicting the mean), and a negative value indicates performance that is worse than a horizontal line. The other metric measured was MAE, which is the average size of the RMAD errors between the model predictions and the actual RMAD values. For example, an MAE of 5 means that, on average, the model's predictions are off by 5 units, i.e., its predictions would be off by 5% benchmark variability on average.

In order to assess the model's performance, the R^2 and MAE metrics were computed for each heuristic model on the study objects individually (as seen in Tables 6.1, 6.2 and 6.3) and on the study objects combined (as seen in Table 6.4). The computed metric values were acquired through cross-validation using repeated K-fold cross-validation.

Table 6.1: Heuristic performance for RxJava with metrics R^2 (R-squared) and MAE (Mean Absolute Error).

Heuristic	R^2	**MAE**
Linear	-0.047	2.347
Ridge	-0.044	2.462
Random Forest	-0.120	2.449
Gradient Boosting	-0.334	2.499

Table 6.2: Heuristic performance for Mockito with metrics R^2 (R-squared) and MAE (Mean Absolute Error).

Heuristic	R^2	**MAE**
Linear	-6457.521	26.320
Ridge	-8.987	4.561
Random Forest	0.027	3.753
Gradient Boosting	0.005	3.770

Table 6.3: Heuristic performance for SonarQube with metrics R^2 (R-squared) and MAE (Mean Absolute Error).

Heuristic	R^2	**MAE**
Linear	-641.913	10.397
Ridge	-0.040	5.055
Random Forest	-0.029	5.004
Gradient Boosting	-0.010	4.909

Table 6.4: Heuristic performance for all projects with metrics R^2 (R-squared) and MAE (Mean Absolute Error).

Heuristic	R^2	**MAE**
Linear	-3.989	4.259
Ridge	0.061	3.999
Random Forest	0.214	3.491
Gradient Boosting	0.191	3.554

Examining the performance of the heuristics in Tables 6.1, 6.2, 6.3 and 6.4, none of the heuristic models had any real meaningful performance. Most of the heuristics had an R^2 value less than or close to 0, which is equivalent to the performance of a horizontal line that always predicts the mean. There are several possible reasons for these poor results. The simplest and most naive answer would be that the selected source code features in general do not provide enough explanatory power to predict benchmark stability. Other reasons could include the use of insufficient JMH arguments (such as few iterations and low measurement time), a suboptimal parser (it was unable to parse all benchmarks and had a loss of information), and more (see Section 7.3 for further details). From the heuristic models, the best performing one was Random Forest from the combined projects (Table 6.4), with an R^2 value of 0.214 and MAE of 3.491. This R^2 value means that 21.4% of the variance in the stability can be explained by the predictor variables (source code features). While this model performs better than always using the median stability value for its predictions, it is still of low explanatory power. An interesting observation from the tables is that the metrics for each individual project had close to or negative R^2 values, but when using data from all projects the overall R^2 values improved. This suggests that collecting more data from additional projects could improve the results further, something that could have been explored if there had been more time.

6.1.3 Feature importances

The best-performing model from the previous section (Section 2.7) was a Random Forest using the combined dataset, with an R^2 value of 0.214. Performing permutation feature importance on this model was done to see which features it deemed the most important in making its predictions [53]. Table 6.5 shows how significant each feature is for this model. In this table, we see that the most important feature was "java.util.concurrency", having an importance of 15.9%. It represents the total number of times a JUnit test, or any method called during the execution of the test, uses the "java.util.concurrency" package, either by instantiating a class or calling a method from a class in the package. Intuitively, concurrency being the most important feature seems logical, as it involves running multiple threads or processes simultaneously. This can lead to resource contention, non-deterministic scheduling, and synchronization overhead, all of which impact execution times. The feature with the lowest importance for the model was "numNestedLoops", having an importance of 0.6%. Upon further inspection of the data, this low feature importance also makes sense considering that 1833 of the 1935 tests, or 94.7%, had a sum of 0 nested loops in the code called by the tests, making the feature uneventful in most cases.

Table 6.5: Permutation feature importances of the Random Forest model from all projects. Features are ranked based on percentage of importance.

Feature	**Importance (%)**
java.util.concurrency	15.9
java.lang	13.5
linesOfCode	9.9
numConditionals	9.5
numMethodCalls	7.1
java.io	6.7
linesOfCodeJunitTest	6.4
numMethodCalls	6.4
numTotalPackages	6.2
java.util	6.0
numObjInstantiations	5.4
numLoops	3.0
org.junit	2.2
java.lang.Thread	1.2
numNestedLoops	0.6

6.2 RQ2: Predicting stable high coverage benchmarks

RQ2 explored a suitable approach for combining the benchmark's stability prediction from RQ1 with their code coverage. This combination process was done using average rank aggregation, by separately ranking the benchmarks based on stability and code coverage. These two ranked lists were then aggregated into one combined list, creating a new combined rank for each benchmark. When separate microbenchmarks received the same combined rank, the benchmark with the highest position in either of the two individual lists (stability, code coverage) would be prioritized. The prioritization would be random if the highest position was also of the same value. The benchmarks were the same as in the dataset used for evaluation in RQ1 (20% of the total benchmark entries), and this set contained 145 benchmarks from RxJava, 133 from Mockito, and 110 from SonarQube, which sums up to 388 tests that were ranked based on stability and code coverage.

6.2.1 Stability list

For the stability list, RQ1's best-performing model, Random Forest, was utilized to predict the stability of the benchmarks in the dataset. The predictions can be seen in Table 6.6, where the 15 microbenchmarks with the highest stability, i.e. lowest RMAD values, are displayed. It is important to note that despite Random Forest being the best model from RQ1 it still had poor predictions and explanatory power, thus the benchmark's stability values are certainly not very accurate to their real stability. In spite of the mostly even division of benchmarks between the study objects, it is noticeable that RxJava is dominating the table. Naturally, this could be because RxJava generally contains more stable JUnit tests compared to Mockito and SonarQube. It may also be due to how RxJava was used to evaluate ju2jmh during its development [10], which could result in ju2jmh being better optimized for that specific project.

Table 6.6: List of the top 15 microbenchmarks (each a single test case) from all the study objects with highest stability (lowest RMAD). Benchmark names may be abbreviated with "[..]" to indicate excluded sections for brevity.

Rank	Benchmark	RMAD
1.	rxjava3.[..].noUnsubscribeDownstream	0.729
2.	rxjava3.[..].innerWithScalar	0.73
3.	rxjava3.[..].noBackpressureWithInitialValue	0.787
4.	rxjava3.[..].single	0.788
5.	rxjava3.[..].dispose	0.818
6.	rxjava3.[..].SingleOfTypeTest.normal	0.821
7.	rxjava3.[..].reentrantOnNextCancel	0.837
8.	rxjava3.[..].flatMapSingleValueDifferentType	0.840
9.	rxjava3.[..].timespanDefaultSchedulerSize	0.859
10.	rxjava3.[..].nearMaxValueWithoutBackpressure	0.887
11.	rxjava3.[..].concatMapDelayErrorJustSource	0.940
12.	rxjava3.[..].concatMapDelayError	1.010
13.	rxjava3.[..].onCompleteQueued	1.029
14.	rxjava3.[..].sourceFlowableRetry0	1.040
15.	rxjava3.[..].mergeScalar2	1.044

6.2.2 Code coverage list

The code coverage list was ranked based on the LOC of the code called by each benchmark, which was gathered by the parser. The microbenchmarks with the highest LOC are displayed in Table 6.7. As can be seen, a majority of the entries are from Mockito ("mockitousage" included), with some RxJava benchmarks sprinkled throughout. One possible reason for Mockito being so prevalent could be because the project is built for mocking (removing external dependencies from code and surrounding it with a controlled environment, usually by creating dependency-free simulations of classes), which causes the study object's JUnit tests to heavily test these mocking capabilities. The process of mocking can increase LOC since each time a mock object is created it needs to be set up and configured with simulations of the original dependencies it utilized, so it can have the same capabilities as the original object without using its dependencies. This can cause even higher LOC if the class that needs to be mocked is more complex since now more advanced code/dependencies need to be simulated. It is also worth noting how the top two benchmarks have the same LOC, at a staggering value of over 600000, which seems very improbable and unrealistic. The reason for this is likely because the measurement of LOC does not account for repetitions, which causes certain lines of code to be counted multiple times. The use of LOC as a static proxy for code coverage is further discussed in Section 7.4.1. It is also possible that the parser somehow got stuck in an endless loop during the parsing of these two test cases, causing the lines

of code to be continuously incremented until the parser reached one of its two limits (maximum recursion depth, one-minute timeout), see Section 5.3 for further details on these limits.

Table 6.7: List of the top 15 microbenchmarks (each a single test case) from all the study objects with highest code coverage (LOC). Benchmark names may be abbreviated with "[..]" to indicate excluded sections for brevity.

Rank	Benchmark	LOC
1.	mockito.[..].testCharArrayHiddenByObject	633969
2.	mockito.[..].testLongArray	633969
3.	mockito.[..].testNpeForNullElement	126793
4.	mockito.[..].shouldKnowIfObjectsAreEqual	76936
5.	mockito.[..].should_invalidate_null_throwable	24994
6	mockito.[..].paramTyp_return_type_of_values_resolved[..]	12065
7.	rxjava3.[..].badSource	2796
8.	rxjava3.[..].undeliverableUponCancelDelayError	1020
9.	mockito.[..].should_return_empty_array	906
10.	rxjava3.[..].windowAbandonmentCancelsUpstream	823
11.	mockito.[..].should_leave_causing_stack_with_two_spies	758
12.	rxjava3.[..].upstreamErrorAllowsRetry	720
13.	mockitousage.[..].listener	707
14.	rxjava3.[..].doubleOnSubscribe	705
15.	mockitousage.[..].shouldAllowAtLeastZero[..]	680

6.2.3 Combined list

The combination of the rankings from the two previous lists was done by taking their average rank. The aim was to create a heuristic for finding benchmarks with both good stability and high code coverage. Results for the 10 benchmarks with the highest combined rank for each individual study object can be observed in their respective table, Table 6.8 (RxJava), Table 6.9 (Mockito), and Table 6.10 (SonarQube).

Table 6.8: List of the top 10 microbenchmarks from RxJava (each a single test case) with highest combined rank, derived from both stability and code coverage ranks. All of the ranks are relative to benchmarks from the RxJava project. Lower ranks indicate a better performance, where rank 1 is the best performing. Benchmark names may be abbreviated with "[..]" to indicate excluded sections for brevity.

Benchmark	Combined rank	Stability rank	Coverage rank	RMAD	LOC
undeliverableUpon[..]	1.	26.	2.	1.389	1020
innerWithScalar	2.	2.	27.	0.732	480
windowAbandon[..]	3.	29.	3.	1.437	823
timespanDefault[..]	4.	9.	32.	0.859	445
concat[..]ErrorPrefetch	5.	39.	9.	1.624	595
retryUntil	6.	28.	24.	1.415	485
resultSelectorThrows2	7.	47.	10.	1.795	590
concatMapDelayError	8.	12.	49.	1.011	374
mergeScalar2	9.	15.	45.	1.044	396
concat[..]ErrorJust[..]	10.	11.	54.	0.940	358

Table 6.9: List of the top 10 microbenchmarks from Mockito (each a single test case) with highest combined rank, derived from both stability and code coverage ranks. All of the ranks are relative to benchmarks from the Mockito project. Lower ranks indicate a better performance, where rank 1 is the best performing. Benchmark names may be abbreviated with "[..]" to indicate excluded sections for brevity.

Benchmark	Combined rank	Stability rank	Coverage rank	RMAD	LOC
testLongArray	1.	11.	2.	3.952	633969
testCharArray[..]	2.	16.	1.	4.296	633969
shouldKnowIf[..]	3.	22.	4.	4.706	76936
should_invalidate[..]	4.	24.	5.	4.776	24994
paramType_return[..]	5.	31.	6.	5.062	12065
should_verify_with[..]	6.	1.	42.	1.565	219
shouldUseArgument[..]	7.	21.	27.	4.643	388
shouldNotAllow[..]	8.	6.	49.	3.178	209
shouldNotVerify[..]	9.	44.	14.	5.738	566
should_show[..]	10.	32.	26.	5.073	395

Table 6.10: List of the top 10 microbenchmarks from SonarQube (each a single test case) with highest combined rank, derived from both stability and code coverage ranks. All of the ranks are relative to benchmarks from the SonarQube project. Lower ranks indicate a better performance, where rank 1 is the best performing. Benchmark names may be abbreviated with "[..]" to indicate excluded sections for brevity.

Benchmark	Combined rank	Stability rank	Coverage rank	RMAD	LOC
execute[..]NoColumns	1.	22.	2.	5.108	44
execute_whenNew[..]	2.	10.	14.	3.752	19
return_only[..]	3.	15.	13.	4.553	19
dontFailIfNot[..]	4.	19.	31.	4.995	11
should_execute_all[..]	5.	16.	45.	4.723	8
count_single[..]	6.	2.	61.	1.211	6
given_runtime17[..]	7.	12.	52.	4.246	7
should_optimize[..]	8.	26.	44.	5.388	8
throw_exception[..]	9.	6.	66.	2.777	6
isDetected	10.	64.	11.	8.292	22

From the tables, it can be deduced that some reasonable trade-offs between stability and code coverage can be gained through this combination approach. For example, benchmarks "testLongArray" and "testCharArray[..]" from Mockito in Table 6.9 have high combined ranks (1 and 2 respectively) because they achieve a reasonable balance between stability (ranks 11 and 16 respectively) and code coverage (ranks 2 and 1 respectively). Some moderate balance between the two benchmark properties is achieved by other microbenchmarks, such as the top three RxJava entries in Table 6.8, with the difference between the two separate ranks being at most 26 ranks. In addition, there are a few cases where the trade-off is generally low, as seen with benchmarks "execute_whenNew[..]" and "return_only[..]" from SonarQube in Table 6.10 where there is at most a 4 rank difference between the two individual rankings.

More extreme cases of trade-offs exist as well, where there is either high stability and low coverage, or high coverage and low stability. One such example is "should_verify_with[..]" from Mockito in Table 6.9 (combined rank 6). This benchmark has the best stability rank due to its low RMAD (1.565) but suffers in code coverage with a rank of 42. Despite this, it maintains a combined rank of 6, indicating its strong stability compensates for the lower coverage to some extent. It also highlights that while high stability can be achieved with simpler tests, it may not always correlate with comprehensive code coverage. Further examples of this is two benchmarks from SonarQube, "count_single[..]" and "throw_exception[..]", shown in Table 6.10, in addition to "concat[..]ErrorJust[..]" and its neighboring entries in Table 6.8. Overall, such extreme trade-offs indicate that prioritizing either stability

or code coverage alone might not be sufficient for optimally finding regression faults in code. Their inclusion in the top 10 combined ranks emphasizes that even benchmarks with less impressive individual ranks can still be valuable to include in the testing suites.

At the same time, it is worth observing how by looking at the combined rank only, one might assume very good performance from the individual ranks as well. This can sometimes make the combined rank almost look a bit misleading at a glance (even though it correctly follows the rules of the implemented combination approach), especially in cases of extreme trade-offs between stability and coverage. One example of this is the first entry, benchmark "undeliverableUpon[..]", in the RxJava table (Table 6.8). This benchmark has, relatively speaking, very high code coverage, being second in place out of all the RxJava benchmarks. But its stability rank is 26, which is not great, yet also not terrible. Despite this, it gained the first position in the combined rank which, if observed in isolation, could make it seem as if the benchmark performed well in both separate rankings. The same occurrence can be observed for other microbenchmarks, such as for benchmarks "innerWithScalar" (combined rank 2) and "windowAbandon[..]" (combined rank 3) from RxJava, benchmark "shouldKnowIf[..]" (combined rank 3) from Mockito seen in Table 6.9, and benchmark "execute[..]NoColumns" (combined rank 1) from SonarQube seen in Table 6.10.

Furthermore, it is interesting how there appears to be an inverse relationship between stability and code coverage, where a higher rank in one property decreases the rank in the other. This can be noticed for the majority of benchmarks in the tables, since it is rare to find a benchmark that has similar ranks in both aspects. For instance, the benchmark "mergeScalar2" (combined rank 9) from RxJava in Table 6.8 displays this quite clearly; it has a high coverage rank and significantly lower stability rank, resulting in a rank delta of 30. This suggests that higher volumes of code (as measured by LOC), and thus a better coverage rank, might introduce more variability in performance, leading to poorer stability.

7
Discussion

In this chapter, the results of the experiment are summarized and discussed. Additionally, areas that could be of improvement for the study are explored, and the overall validity of the thesis is also examined.

7.1 RQ1

RQ1 asked what a suitable heuristic is for predicting the stability of microbenchmarks using source code features computed through static code analysis. From our results in Figure 6.4, our answer to this question would be that a Random Forest trained on all three study objects is the "most" suitable heuristic. It had an R^2 of 0.214 and MAE of 3.491 which performs better than always predicting using the median value of the dataset, but not much more. We believe the main reason for the heuristics not living up to the standards we had hoped for was because of more optimal approaches in the implementation process not being feasible due to time limitations. We believe that these less ideal approaches added up to ultimately affect the heuristic performance for the worse. Specific information on why the heuristics did not perform well and how better performance could be achieved is discussed in further detail in Section 7.3, where areas of improvement of the study are brought up.

Predicting the stability of microbenchmarks is a difficult task where there are a lot of things that can go wrong. Initially, training data for the stability of benchmarks has to be gathered. Capturing the stability of benchmarks is challenging because the variability can vary, and thus be inconsistent when measured. The variability being captured is affected by aspects such as the underlying hardware being used, the software environment, and the utilized JMH arguments when executing the benchmarks, etc. This means that when capturing the variability of a benchmark, using two different machines with the same setup and software configuration could still result in different variability values. Running the benchmarks also takes a long time because multiple iterations are needed to stabilize the machine to obtain reliable measures of the benchmark execution time. This results in the data-gathering process taking a long time which is especially problematic if benchmarks fail to be executed, thus requiring reruns or choosing new benchmarks altogether. Another challenge emerges due to the parser needed when extracting source code features of the underlying JUnit tests of the benchmarks. In theory, the values of each source code feature would be parsed with perfect precision, but this is challenging to accomplish in practice, see Section 7.3.2 for more details.

Although there weren't promising results from the static code features we selected, it would be interesting to see how well dynamic features would perform. Utilizing dynamic features would allow many more code features to be used, as they cover characteristics of code that can only be observed when the program is executed. Metrics of interest could include code coverage, memory usage, CPU usage, I/O operations, and more. However, using dynamic features does come with a cost, as extracting these features takes significantly more time since all of the unit tests must be run to gather their feature values.

Overall, the results from the heuristic models in RQ1 were somewhat expected. Our thesis followed a similar structure to "Predicting unstable software benchmarks using static source code features" by Laaber *et al.* (see Section 3.1). They had initially attempted a regression-based approach for their experiment but without much success, where the error was especially high. Instead, they transformed the problem into a binary classification problem, predicting if the benchmarks are either stable or unstable, rather than a continuous value for stability. The transformation to a classification problem did result in good prediction performances for their models, but this also makes the problem a lot simpler since the model only has to predict between two classes instead of a continuous range, as is done in this study.

7.2 RQ2

RQ2 sought to explore a suitable approach for combining each benchmark's stability, predicted using RQ1's heuristic, with their code coverage, to create a new heuristic with an equal focus on both properties. The static proxy utilized as a metric for code coverage was LOC, one of the source code features documented by the parser. This combination process was done by creating two separate ordered lists of the microbenchmarks, one ranking based on stability and one for code coverage. These two lists were then aggregated into one combined list, where a benchmark's position in the new list is based on the average rank from the two previous lists. In the scenario where separate microbenchmarks received the same aggregated rank in the combined list, the benchmark with the highest position in either of the two individual lists (stability, code coverage) would be prioritized. If the highest position was also of the same value, then the prioritization would be random.

The balance of this approach in regards to both stability and code coverage without favoring one over the other ensures that the benchmarks provide a broader evaluation of performance and regressions. But in some cases, this balance might not meet certain requirements of the developers or system under test. For example, a slightly higher focus on stability than coverage could be needed in systems where more stability is required, such as in real-time or mission-critical applications. Thus, changing the ranking technique of the approach to give stability a bit higher priority in these situations may produce more appropriate results, and the same holds for code coverage in scenarios where it is of higher importance. Of course, if there is purely a need for one of the properties then benchmarks of that type should be the

focus, but developers might want to adjust the balance of the properties slightly in favor of one or the other. The ability to modify the weighing for either stability or code coverage, based on particular needs, would therefore improve the usefulness of this heuristic in multiple situations, even if the current approach offers a helpful overall evaluation. Because of this balance, it is observed in the results how this approach undervalues certain benchmarks that are highly ranked in one property over the other. For instance, if the most stable microbenchmark (stability rank 1) has far less code coverage (e.g. coverage rank 40), the combined rank might not judge that benchmark highly despite its potential value as the most stable benchmark.

The results also seemingly displayed an inverse relationship between the stability and code coverage of benchmarks. For the majority of benchmarks in the dataset, an increase in stability led to a decrease in coverage, and vice versa; there were very few benchmarks that were similarly ranked in both properties. This can be due to how higher code coverage often means more extensive and complex test cases, which might introduce variability and thus affect stability negatively. JUnit tests that are more extensive may also put a higher strain on the system or its resources, causing inconsistency in performance. This inverse relationship could therefore be a possible way to predict stability, i.e., by looking for benchmarks with low code coverage to find stable benchmarks (although the results for RQ1 did not suggest this was the case for this specific experiment).

Summarized, the technique studied in this thesis for combining the stability and code coverage of benchmarks presents itself as an overall suitable approach. While the approach is fundamentally simple to understand and implement, the results showed it can appropriately rank microbenchmarks based on both properties as long as they are needed to be equally valued. In general, the heuristic demonstrates a balance in trade-offs between stability and coverage, where the strengths of one property compensate for the disadvantages of the other. By the nature of the heuristic, the approach is not suitable when either of the two benchmark properties is of higher importance for a system. Additionally, benchmarks that are valuable to just one of the two properties might get less overall recognition by the heuristic due to the lack of value in the other property. While RQ2 specifically sought a heuristic with an equal focus on both properties, it could still be worthwhile to expand this approach by allowing the heuristic to weigh stability and code coverage separately. This would allow developers to tailor the importance of either benchmark property during the combination process, so the specific requirements of the system are more appropriately tested.

7.3 Areas of improvement

After conducting the experiment, several lessons were learned. This section outlines multiple improvements and recommendations to the study's method based on what was learned so that it can be performed better in the future. The improvements regard various aspects of the experiment, such as the parser utilized for static code analysis and the arguments used for JMH.

7.3.1 Improvements to JMH arguments

One possible improvement could be to the JMH arguments utilized during the experiment (see Section 5.4.2.2 for the used arguments). JMH and microbenchmarking in Java are not a typical field in which research is conducted. Thus, work and papers regarding this area are quite scarce, especially in comparison to more common topics such as cloud computing, machine learning, Artificial Intelligence (AI), and cybersecurity. Because of this, the appropriate (or even reliable) arguments to use for the execution of the benchmark suites were not readily available. These arguments had to be experimented on as a result, specifically in regards to forks, warmup iterations, and measurement iterations. Due to the time constraints of the thesis and how benchmark suites can run for multiple days to weeks in certain events, a balance between execution time and result accuracy needed to be held. A low execution time was sought while also keeping the results from the suites reliable at the same time. Thus, if the run time of the benchmarks is not an issue, an improvement would be to increase the number of forks and iterations, which should improve the result accuracy from JMH. The minimum time spent per iteration could also be improved. Our selected value for minimum time resulted in more than a thousand benchmark executions per iteration for 89.3% of the microbenchmarks. Increasing the minimum time would, again, mean a longer execution time for the suites, but would as a result also mean a higher amount of benchmark runs per iteration, further increasing the result's precision and reliability.

7.3.2 Improvements to parser

Something this study could have improved is the data gathering of the source code features. The parser that was developed for this study can only function on files containing readable Java source code, which is due to the JavaParser [41] library that was used to create the parser; the library can only analyze source code in Java. This causes issues with the dependencies of the projects, since the dependencies are loaded as *.jar* files through the Gradle build tool. These *.jar* files do not contain the Java source code, rather they carry the compiled Java bytecode of the Java source code that is then directly executed by the JVM. This meant that the parser could not document any source code features contained within the study object's dependencies, which resulted in the loss of a lot of potential context and information in relation to each project's code features. While there are a few decompilers for *.jar* files out there, such as JD-GUI [54] and Vineflower [55], these tools only generate source code that is similar to the original, thus there might be some loss of

information or clarity, especially in areas where the code was heavily optimized by the compiler. Additionally, the time constraints of this study made the exploration of *.jar* decompilers not feasible, as a lot of time had to be spent on creating the parser itself and writing test cases to examine its accuracy. One improvement could therefore be to the parser, so it gains the capability of also parsing the dependencies of a project, whether it be through *.jar* decompilers or some other means. This improvement could also come in the shape of utilizing a whole different source code parser that has this capability. While a free parser with this study's specific needs could not be found, it is still very possible that a non-free or proprietary option exists.

A second improvement to the parser could be for it to be able to manage even more variations of the selected code features. While 125 JUnit tests were written to verify the correctness of the parser, there are just too many variations, edge cases, and variables to test and account for in regard to all of the selected code features. Therefore, only the most common and typical configurations of these code features were tested, and this proved to be mostly enough with a few issues in certain cases and more complex configurations. More edge cases and variations could be tested, however, to ensure the parser can handle further code feature configurations within methods. Such an improvement would result in a reduction in the amount of failed microbenchmarks during the parsing step. It would also be an improvement if the parser could document a separate LOC feature that does not count repetitions of code lines. This code feature would be a more accurate metric for the actual code coverage of the benchmarks, and could thus be applied to RQ2 to improve the result of the heuristic.

7.3.3 Improvements to data quantity

Further improvements could have been made to the quantity of data utilized during this thesis. As seen in Table 5.2, in total 2250 JUnit tests were selected, and 315 (14%) of them failed due to issues during benchmark execution or the source code parsing. These 315 test cases could therefore not be utilized during the experiment. Efforts could have been made to reduce the percentage of unusable tests, for example by re-sampling the test cases that failed at either stage. This was primarily not done because a new execution of the benchmark suite containing the newly selected tests would need to take place, and even then there is no guarantee that the tests would not fail once more. Essentially, this situation required a process of continuously re-sampling JUnit tests and executing their benchmarks until a certain amount of usable test cases were gained. It was deemed this process would not be feasible due to the study's time constraints, and it was decided that this time could instead be spent on other areas of the study, such as creating the candidate heuristics. One way to possibly remedy this issue, even if just slightly, without re-selecting any tests, would be to either select more study objects or simply select more JUnit tests from the three already existing projects; in either case, this would result in an overall increase of test cases, which could possibly have reduced the percentage of failing microbenchmarks, thus increasing the data quantity.

7.3.4 Improvements to regression models

The process of creating the regression models during RQ1 could have seen a few improvements. Mainly, the time limitations of this thesis did not allow for hyperparameter tuning (also known as hyperparameter optimization), which is the act of adjusting the parameters of a machine learning model to find optimal results. During this process, the model is trained in stages using various sets of hyperparameters, which can be very time-consuming because of the high number of parameters (and their sets of values) being assessed. This resulted in the standard configuration of each model being used instead. Thus, it would be a worthwhile improvement to RQ1 to perform some method of hyperparameter tuning for the regression models. Further improvements could also have been made to the selection of preprocessed features that were utilized for the models. For instance, an even broader selection of packages or classes could be added as features, since Java has other noteworthy components that were not analyzed during this thesis. Additionally, a more rigorous feature selection process could have been implemented, such as continuously evaluating feature importances throughout the heuristic creation process. This technique would have helped to remove redundant or less informative features in order to reduce the complexity of the models and thus improve their performance. Again, none of these suggested improvements could be applied during the study due to lack of time.

7.4 Threats to validity

This section will cover validity threats concerning this study. The subsections below will elaborate on two primary threats to validity: internal validity, which relates to the validity of the study design; and external validity, which examines the validity of how the study's findings can be generalized and applied outside of the specific context of the thesis.

7.4.1 Internal validity

This study has some internal validity threats. The first one relates to the specialized parser tool written for the static code analysis. A collection of roughly 125 JUnit tests was written to make sure the tool documented the source code features correctly. The tests checked that the parser could provide correct data and information for all of the selected source code features (see Section 4.4.2.1), and that no bugs were present in the implementation. Stub classes containing a multitude of methods with various arrangements and cases for all of the code features were created. The stub methods in these classes were passed into the parser, where the tool would collect the source code data. This data was then controlled to match and be accurate to how the stub method was implemented. Despite these efforts, there are simply too many edge cases and variables to account for in regard to all of the selected code features. Thus, the best possible efforts were made, considering the time constraint of the thesis, to test the most common configurations of these features and make sure they would be documented correctly.

Another aspect that may be a threat is how some of the data from the study objects could not be utilized. The issue was two-fold: certain JUnit tests could not be executed as benchmarks after being converted to microbenchmarks with ju2jmh, because they would fail and throw an exception without completing the execution; and certain JUnit tests could not be parsed using the specialized parser, either because the parser or the JavaParser library could not understand a given method's configuration in terms of source code features. For example, JavaParser had in some instances difficulties parsing certain generic types, lambda functions, and ternary operations. Essentially, a failed JUnit test could either not provide the needed stability value or source code data collected from the static code analysis, both of which were instrumental to the implementation of the experiment. As Table 5.2 shows, 315 JUnit tests out of the 2250 selected could not be utilized. This means that the data of 14% of the selected JUnit tests were of no use during the experiment, and thus had to be removed. There is a chance that any of the removed JUnit tests contained valuable data for the study, which would mean their removal could have affected the results of the experiment. At the same time, almost 2000 test cases remained which is a significant amount of data entries. Because of the large difference in the amount of utilized test cases compared to the number of removed test cases, the removal of these tests should not have an overly significant impact on the study results.

One additional internal threat is related to the selected values for the JMH arguments during benchmark execution, see Section 5.4.2.2 for in-depth details about the arguments. This is due to the lack of studies on JMH and Java benchmarking in general, which made it difficult to know the appropriate values to utilize. One of the arguments that could be a threat is the minimum time for executing each measurement and warmup iteration, and this was put at 1 second. This means that each iteration was run for a minimum of 1 second, after which it would continue to the next iteration as soon as the currently running benchmark was complete. This allowed for 89.3% of the microbenchmarks to execute over a thousand times per iteration since they had an average runtime of less than 1 ms. However, only 98.7% of the benchmarks had an average execution time below 1s, which means that 1.3% of the microbenchmarks only had a chance to run once per iteration because no time was left for execution in each iteration. However, since this only affected a very minor section of the total benchmarks, it should not have a major effect on the end results of this study. Additionally, the selected number of iterations per fork (10 warmups, 20 measurements) had to be determined through experimentation. Thus, there is a possibility that the selected amount might not be high enough for this thesis's needs, but this is difficult to assert. Nonetheless, best efforts were made to put these values as high as possible to maximize the precision and accuracy of the benchmark results, while also keeping the execution time at a couple of days at most due to time limitations.

Furthermore, the usage of a benchmark's LOC, a static metric, as a proxy for a benchmark's code coverage, a dynamic property, could also be seen as an internal threat. LOC was utilized as the metric for code coverage, specifically line coverage, because one of the goals of the study was for the candidate heuristics to only rely on static code analysis, which meant avoiding the usage of dynamically collected data when the heuristics needed to be executed. This goal was set due to the common overhead of dynamic approaches, and by avoiding such overheads the heuristics would be faster in practice and easier for developers to implement. Using LOC as a proxy for line coverage is a naive approach, since the LOC for a piece of code will typically be larger than the actual number of lines that are executed during runtime, due to how execution paths can vary based on initial arguments and dynamic code variables. It is also important to note that LOC does not exclusively document unique code lines and thus does not account for possible line repetitions, which creates further discrepancies between LOC and the actual code coverage. For example, this repetitive counting of lines would mean that the code coverage ranking (see Section 6.2.2) could favor methods with multiple calls to the same method over test cases that call different methods (calling more unique methods increases the code coverage). At the same time, LOC is quite easy and effortless to both understand and compute, making it a more practical metric for the heuristics in this study, which is the main reason it was selected. Cyclomatic complexity was also considered as a static proxy for code coverage. But due to its inherent difficulty to calculate in comparison with LOC, it was ultimately decided against. Much greater effort would be needed to develop the parser to calculate the cyclomatic complexity of each benchmark, in addition to verifying it does so accurately and reliably. The experiment was already under time constraints, and these efforts would unfortunately take away important time from other parts of the experiment. Because of the dynamic nature of code coverage, calculating it accurately without executing the code is a difficult task. Nevertheless, measures were taken to increase the reliability of using LOC as a proxy for the code coverage of the benchmarks. This mainly included making sure the parser did not count code documentation as part of the LOC, and writing several JUnit tests to verify that various configurations of documentation were ignored. Code lines only containing curly brackets ("{", "}") were also ignored since these symbols are just used to signify the start/end of a code segment and hold no actual executable code. Finally, blank or empty lines were not counted for the LOC either. Despite these efforts, the collected LOC will still be higher than that of the actual line coverage of each microbenchmark. Avoiding repetitive counting of the same code lines is something that would further improve the proxy toward line coverage, but was not added due to time limitations. It is also unknown how much the parser's overhead would increase in order to only count unique lines, and this is especially important since overhead was an aspect the study sought to minimize. Therefore, the implementation of LOC and its use as a static proxy for code coverage may affect the results of the experiment to not be entirely accurate.

7.4.2 External validity

In addition to the internal threats to validity as outlined previously, this thesis has some external threats as well. One of these threats is that all of the results from this thesis are based on open-source projects from GitHub [33]. Thus, it is uncertain how applicable and valid the results are in other environments and contexts, such as for projects available on other similar Git repository platforms or industrial and proprietary software. It is also important to note that only three open-source projects out of the tens of thousands available on GitHub were utilized throughout the study. This adds further uncertainty for the overall generalizability of the study results for all open-source projects as well. The generalizability can be increased by conducting the experiment with more open-source projects as study objects.

Some of the source code features selected for the study are only applicable to certain categories of programming languages (see Section 4.4.2.1), which could be a threat. For example, the feature "Object instantiations" is only applicable to Object-Oriented programming languages, while "Package accesses" is specifically a feature for JVM languages. This means that the applicability of the results are dependent on the environment in question, and also that the results themselves could differ based on which programming language is being applied.

Furthermore, this study is reliant and established upon the JMH framework and their implementation of benchmarking. JMH is also a JVM-specific framework, thus it is only applicable to certain programming languages such as Java. While the benchmarking properties in this study (stability, code coverage) are not specific to JMH and can be utilized outside of the Java language as well, it is important to note that the results may vary when these properties are applied to other programming languages and benchmarking frameworks.

Another possible threat to external validity is that this study is primarily focused on Java version 17, so how well the study adapts to previous Java versions is questionable. Additionally, Java is a language that is continuously developed and altered. It is therefore debatable how valid the end results are for future versions of Java, as differences in the language or the JVM could possibly have a significant impact on microbenchmarks and benchmarking overall.

8

Conclusion

This thesis explored a suitable heuristic for predicting the stability of microbenchmarks using only code features from static code analysis. It also explored a second heuristic for a suitable approach for combining the stability of benchmarks with their code coverage. The primary purpose of these two heuristics is to determine the Java unit tests to utilize when creating a microbenchmark suite focused on stability (first heuristic), or stability and code coverage simultaneously (second heuristic), with the goal of minimizing benchmark suites and thus reducing their execution time. For the heuristics to execute, only data from a static code analysis is to be needed. This allows the heuristics to avoid the common overhead of dynamic approaches, which makes them easier for developers to implement and more practical to execute. The exploration of these two heuristics was done through a laboratory experiment with three study objects, using a subset of their unit tests converted to benchmarks.

The first heuristic was created through regression models, where four separate models were explored. These models utilized source code features as predictors and the actual stability values of the benchmarks as the target. Each model was analyzed and evaluated using the metrics R^2 and MAE, two separate measures that uniquely convey a model's ability for accurate predictions. The experiment resulted in a Random Forest model being the most suitable heuristic for predicting stability. The model gained a R^2 score of 0.214 and an MAE of 3.491, which means an overall poor performance with low predictive capabilities. Multiple reasons for the weak performance were observed, such as a suboptimal tool used to perform the static code analysis, lack of hyperparameter tuning for the regression models, some amount of failing benchmarks that thus went unutilized, and an uncertain configuration for the arguments used to execute the benchmarks. It is therefore worthwhile to implement these improvements to see how they affect the resulting heuristic.

The second heuristic was designed using average rank aggregation, where benchmarks are separately ranked based on stability and code coverage, and then combined into one singular rank. This approach was evaluated through a qualitative comparison and showed promising results. Generally, the combination approach reflected a balance where strengths in either stability or coverage compensated for lesser performance in the other. However, the heuristic can only be applied in scenarios where stability and coverage are of equal importance to the benchmark suite. This is due to the combination approach judging the two benchmark properties equally. Thus, it could also be beneficial to expand the heuristic so the user can weigh the importance of stability and code coverage individually.

8.1 Limitations

This study had some defined limitations, mainly in terms of scope. The scope had to be limited because of the time constraints of this thesis. These time constraints were mainly due to how benchmark suites can execute for many days to weeks, therefore limitations had to be put in place so the needed data could be collected and the thesis could be conducted and finalized within the given time frame. Overall, these time constraints hindered multiple areas of the thesis's methods, such as the number of test cases that could be selected and studied, and the amount of time that could be spent on creating and verifying the static code analysis parser. It also limited the arguments that could be used when executing the microbenchmark suites since the suites could not run for too long.

Firstly, one of the defined limitations is how the study chose to only focus on the programming language Java when conducting the experiment. The main reason is that Java has a well-defined benchmarking framework, JMH (Java Microbenchmark Harness), which provides a reliable way of conducting performance measurements of source code. In addition, Java is the only studied language due to the aforementioned time constraints of this thesis; conducting the study and its experiment on other programming languages would prove to be a challenge within the time that was available. Secondly, as part of the research method used for this study, only three separate open-source Java projects were utilized. Using a larger set of study objects could have been beneficial for the study since data would be gained from a more diverse set and thus possibly become more reliable, but this was not feasible because of the time limitations. The added complexity of exploring more projects would act as a hindrance to the experiment being conducted in a timely manner, thus resulting in a scenario where the reliability of the achieved results could be questioned. Finally, this study only researches microbenchmark suites focused on the properties of stability and code coverage. Other properties for benchmarks were not studied or experimented on.

8.2 Future work

This thesis revealed multiple areas of interest for additional study, both in regards to possible research areas that could be intriguing to explore and improvements to the methods applied by this study (see Section 7.3 for more in-depth details on areas of improvement). Firstly, while this study focused on a specific selection of regression models for predicting stability, it could be interesting to explore other regression models as well. Additionally, it could be worth utilizing more sophisticated models during the heuristic creation process, possibly even delving into a higher degree of machine learning, such as using Support Vector Machines (SVM) and Neural Networks. Exploring and implementing further complex approaches for predicting stability could also give some insights, such as developing new ensemble methods that utilize multiple different models and techniques in tandem, since they may better capture any possible correlations in the dataset.

The time constraints of this thesis limited several aspects of the study. It could thus be interesting to repeat the study experiment in an environment where time is not a restriction. This could, for example, allow for the experiment to execute more benchmarks in order to gain greater quantities of data and utilize more study objects to further diversify the benchmarks. Further time could also allow for the JMH arguments used during benchmark execution to be improved, and for the regression models to receive hyperparameter tuning to improve their performance. Despite the cost of increased time to perform the experiment, the exploration of such improvements could prove to be insightful as it might lead to different results. Furthermore, the parser that was developed for this study was not perfect, once again due to time limitations, which resulted in a few key improvements it could receive. Thus, it could be worthwhile to implement the discussed improvements and repeat the experiment to see how it affects the study results. Otherwise, an alternative to the parser could also be explored for the same purpose; any tool that can perform a static code analysis and gather the needed source code features would suffice.

In addition, it might be interesting to explore further improvements to the RQ2 heuristic. The combination approach is not suitable when one of the two benchmark properties is of greater significance to the benchmark suite. Expanding this approach to allow the heuristic to evaluate stability and code coverage independently may thus be beneficial. This would enable developers to adjust the weight of each benchmark property during the combination process, resulting in more appropriate testing of the unique needs of every system.

Finally, it could also be of interest to explore other metrics for benchmark stability and code coverage. RMAD was selected as the metric for stability, but it would intriguing to look at how RMAD's simpler variant, MAD, would compare. Other different metrics and statistics are also available for capturing a benchmark's variability, such as CV, and these could be investigated for future work as well. As for code coverage, other coverage criteria may be worthwhile to look at, such as branch coverage, function coverage, or statement coverage. In such cases, if the goal of statically collected data is to be continued, new proxies would need to be chosen based on the selected criteria. While it was not the goal of this study to explore dynamic approaches, it is definitely also a path that could be visited in regard to code coverage. For example, a tool such as JaCoCo [25] could be utilized to capture the actual dynamic code coverage of the benchmarks, which would eliminate the need for a proxy for code coverage altogether. This should result in a more reliable and accurate heuristic for the research question, with the consequence of losing the static nature it previously had.

Bibliography

[1] N. Alexandersson, junit-to-jmh*, GitHub*. [Online]. Available: https://github.com/alniniclas/junit-to-jmh/ (Accessed: 26/03/2024).

[2] OpenJDK, Java Microbenchmark Harness (JMH)*, GitHub*. [Online]. Available: https://github.com/openjdk/jmh (Accessed: 26/03/2024).

[3] M. Jangali, Y. Tang, N. Alexandersson, P. Leitner, J. Yang, and W. Shang, "Automated generation and evaluation of jmh microbenchmark suites from unit tests," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 1704–1725, 2023. DOI: 10.1109/TSE.2022.3188005.

[4] C. Laaber and P. Leitner, "An evaluation of open-source software microbenchmark suites for continuous performance assessment," *Proceedings of the 15th International Conference on Mining Software Repositories*, pp. 119–130, 2018. DOI: 10.1145/3196398.3196407.

[5] H. Hemmati, "How effective are code coverage criteria?" In *2015 IEEE International Conference on Software Quality, Reliability and Security*, 2015, pp. 151–156. DOI: 10.1109/QRS.2015.30.

[6] M. Lipow, "Number of faults per line of code," *IEEE Transactions on Software Engineering*, vol. SE-8, no. 4, pp. 437–439, 1982. DOI: 10.1109/TSE.1982.235579.

[7] A. S. Foundation, Apache JMeter. [Online]. Available: https://jmeter.apache.org/ (Accessed: 27/04/2024).

[8] OpenText, OpenText LoadRunner Professional. [Online]. Available: https://www.opentext.com/products/loadrunner-professional (Accessed: 27/04/2024).

[9] R. Khatchadourian, Y. Tang, M. Bagherzadeh, and S. Ahmed, "Safe automated refactoring for intelligent parallelization of java 8 streams," Sep. 2020. DOI: 10.1016/j.scico.2020.102476.

[10] N. Alexandersson, "Junit-to-jmh: Automatic generation of performance benchmarks from existing unit tests in java," Chalmers University of Technology, 2022, https://hdl.handle.net/20.500.12380/305702.

[11] D. Graham, R. Black, and E. van Veenendaal, *Foundations of Software Testing ISTQB Certification, 4th edition*. Cengage Learning, 2021, ISBN: 9780357884157. [Online]. Available: https://books.google.se/books?id=mOwxEAAAQBAJ.

[12] J. Y. Gil, K. Lenz, and Y. Shimron, "A microbenchmark case study and lessons learned," in *Proceedings of the Compilation of the Co-Located Workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, & VMIL'11*, 2011, pp. 297–308. DOI: 10.1145/2095050.2095100.

[13] Gradle, Gradle Build Tool. [Online]. Available: https://gradle.org/ (Accessed: 21/03/2024).

[14] A. S. Foundation, Apache Maven. [Online]. Available: https://maven.apache.org/ (Accessed: 21/03/2024).

[15] Valloric, jmh command line options*, JMH Playground, GitHub*. [Online]. Available: https://github.com/Valloric/jmh-playground/blob/master/README.md (Accessed: 21/03/2024).

[16] Oracle, Java. [Online]. Available: https://www.java.com/ (Accessed: 25/04/2024).

[17] Oracle, Java Virtual Machine Guide*, 1 Java Virtual Machine Technology Overview*. [Online]. Available: https://docs.oracle.com/en/java/javase/21/vm/java-virtual-machine-technology-overview.html (Accessed: 25/04/2024).

[18] JetBrains, Kotlin. [Online]. Available: https://kotlinlang.org/ (Accessed: 25/04/2024).

[19] Oracle, Java Developer's Guide*, 9.1 Oracle JVM Just-in-Time Compiler (JIT)*. [Online]. Available: https://docs.oracle.com/en/database/oracle/oracle-database/21/jjdev/Oracle-JVM-JIT.html (Accessed: 25/04/2024).

[20] Oracle, Java Garbage Collection Basics. [Online]. Available: https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html (Accessed: 25/04/2024).

[21] JUnit, JUnit. [Online]. Available: https://junit.org/ (Accessed: 24/04/2024).

[22] JUnit, Annotation Type Test*, junit.org*. [Online]. Available: https://junit.org/junit4/javadoc/latest/org/junit/Test.html (Accessed: 02/04/2024).

[23] C. N. Arachchige, L. A. Prendergast, and R. G. Staudte, "Robust analogs to the coefficient of variation," *Journal of Applied Statistics*, vol. 49, no. 2, pp. 268–290, 2020. DOI: 10.1080/02664763.2020.1808599.

[24] S. Bhatia and J. Malhotra, "A survey on impact of lines of code on software complexity," Aug. 2014, pp. 1–4. DOI: 10.1109/ICAETR.2014.7012875.

[25] EclEmma, JaCoCo Java Code Coverage Library. [Online]. Available: https://www.eclemma.org/jacoco/ (Accessed: 27/04/2024).

[26] JaCoCo, Implementation Design. [Online]. Available: https://www.jacoco.org/jacoco/trunk/doc/implementation.html (Accessed: 27/04/2024).

[27] A. Lindholm, N. Wahlström, F. Lindsten, and T. B. Schön, *Machine Learning - A First Course for Engineers and Scientists*. Cambridge University Press, 2022. [Online]. Available: https://smlbook.org.

[28] C. Laaber, M. Basmaci, and P. Salza, "Predicting unstable software benchmarks using static source code features," *Empirical Software Engineering*, vol. 26, no. 6, Nov. 2021. DOI: 10.1007/s10664-021-09996-y.

[29] J. Chen and W. Shang, "An exploratory study of performance regression introducing code changes," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2017, pp. 341–352. DOI: 10.1109/ICSME.2017.13.

[30] D. Costa, C.-P. Bezemer, P. Leitner, and A. Andrzejak, "What's wrong with my benchmark results? studying bad practices in jmh benchmarks," *IEEE Transactions on Software Engineering*, vol. 47, no. 7, pp. 1452–1467, 2021. DOI: 10.1109/TSE.2019.2925345.

[31] A. Path, "Optimizing the generation of java jmh benchmarks," University of Gothenburg, 2022, https://hdl.handle.net/2077/72730.

[32] K.-J. Stol and B. Fitzgerald, "The abc of software engineering research," *ACM Trans. Softw. Eng. Methodol.*, vol. 27, no. 3, Sep. 2018, ISSN: 1049-331X. DOI: 10.1145/3241743. [Online]. Available: https://doi.org/10.1145/3241743.

[33] GitHub, GitHub. [Online]. Available: https://github.com/ (Accessed: 22/04/2024).

[34] ReactiveX, RxJava: Reactive Extensions for the JVM, *GitHub*. [Online]. Available: https://github.com/ReactiveX/RxJava (Accessed: 26/03/2024).

[35] Mockito, Mockito, *GitHub*. [Online]. Available: https://github.com/mockito/mockito (Accessed: 26/03/2024).

[36] SonarSource, sonarqube, *GitHub*. [Online]. Available: https://github.com/SonarSource/sonarqube (Accessed: 19/04/2024).

[37] T. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, 1976. DOI: 10.1109/TSE.1976.233837.

[38] S. Lin, "Rank aggregation methods," *WIREs Computational Statistics*, vol. 2, no. 5, pp. 555–570, 2010. DOI: 10.1002/wics.111. [Online]. Available: https://wires.onlinelibrary.wiley.com/doi/abs/10.1002/wics.111.

[39] S. Salek and M. Åkvist, Benchmark Heuristics, *GitHub*. [Online]. Available: https://github.com/samsalmag/benchmark-heuristics (Accessed: 25/03/2024).

[40] S. Salek and M. Åkvist, Benchmark Heuristics Parser, *GitHub*. [Online]. Available: https://github.com/samsalmag/benchmark-heuristics-parser (Accessed: 02/04/2024).

[41] D. van Bruggen, JavaParser. [Online]. Available: https://javaparser.org/ (Accessed: 21/03/2024).

[42] Google, gson, *GitHub*. [Online]. Available: https://github.com/google/gson (Accessed: 02/04/2024).

[43] scikit-learn, scikit-learn. [Online]. Available: https://scikit-learn.org/stable/ (Accessed: 10/05/2024).

[44] JetBrains, IntelliJ IDEA. [Online]. Available: https://www.jetbrains.com/idea/?var=1 (Accessed: 27/03/2024).

[45] Matplotlib, Matplotlib. [Online]. Available: https://matplotlib.org/ (Accessed: 02/04/2024).

[46] pandas, pandas. [Online]. Available: https://pandas.pydata.org/ (Accessed: 02/04/2024).

[47] NumPy, NumPy. [Online]. Available: https://numpy.org/ (Accessed: 02/04/2024).

[48] V. S. Code, Visual Studio Code. [Online]. Available: https://code.visualstudio.com/ (Accessed: 27/03/2024).

[49] Google, Google Cloud Platform (GCP). [Online]. Available: https://cloud.google.com/ (Accessed: 21/03/2024).

[50] S. Salek and M. Åkvist, Data for the thesis "Exploring Heuristics for Predicting Microbenchmark Stability and Code Coverage using Static Code Analysis", *Zenodo*. [Online]. Available: https://zenodo.org/records/11504043 (Accessed: 06/06/2024).

[51] S. Boslaugh, *Statistics in a Nutshell, 2nd Edition*. O'Reilly Media, Incorporated, 2012, ISBN: 9781449361129. [Online]. Available: https://books.google.se/books?id=s1llAQAACAAJ.

[52] S. Bonovas and D. Piovani, "Simpson's paradox in clinical research: A cautionary tale," *Journal of Clinical Medicine*, vol. 12, no. 4, 2023. DOI: 10.3390/jcm12041633.

[53] scikit-learn. (2024), [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.inspection.permutation_importance.html.

[54] java-decompiler, JD-GUI, *GitHub*. [Online]. Available: https://github.com/java-decompiler/jd-gui (Accessed: 13/05/2024).

[55] Vineflower, Vineflower, *GitHub*. [Online]. Available: https://github.com/Vineflower/vineflower (Accessed: 13/05/2024).