Bachelor Thesis

August 17, 2020

Identifying flaky tests by classifiers

A performance analysis of various machine learning models

Christian Birchler

of Lucerne, Switzerland (15-924-160)

supervised by

Prof. Dr. Harald C. Gall
Giovanni Grano
Christoph Laaber

University of Zurich[UZH]

s.e.a.l.
software evolution & architecture lab

Bachelor Thesis

Identifying flaky tests by classifiers

A performance analysis of various machine learning models

Christian Birchler

University of Zurich[UZH]

s.e.a.l.
software evolution & architecture lab

Bachelor Thesis

Author: Christian Birchler, christian.birchler2@uzh.ch

Project period: 17. February 2020 - 17. August 2020

Software Evolution & Architecture Lab
Department of Informatics, University of Zurich

Acknowledgements

First of all I would like to thank Giovanni Grano and Christoph Laaber for supervising me during my bachelor's thesis. You gave me valuable feedback on a weekly basis although we had to do the meetings virtually due to the COVID-19 pandemic.

Furthermore, I would like to thank Professor Gall for allowing me to do the thesis at his research group and to use the infrastructure on ScienceCloud.

I want also to thank my sister Alexandra Birchler for proofreading the thesis especially to improve the grammar and my writing. Last but not least I want to thank my friend Svenja Keller for giving motivation and insights of her master thesis.

Abstract

Testing is a crucial part in software development. Most of the current bigger software projects integrate the testing in a continuous integration (CI) pipeline. A failing test will prevent the deployment of the software. In case of flaky tests where tests may fail and pass non-deterministically without a change to the code or the environment is an issue to deal with, since the developer is probably spending time to find a bug although the code under test is not defect. Previous studies focused mainly on the root causes of flakiness but only a few research was done on how to mitigate flaky tests. In this thesis we investigated the impact of different memory related JVM metrics on the predictability of flaky tests. For this purpose we took JVM metrics of 82 open-source Maven projects which had already recorded flaky tests. In order to take the measurements a toolchain script was developed, that injected the necessary code in to the test code so that JVM metrics could be collected during test executions. The toolchain ran for each project the test suites ten times to identify flaky tests that have different outcomes. The toolchain ran on different machines with different RAM sizes to see if there is a difference in the data. We did a PCA and a biplot to identify cluster structure in a lower dimensional space and applied various parametric and non-parametric classification models on the data. The results show that flaky tests are to a certain degree predictable by JVM metrics and the RAM size has also an impact on the predictability of flakiness. This insights allows to develop new tools to handle flaky tests and motivate more research.

Zusammenfassung

Testen ist ein entscheidender Bestandteil der Softwareentwicklung. Die meisten der derzeit größeren Softwareprojekte integrieren die Tests in eine CI-Pipeline (Continuous Integration). Ein fehlgeschlagener Test verhindert die Bereitstellung der Software. Bei Flaky-Tests, bei denen Tests fehlschlagen und nicht deterministisch ohne Änderung des Codes oder der Umgebung bestehen können, ist dies ein Problem, da der Entwickler wahrscheinlich Zeit damit verbringt, einen Fehler zu finden, obwohl der zu testende Code nicht fehlerhaft ist. Frühere Studien konzentrierten sich hauptsächlich auf die Ursachen von Flaky-Tests, es wurden jedoch nur wenige Untersuchungen durchgeführt, um Flaky-Tests zu erkennen. In dieser Arbeit untersuchten wir den Einfluss verschiedener speicherbezogener JVM-Metriken auf die Vorhersagbarkeit von Flaky-Tests. Zu diesem Zweck haben wir JVM-Metriken von 82 Open-Source-Maven-Projekten verwendet, die bereits Flaky-Tests aufgezeichnet hatten. Um die Messungen durchzuführen, wurde ein Toolchain-Skript entwickelt, das den erforderlichen Mess-Code in den Testcode einfügt, damit JVM-Metriken während der Testausführung erfasst werden können. Die Toolchain wurde für jedes Projekt zehn Mal ausgeführt, um Flaky-Tests mit unterschiedlichen Ergebnissen zu identifizieren. Die Toolchain wurde auf verschiedenen Computern mit unterschiedlichen RAM-Größen ausgeführt, um festzustellen, ob sich die Daten unterscheiden. Wir haben eine PCA und einen Biplot durchgeführt, um die Clusterstruktur in einem Raum mit niedrigeren Dimensionen zu identifizieren, und verschiedene parametrische und nicht parametrische Klassifizierungsmodelle auf die Daten angewendet. Die Ergebnisse zeigen, dass Flaky-Tests bis zu einem gewissen Grad durch JVM-Metriken vorhersagbar sind und die RAM-Größe auch einen Einfluss auf die Vorhersagbarkeit von Flaky-Tests hat. Diese Erkenntnisse ermöglichen es, neue Werkzeuge zu entwickeln, um Flaky-Tests zu handhaben und um für mehr Forschung von Flaky-Tests zu motivieren.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Software testing is nowadays automated and embedded in a *continous integration/delivery* (CI/CD) pipeline. Such automation makes it easier to bring the latest code changes into production. The production build usually relies on a passing test suite, which is a part of a CI/CD pipeline. Any unwanted interruptions in terms of failing tests will lead to a postponed deployment of the software and will produce more costs. Usually a failing test will indicate that in a certain module of the software is a defect, which must be resolved by the developers. But in case of so called *flaky tests* a developer usually does not know if the defect lies in the test code itself or in the code under test [2]. Flaky tests are tests that have a non-deterministic behavior. Such tests pass and fail on different runs without any code changes.

The first intense study on flaky tests were done by *Luo et al.* [3]. They had the main goal to understand the root causes of flaky tests and presented a categorization of various causes of flakiness. In total they identfied 10 root causes (see table 2.1). In addition to the first study *Eck et al.* [4] continued studying flaky tests but from a developer's perspective. They asked 21 professional developers to classify 200 flaky tests they previously fixed. One of the main insights was that they identified four more root causes for flakiness next to the root causes identified by *Luo et. al*. There is also research on flaky tests for platform specific application like *Android*. *Thorve et al.* [5] focused their study on mobile applications for Android, which also identified new root causes for flakiness. Next to the studies on understanding the causes of flakiness there are also studies on tools for handling flaky tests like *DeFlaker* from *Bell et al.* [6].

The current situation in dealing with flaky tests is not as evolved as automated test case generation for example. The main method to mitigate flaky tests is to rerun the tests when they fail. This approach requires more computational resources and is not optimized like *DeFlaker*, which rather keeps track of code changes to identify flaky tests than to do multiple reruns. But for all approaches like *Rerun* and *DeFlaker* there is always a failing test required to identify a flaky test. So far there are no tools to give measures about flakiness of an initial passing test.

The discussion of flaky tests not only arrived in the academic but also in the practical field. *Listfield* [7] published his insight of flaky tests that correlates with memory consumption on the *Google Testing Blog*. The main insights are that the higher the memory size and the RAM usage for a test are the higher is the probability that the respective test is flaky. The blog post does not give much information on what kind of test suites are used and it is not clear what kind of memory is used especially if the tests run in a *Java Virtual Machine* (JVM) that have different memory segments and other memory related metrics. There is no other research present, which investigated the relationship between memory related metrics and flakiness of test cases.

For this thesis we developed a toolchain script that collected JVM metrics. The toolchain ran for each project the according test suite ten times. Before running the test suites, the test code was modified in order to be able to collect JVM metrics. The toolchain ran on two machines with different memory sizes. The dependent variable of the data is the flakiness. The flakiness was determined in two different ways. Once by labeling the data according to the own identified flaky tests. And the other one by taking the labels from another source. For the different labeled data sets various classifiers were trained and evaluated. The evaluation showed the predictability of flaky tests by JVM metrics is dependent on memory and the choice of the classifier.

Since the blog post of *Listfield* [7] gives the main motivation to do further research it is worth to answer the following research questions:

RQ1 Which classifier perform best to predict flaky tests based on JVM metrics?

RQ2 Is there a difference between test case related JVM metrics versus the general state of the JVM for predicting flaky tests?

RQ3 Do machines with different RAM sizes impact the flakiness?

RQ4 How are the prediction performances with different sources for the labels?

The results of this thesis can be used as another motivation for further research on flaky tests that focuses on memory consumption. Furthermore, an optimized machine learning model could be integrated into a tool that gives information if a test is flaky or not. Such a tool might lower unnecessary costs in dealing with flaky tests. For that reason further research on flaky tests is desirable.

The thesis is structured as follows: In chapter 2 we will discuss in more detail the previous research. An important role plays the very first studies on flaky tests in section 2.1 that examine the root causes of flakiness. In section 2.2 we also discuss the available tools to handle flaky tests. In the methodology chapter 3 we first take a look on the data set (section 3.1) with known flaky tests. This data set is used in the toolchain section 3.2 as a basis of open-source projects. Furthermore, this section will explain the needed code manipulation and test execution environment. Then section 3.3 will explain the need and use of a feature selection technique for doing the analysis. The various classifiers are conceptually explained in section 3.4.

Chapter 2

Related Work

The research on flaky tests is an ongoing process. The history on research in this field is young, since the first study was presented in 2014. The following sections give an overview of the research and their insights.

2.1 Categorization

The first extensive study on flaky tests was published back in 2014 by *Luo et al.* [3], that categorizes the root causes of flaky tests [3]. The motivation of this study came from regression testing, which checks that code changes do not break any existing functionality. Regression testing assumes that the tests have a deterministic behavior, which is not the case of flaky tests. To gain a better understanding of flaky tests and how to fix them, 201 commits, which refer to fixes of flaky tests of 51 open-source projects are analyzed. Those commits are selected by searching through the commit logs of the *Apache Subversion* (SVN) [8] repository of the *Apache Software Foundation* [9]. The search is based on keywords which were (1) "flak" and (2) "intermit". This search by keywords resulted in 1'129 commit messages, which are inspected manually to ensure that they fix indeed a flaky test. After the inspection 855 commits are like to be about to fix flaky tests and the other 274 commits are either only about the CUT or other code modules but not about the test code. 486 commits of these 855 are likely to be distinct fixed flaky tests, which were labeled as *LDFFT*. The other 369 commits are duplicates, which mention already identified flaky tests or they do not properly fix the test. For further analysis they sampled 201 commits which were labeled as *LDFFT* to identify different categories of flaky tests. They defined 10 categories of root causes, which are explained in the following table 2.1.

To pursue the primary research insights on the root causes of flaky tests *Eck et al.* [4] did a study on the developer's perspective on dealing with flaky tests. The goal was to see the developers' perception on the causes and fixing strategies of flaky tests [4]. They asked 21 developers from Mozilla to categorize 200 flaky tests from their own data base according to their nature, origin and fixing efforts. 31% of the categorized tests do not belong to a category defined by *Luo et al.* [3] and as such require a new definition of categories. In total they identified four new root causes of flakiness, which are listed in the table 2.2.

All the prior studies did not focus on mobile apps. *Thorve et al.* [5] did an empirical study on flaky tests in *Android* apps. The assumption was that there could be more root causes for flakiness specifically for *Android* apps. *Android* apps make use of many third-party software and libraries, which might have an impact on the test flakiness. They used a similar approach to collect data as *Luo et al.* [3], which is based on commit messages of open-source projects. They searched on

Category	Explanation
Async Wait	Test makes an asynchronous call and does not properly wait for the result
Conccurrency	Different threads interact in a non-desirable manner so that the result is affected (e.g., data races or deadlocks)
Test Order Dependency	Flaky tests belong to this category if its results depend on the order of execution. Ideally, each tests should be isolated.
Resource Leak	A resource leak occurs when the application doe not manege correctly its resources (e.g., memory, connection, etc.)
Network	A test belongs to this category if its execution depends on the internet connection which cannot be controlled by the developer
Time	If a test relies on the system time then the test outcome can have a non-deterministic behavior
I/O	If a file is not properly closed (e.g., a file reader is not garbage collected) then the test can have an undesirable result
Randomness	A flaky test that depends on random number generators
Floating Point Operations	Simple floating point operations can lead to different results (e.g., non-associative addition)
Unordered Collections	If the test assumes that iterations over unordered collections give the elements in a particular order then the test can be flaky

Table 2.1: Overview of root causes of flaky tests by *Luo et al.* [3]

GitHub for *Android* projects and selected commits which contain the keywords (1) "flaky", (2) "intermittent", (3) "async" or (4) "unstable". After collecting those commits they were manually inspected to ensure that they relate to flakiness issues and the code differences belong to a flaky test. After the filtering they had a data set of 77 commits from 29 *Android* projects. The analysis showed that the flaky tests could be assigned to 6 categories. Furthermore, the study identified three additional root causes for flakiness which are *Dependency*, *Program Logic*, and *UI*. These new categories are summarized in table 2.3. The commits are assigned as following: (1) 36% belong to *Concurrency*, (2) 22% belong to *Dependency*, (3) 12% belong to *Program Logic*, (4) 8% belong to *Network*, (5) 8% belong to *UI*, and (6) 11 commits could not be assigned to any known category (*Hard to Classify*).

2.2 Tools

Besides the research on root causes of flaky tests there are also implemented tools to deal with flaky tests in an automated manner. The first tool we look at is *DeFlaker* developed by *Bell et al.* [6]. After this we consider the tool *iDFlakies* which was developed by *Lam et al.* [1].

Category	Explanation
Too Restrictive Range	Some valid output values are outside the assertion range condidered at test design time, so the test fails when they show up
Test Case Timeout	The size of a test case is growing over time without adjusting the max runtime value. This category is very similar to Test Suite Timeout.
Platform Dependency	Test outcome varies across different platforms (e.g., 32-Bit-, 64-Bit system, debug build, etc.)
Test Suite Timeout	Test suites grow over time and the max runtime value is not always adjusted accordingly. This is not a particular test case flaky issue.

Table 2.2: Overview of additional root causes of flaky tests by *Eck et al.* [4]

Category	Explanation
Dependency	Flakiness due to certain hardware, Android OS version, or a third-party library
Program Logic	Flakinessdue due to wrong assumptions on the apps' program behaviors
UI	Flakiness due to poor widget designs or bad rendering

Table 2.3: Overview of additional root causes of flaky tests for Android apps by *Thorve et al.* [5]

2.2.1 DeFlaker

To overcome the widely used *Rerun* technique, which leads to performance overhead *DeFlaker* uses a different approach to identify flaky tests. *DeFlaker* is able to declare a failing test as flaky immediately after its execution without to rerun it multiple times. The tool keeps track of the changes of the *Code Under Test* (CUT) by the *Version-Control System* (VCS). If the test outcome changes without any changes of the according CUT then the test is flaky. Furthermore, the tool keeps track of statement coverage for the CUT and the test code. If the coverage shows that there were no lines covered of a newer version of the code and the test outcomes changes then test is also identified as flaky. *Bell et al.* evaluated the performance in terms of detecting flaky tests of different *Rerun* methods and also the performance of their developed tool *DeFlaker*. They conducted three different *Rerun methods*: (1) The rerun approach of the *Maven Surefire* test runner [10], which reruns the tests in the same *Java Virtual Machine* (JVM), (2) for each rerun a fork of the JVM is done, and (3) reboot for each rerun the machine. For the study 5'966 builds of 26 open-source projects are used to evaluate the performance to identify flaky tests for each of the three rerun methods. The results show that 1'162 tests were identified as flaky by method (1), 4'186 by methods (1) or (2), and 5'075 tests by methods (1), (2) or (3). This results are interesting, since the implemented rerun method by the *Maven Surefire* plugin detected only 23% of all known flaky tests. On the other hand the *DeFlaker* detected 95.5% of the known flaky tests with a very low false positive rate of 1.5%.

DeFlaker proposes an improved way to detect flaky tests. Despite of the significant run-time and detection improvements the tool works as with the *Rerun* approaches only on failing tests. This approach does not take into account of having flaky tests, which pass in most cases.

2.2.2 iDFlakies

The framework *iDFlakies* developed by *Lam et al.* [1] is able to classify order-dependent and non-order-dependent flaky tests of a test suite. *Bell et al.* also created a data set of flaky tests of various open-source *Maven* projects which is used as a data base for this thesis. In order to collect the data the framework runs a test suite multiple times but with different test orders. The framework provides its own test runner for *JUnit* tests, which can be integrated into any *Maven* project as a plugin.

Their study uses three different sources of *Maven* projects: (1) 44 projects from *Bell et al.* [6] and *Palomba et al.* [11], (2) 150 most popular Java projects from *GitHub* [12] up to October 2018, and (3) 500 most popular Java projects from *GitHub*, which were updated in November 2018. The *iDFlakies* framework detected 422 flaky tests from 111 moduls in 82 projects. 50.5% of these flaky tests are classified as *order-dependent* and 49.5% as *non-order-dependent*. Furthermore, they showed that the probability of having a flaky test in a single round varies a lot. The probability for some rounds are lower than 1% and for others over 50%. The *random-class-method* approach to reorder the tests in the test suite detects the most flaky tests [1].

The study from *Lam et al.* [1] is highly motivated through the assumption that the *Test Order Dependency* category defined by *Luo et al.* [3] is one of the most frequent root cause of flakiness [3], [11]. As shown in the evaluation above, 50.5% of the detected flaky tests are due to the *Test Order Dependency*. To overcome the gap of 49.5%, which are considered as *non-order-dependent* it is desirable to have a tool, which is able to classify the rest into the categories defined by *Luo et al.* [3] and *Palomba et al.* [11]. This might be a challenge to implement a programmatic approach that is able to classify also the other root causes.

An important aspect from this framework is that it runs the tests in a *depth-first* manner. This means that the tool runs the defined number of rounds module wise. The comparison with a *breadth-first* test execution is still an missing part which *Lam et al.* are going to do [1]. In contrast, this bachelor thesis presents also a tool which reruns the test suits but in a *breadth-first* manner. Furthermore, it is not clear from the paper if the different reruns of the tests are done in a new JVM whereas this thesis spawns always a new JVM for each test suite execution.

Chapter 3

Methodology

In this thesis we want to investigate the impact of different JVM metrics on the flakiness of test cases in a test suite. The main goal is to do a binary classification of test cases into *flaky* and *non-flaky*. We will automate the data gathering across popular *Maven* projects. Furthermore, we will analyze data by dimensionality reduction and different machine learning models.

The following sections explain the methodology used in this research. We start with the choice of the data set then we take a look on how the metrics gathering worked with a toolchain. At the end we look on the analysis part and which classifiers we used.

3.1 Data set

To collect metrics of popular *Maven* project it is useful to know which projects and which versions have a fair amount of well known flaky tests. We decided to use the provided data set of the *iDFlakies* study by *Lam et al.* [1]. They developed a framework to detect flaky tests and to categorize them into *order-dependent* and *non-order-dependent*. They identfied many new flaky tests of various open-source software projects. These new identfied flaky tests are listed in a data set which is online available [13]. The data set has the following attributes:

- *URL*: The url to the Git repository
- *SSH*: The commit hash of the Git version control system
- *Test Count*: Number of tests in the module
- *Module Name*: Name of the module which the flaky test belongs to
- *Test Name*: The full package name with the class and test name
- *Category*: The test is either *order-dependent* or *non-order-dependent*
- *Version*: Source of the data

The *iDFlakies* data set has in total 422 entries of known flaky tests over 82 open-source projects. The table 3.1 shows an overview of the projects with the most known flaky test. 211 flaky test cases are categorized as *order-dependent*, 18 test cases are simpli skiped and 193 test cases are *non-order-dependent*.

Project	# Flaky tests
hadoop	68
Java-Websocket	52
wildfly	44
http-request	28
incubator-dubbo	24
Activiti	20
fastjson	15
elastic-job-lite	13
retrofit	9

Table 3.1: Overview of 9 projects with the most flaky tests identfied by *Lam et al.* [1]

3.2 Toolchain

In order to collect the metrics of various open-source projects and to save them into a single *csv* file it was useful to develop a toolchain python script, which automated this process. In figure 3.1 is shown the high-level flow diagram of the toolchain script. First the toolchain reads the data set of the *iDFlakies* project [1]. Then the script dowonloads the the project of the corresponding URL. For doing the measurements code needs to be injected in the test suite so that metrics of the JVM can be gathered and written to a csv file. More details about the code injection tooling can be found in section 3.2.1. After the code injection is completed the test suite is run on a *ScienceCloud* [14] instance (more details in section 3.2.2). The whole test suite is run 10 times. If all iterations are completed then the toolchain looks if there is another version of the same project, which has a recording in the *iDFlakies* data set. If so then the new version is checked out and the test suite is run again else the toolchain proceeds with the next project. The following sections give more insights about the implementation of the different states in the toolchain.

3.2.1 Code injection

An important aspect of this toolchain is the process of injecting the code for taking the actual measurements of the JVM. A first approach was to use a Java agent, which allows to change the byte code of the classes during run-time. Java agents are interesting to change code because it is possible to take an already compiled project and instrument it without re-compiling from source. A Java agent can be attached to a JVM at start so that before the execution of the main method of the application a so called *premain* method is called. This *premain* method allows as mentioned above to add an instrumentation tool by using the provided Java *Instrumentation* API. Java agents and the *Instrumentation* API are not new features in the Java ecosystem since they are already specified in Java 5 [15]. The concrete bytecode manipulation could be done with *ASM* developed by *Bruneton et al.* [16], which is a Java library for manipulating Java class files. This library is low level and it requires more time to implement an agent than using the *Bytebuddy* [17] library. *Bytebuddy* depends only on the *ASM* library. Therefore, *Bytebuddy* can be viewed as an wrapper of *ASM*, which also provides some interfaces that allow to develop in a consistent way. Although the approach with a Java agent to incect code to collect metrics of a test case seems to be good because there is no need to change the source code it has some difficulties in the environment of the toolchain. For developing a toolchain which runs several test suites of different Maven projects it is necessary to assume some convention that the projects follow. Since the projects are all Maven projects they must have a *pom.xml* where the agent can be attached via the *Surefire* plugin configuration. The problem is that most projects are multi module projects with multiple *pom.xml* files

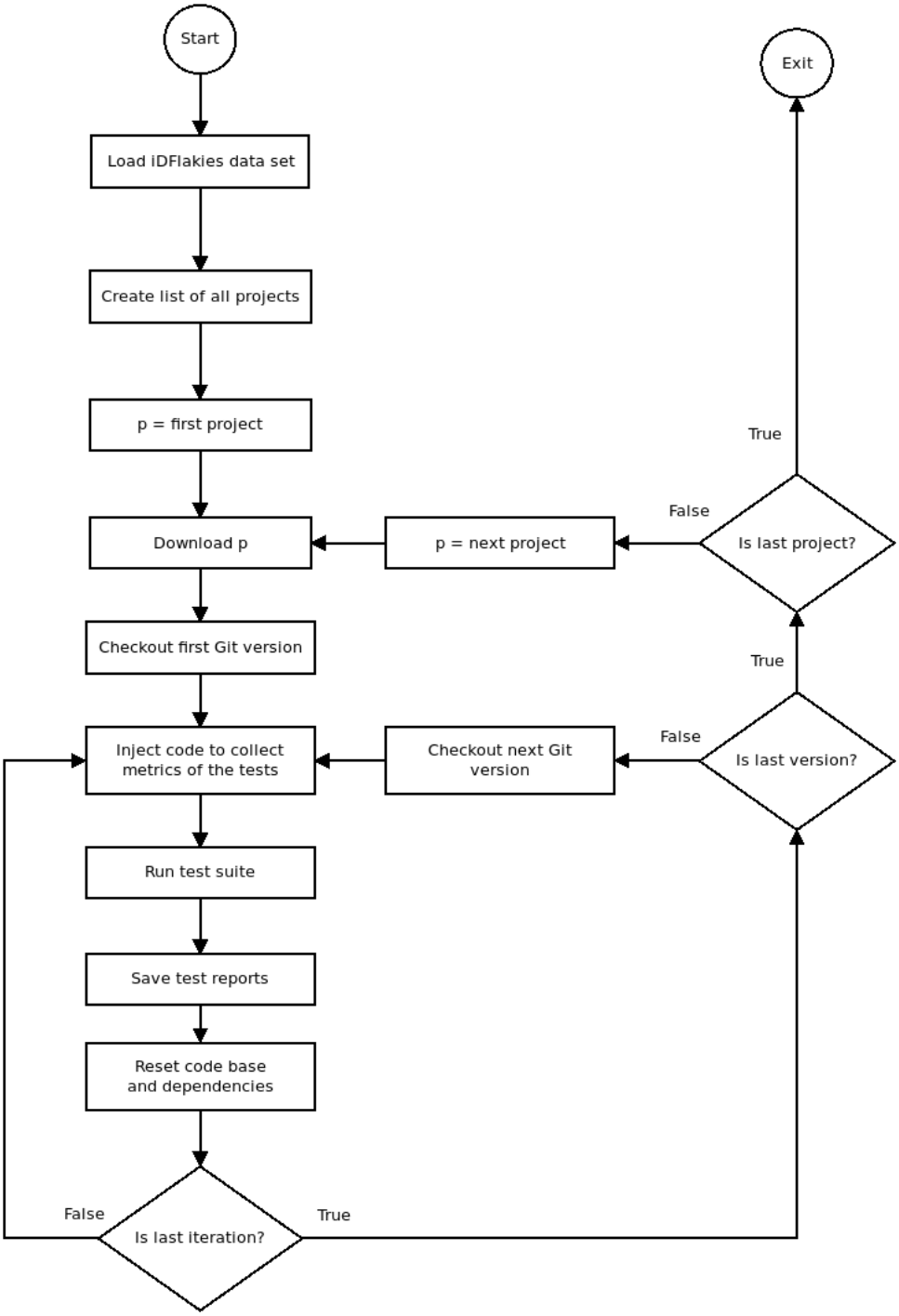

Figure 3.1: High-level flow diagram of the toolchain.

and with multiple Maven profile configurations. So each *pom.xml* of all projects need to be adjusted individually. However, this fact make it difficult to automate that very process. Therefore, an other approach to inject the code for taking the measurements was taken.

Since the approach with a Java agent has some problems in terms of attaching the agent at the JVM in an automated manner via the *pom.xml* files an other solution should do this. Another assumption besides of having a *pom.xml* file is the fact that Maven projects follow always a certain folder structure. The folder structure separates the application code from the test code. So it is possible to manipulate the test code on the source code level. The drawback is that it is not possible to attach the code manipulation dynamically during run-time like an Java agent. Furthermore, we have also to take into account that the projects need to be re-compiled, which also require time and therefore slow down the toolchain. The stage of injecting code into the source files of the tests is done in a separate Java process since the injection is implemented in Java with *JavaParser* [18] whereas the toolcahin script is written in Python.

The toolchain invokes the source code injector by running its fat jar (jar with all dependencies included) with the appropriate arguments. The following listing 3.1 shows how to start the code injector.

```
java -jar /path/to/jar /path/to/project projectName commitHash currentIteration
```

Listing 3.1: Call source code injector

Besides the path to the source code injector fat jar there are also 4 more arguments. These arguments are needed to give additional information to the measurements which are also written to the *csv* file. The following table 3.2 explains the arguments.

Argument	Explanation
/path/to/jar	Path to the fat jar of the code injector
/path/to/project	Path to the target Maven project to instrument
projectName	Name of target project (needed for *csv* file)
commitHash	Version identifier of the target project
currentIteration	Current run of the test suite

Table 3.2: Explanation of the arguments for the code injector

The injected code uses a library from *Dropwizard Metrics* [19] to take measurements of the JVM. The tables 6.1 and 6.2 show an overview of all metrics that are memory related. The table 6.4 summarizes the metrics of threads in the JVM. Furthermore, the table 6.3 shows the metrics that are related to the JVM garbage collector.

3.2.2 ScienceCloud

The execution environments of the toolchain were *ScienceCloud* instances of the University of Zurich [14]. To run the toolchain on a local machine was not feasible since the run-time took about 70 hours. To overcome this issue the toolchain was run remotely. The first runs were done with a virtual machine that has 1GB of RAM and 1 virtual CPU. This configuration was chosen since it is the smallest that was available on *ScienceCloud* to save computing resources. After looking the results and especially looking at the dump files it turned out that 1GB of RAM was not enough. Therefore two more runs are started with different configurations. One machine had

16 GB of RAM and 4 virtual CPU. The second one had 128 GB of RAM with 16 virtual CPUs. Later we call these machines machine A and machine B accordingly. All virtual machines had a Ubuntu 18.04 operating system running.

3.3 Feature selection

The post processed data set contains 72 variables in total. For doing further analysis on this data and especially to train classifiers on this data we want to avoid overfitting. overfitting occurs when too much data points is used for training. This leads to models, which do not generalize the data well. It is likely that couple of the variables are redundant and or irrelevant to do a binary classification whether test is flaky or not. Many machine learning algorithms are costly in terms of fitting the models. This is clearly true because with more features we also have more data, which needs to be processed of the computer. The field of machine learning algorithms and feature selection is broad and it is not the objective of this thesis. However, we need a method to overcome the problem of overfitting and feature irrelevance. *Ghotra et al.* [20] published a large-scale study on the impact of different feature selection techniques. Their objective was to identify a selection technique that outperforms best among the most known machine learning classifiers. They figured out that a correlation-based filter-subset technique with the *BestFirst* search algorithm performs in general best. Therefore, for this thesis we used this method to do a feature selection.

The correlation based filter subset technique is composed of two parts. For the first part a heuristic function is required and for the second part a search algorithm is needed. With these two components we identify generally the best features for the most classifiers. *Hall et al.* [21] defined a heuristic function that is correlation based. This function considers a whole group of features rather only single features. The heuristic function checks correlation of the features with the class label and the inter-correlation among the features. Therefore, it is also called a filter-subset technique. The following equation 3.1 describes the heuristic function for a subset of features [21].

$$G_s = \frac{k * \overline{r_{ci}}}{\sqrt{k + k * (k-1) * \overline{r_{ii'}}}}, \tag{3.1}$$

where k is the cardinality of the feature subset G_s, and $\overline{r_{ci}}$ is the average Pearson's correlation score between the class label and the features. The denominator considers the average Pearson's correlation score $\overline{r_{ii'}}$ among the features themselves. From this equation it s clear that correlation among the features should be low to get a higher heuristic score.

For the second part, *Ghotra et al.* [20] showed in their study got the correlation-based feature subset selection technique should be combined with the *BestFirst* search algorithm. The best first search method is a graph algorithm. The nodes of the graph represent a feature subset which has a heuristic value. The heuristic value is obtained from the formula above. The aim of the best first search algorithm is to walk through the graph and search for the node that has the best heuristic value.

3.4 Classifiers

In this thesis our aim is to do a binary classification whether a certain test is flaky or not based on the runtime metrics we have collected. For this reason we evaluate 8 classifiers on their prediction performances. We have chosen four parametric classifiers and four non-parametric classifiers. first we will take a look at the parametric classifiers and discuss their properties and how they work. After this, we will discuss the non-parametric classifiers and outline their properties.

3.4.1 Parametric

The first parametric classifier we take a look at is the **Linear Discriminant Analysis** classifier or for short LDA. The LDA looks at the variances of the different groups (in our case only two) and define a classification line separate the groups. In the one-dimensional case we can illustrate this separation by throwing the classification line where the distributions of the groups intersect. This method can also be applied on more than one variable.

In case where the variances of the groups differ then the LDA might be insufficient. The **Quadratic Discriminant Analysis** (QDA) classifier Could be a better solution. Instead of calculating the intersection of the densities the QDA takes the intersection of the log-densities.

The next method is the **Naive Bayes** classifier. This classifier uses the Bayes theorem that describes a probability of an event. This probability is based on a prior knowledge which effects the probability of occurring a certain event.

The last parametric classifier we look at is the **Logistic Regression**. The purpose of the logistic regression is to model the probabilities occurring an event based on a set of features. The probabilities are linked with a link function to an ordinary linear regression model. The inverse of this link function is the logistic function.

3.4.2 Non-parametric

The first non-parametric classifier we look at is the **Classification Tree** classifier, which is also called as regression tree classifier. This classification method uses the approach of defining a set of rules to split the domain of a feature to classify the observation to a group. At each successive step a feature is selected to discriminate between one group and the other. In the end we have a set of rules where to split the domains of the variables to obtain a classification model.

A more sophisticated approach, which also uses the idea of classification trees is the **Random Forest** classifier. The main idea behind this classifier is to grow multiple classification trees, which build a forest. The classification trees in the forest are not the same and can therefore lead to a different prediction. The random forest classifiers predict the outcome which has the most votes.

Adaptive Boosting (AdaBoost) belongs to the boosting algorithms that apply the same model multiple times but in each iteration the model is optimized based on the previous outcome.

Extreme Gradient Boosting (XGBoost) [22] is an optimized implementation of gradient boosting. Gradient boosting is in general used for classical regression and classification problems.

3.5 Cross-validation

For the evaluation of the different classifiers a special kind of cross-validation technique is used namely the *stratified 10-fold cross-validation*. This approach differs from the classic k-fold cross-validation in the folding of the data. In our case we have *a priori* knowledge of having an imbalanced data set where the number of flaky tests is much lower than the non-flaky tests. This assumption verifies also *Micco* [23] who showed that Google has 1.5% flaky tests in their test suites. So flaky tests are rare events and must be treated carefully. The *stratified 10-fold cross-validation* is a way to tackle the problem of imbalanced data in machine learning. This type of cross-validation ensures that the distribution of the class labels are as similar as possible over the different folds [24]. It is important that there is no fold with only one class label. If there is only one class represented in a fold then the performance scores will calculate no reliable values.

3.6 Threats to validity

The methodology above has also some threats to validity. The whole thesis is limited to projects in the Java ecosystem. In particular all the projects are open-source projects that use the Maven build tool. The according pom files of the projects follow not always the same structure which can affect the source code injector. Furthermore, the used projects are from the *iDFlakies* data set. There is no other large public data set on known flaky tests. This data set is limited to only 82 projects. For the classification models the default parameters of the *sklearn* library is used. The defaults are usually not the best and with parameter optimization a significant better prediction performance is possible. Some projects may have very specific requirements to run the tests but in order to automate the test execution of 82 projects it is likely that some specific requirements were not met. For all projects the same JDK vendor was used and the same Java 8 version. The test execution was also done in a virtual machine on *ScienceCloud*. A virtual machine on a server may impact the execution of the toolchain and lead to a different result than the execution of the toolchain on a personal computer.

Chapter 4

Results

In the following sections you will find the main results of the data analysis. To get a first impression of the data and to see how the structure of the data of flaky tests are we did a PCA and a biplot to reduce the dimensionality. The second part you examines the tables that summarize the cross-validation results of different classifiers. Those tables shall answer the research questions.

4.1 PCA

To identify a cluster structure among multivariate data a PCA with an according biplot of the *hadoop* project on machine A was done. The *hadoop* project has the most identified flaky tests. In figure 4.1 you see the biplot of the data of the *hadoop* project on machine A. By a dimension reduction a cluster structure is identified but not completely separated from the non-flaky data points.

Other software projects have less identified flaky tests than the *hadoop* project. Biplots for the other projects would not give a valuable insight since the data for those projects are too imbalanced.

4.2 Cross-validation

The following sections show the results of various stratified 10-fold cross-validations. Furthermore, the sections are organized so that for each machine the cross-validation is done based on the JVM metrics and on the metrics differences. The differences are taken to do the validation on data that are only test case related and not on the general state of the JVM.

The following abbreviations for the classifiers and its explanations will help to read the tables:

- lda: Linear Discriminant Analysis
- qda: Quadratic Discriminant Analysis
- gnbayes: Gaussian Naive Bayes
- logreg: Logistic Regression
- ctree: Classification Tree
- rforest: Random Forest
- adaboost: Ada Boosting

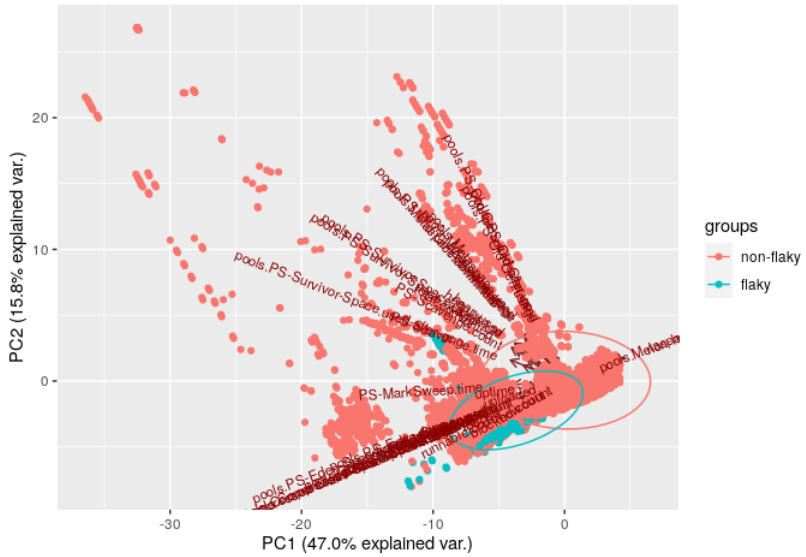

Figure 4.1: PCA biplot of hadoop's flaky tests

- xgboost: Extreme Gradient Boosting

Here is also a list of the abbreviations for the cross-validation scores and its explanations:

- ACC: Accuracy
- PREC: Precision
- RECA: Recall
- F1: F1 score
- MCC: Mathew's Correlation Coefficient
- AUC: Area under the ROC curve

4.2.1 Machine A

In this section you will find the results of machine A which had the following specifications:

- Ubuntu 18.04
- 4 vCPUs
- 16 GB of RAM

Table 4.1 shows the results of the predicting performances of flaky tests that are identified by ourselves as flaky. Whereas table 4.3 shows the results when using the labels from the iDflakies data

set. The tables 4.2 and 4.4 uses the differences of before and after a test execution.

The results of the cross-validation for the data with own identified flaky tests in table 4.1 show that the gnbayes classifier perform worst if we consider the PREC, RECA, F1 and the MCC. The rforest classifier has the best AUC, MCC, F1 and RECA score. The simple classification tree performs almost identical to the random forest classifier. Although the non-parametric classifiers perform generally better than the parametric it is obvious that the adaboost classifier has poor PREC, RECA, F1, MCC and AUC scores.

Classifier	ACC	PREC	RECA	F1	MCC	AUC
lda	0.9872	0.0664	0.4	0.1139	0.1591	0.6942
qda	0.9844	0.0477	0.35	0.0839	0.1247	0.6678
gnbayes	0.9968	0.0	0.0	0.0	-0.0015	0.4994
logreg	0.9979	0.15	0.0008	0.0017	0.0064	0.5004
ctree	0.9987	0.7847	0.5149	0.6211	0.6347	0.7573
rforest	0.9987	0.7745	0.5280	0.6275	0.6387	0.7638
adaboost	0.9977	0.0126	0.0017	0.0030	0.0040	0.5007
xgboost	0.9981	0.9442	0.1175	0.2076	0.3299	0.5587

Table 4.1: Stratified 10-fold cross-validation of own identified flaky tests for machine A.

If we take a look at the results of the cross-validation based on the metrics' differences then the logreg classifier has the poorest performance according to the RECA, F1, MCC and the AUC scores. The qda has a very low accuracy of 19.39% although the data is highly imbalanced. The gnbayes classifier has in contrast to the previous table the larges AUC score but the accuracy is with 89.14% low in comparison to the other classifiers and especially in contrast to the imbalanced data. The ctree and the rforest classifiers hava the highest PREC, F1 and MCC scores.

Classifier	ACC	PREC	RECA	F1	MCC	AUC
lda	0.9942	0.2054	0.5053	0.2913	0.3193	0.7503
qda	0.1939	0.0029	0.9875	0.0057	0.0222	0.5898
gnbayes	0.8914	0.0174	0.8214	0.0340	0.1098	0.8565
logreg	0.9976	0.8	0.0	0.0	-6.2402e-05	0.4999
ctree	0.9976	0.5155	0.2196	0.3067	0.3346	0.6095
rforest	0.9976	0.5198	0.2232	0.3094	0.3377	0.6113
adaboost	0.9974	0.1149	0.0553	0.0682	0.0741	0.5274
xgboost	0.9976	0.5133	0.05	0.0767	0.1216	0.5249

Table 4.2: Stratified 10-fold cross-validation of own identified flaky tests for machine A with metrics differences.

The scores of the cross-validations based on iDFlakies class labes and the absolute JVM metrics are shown in table 4.3. The table shows clearly that the parametric classifiers perform worst according the the PREC, RECA, F1, MCC and AUC. The rforest has the highest overall scores (except the PREC score) followed by the ctree classifier. Note that the xgboost has the highest PREC score but in general it performs worse than the rforest and the ctree classifier. In comparison to the results

of the table 4.1 the rforest and the ctree classifiers have similar scores. The scores of the data with the class labels from iDFlakies are not significantly lower.

Classifier	ACC	PREC	RECA	F1	MCC	AUC
lda	0.9891	0.0	0.0	0.0	-0.0024	0.4996
qda	0.9818	0.0177	0.0131	0.0149	0.0061	0.5025
gnbayes	0.9897	1.0	0.0	0.0	0.0	0.5
logreg	0.9897	0.1	0.0	0.0	-0.0006	0.4999
ctree	0.9932	0.7795	0.4702	0.5865	0.6023	0.7344
rforest	0.9931	0.7701	0.4763	0.5885	0.6026	0.7374
adaboost	0.9899	0.7523	0.0195	0.0378	0.1098	0.5097
xgboost	0.9908	0.9300	0.1195	0.2117	0.3315	0.5597

Table 4.3: Stratified 10-fold cross-validation of flaky tests identified by iDFlakies for machine A.

The table 4.4 shows the results of the data with the metrics' differences and the class labels from iDFlakies. We see that the rforest and the ctree classifiers have the highest MCC scores but in comparison to the previous results they are low. Overall, all the scores for all classifiers are low. Like the previous results also here are the rforest and the ctree classifiers similar in their performances.

Classifier	ACC	PREC	RECA	F1	MCC	AUC
lda	0.9882	0.0769	0.0172	0.0281	0.0318	0.5075
qda	0.9561	0.0563	0.2189	0.0896	0.0938	0.5912
gnbayes	0.9596	0.0526	0.1818	0.0816	0.0813	0.5746
logreg	0.9901	0.7083	0.0084	0.0166	0.0747	0.5041
ctree	0.9902	0.5917	0.0523	0.0956	0.1722	0.5259
rforest	0.9902	0.5990	0.0535	0.0978	0.1755	0.5266
adaboost	0.9901	0.6916	0.0042	0.0083	0.0435	0.5020
xgboost	0.9903	0.8248	0.0282	0.0544	0.1496	0.5141

Table 4.4: Stratified 10-fold cross-validation of flaky tests identified by iDFlakies for machine A with metrics differences.

4.2.2 Machine B

In this section you will find the results of machine B which had the following specifications:

- Ubuntu 18.04
- 16 vCPUs
- 128 GB of RAM

Like for machine A the results are organized in the same way. Tables 4.5 and 4.7 uses different sources for the labels and consider the general state of the JVM. Tables 4.6 and 4.8 show the results performed on the data based on the differences (after test execution minus before test execution).

The table 4.5 shows the results of the cross-validations with the absolute JVM metrics data and the own identified flaky tests as the positive class labels. We see that the logreg classifier has the worst scores especially the PREC, RECA and F1 are equal to zero. Furthermore, the MCC and the AUC scores are the lowest in comparison to the other classifiers. The rforest and the ctree classifiers have for all scores a value over 90% (except for RECA of ctree with 89.44%). The xgboost classifier has similar scores like rforest and ctree but the RECA, F1 and MCC are between 85% and 90%. The adaboost classifier has for the previous mentioned scores lower values than xgboost classifier. The values ranges between 78% and 82%. Note that the parametric classifier perform worst according to the F1, MCC and PREC scores.

Classifier	ACC	PREC	RECA	F1	MCC	AUC
lda	0.9911	0.1603	0.9045	0.2723	0.3787	0.9479
qda	0.9886	0.1306	0.9201	0.2287	0.3443	0.9544
gnbayes	0.9766	0.0680	0.9256	0.1268	0.2476	0.9512
logreg	0.9979	0.0	0.0	0.0	-0.0005	0.4999
ctree	0.9996	0.9306	0.8944	0.9119	0.9120	0.9471
rforest	0.9997	0.9355	0.9009	0.9176	0.9177	0.9504
adaboost	0.9993	0.8556	0.7871	0.8192	0.8199	0.8934
xgboost	0.9995	0.9174	0.8559	0.8855	0.8859	0.9279

Table 4.5: Stratified 10-fold cross-validation of own identfied flaky tests for machine B.

The table 4.6 shows the cross-validation results based on the JVM metrics' differences and with the positive class labels for own identified flaky tests. From the table it is obvious to see that all classifiers have poor scores in comparison to all previous results. To note is that here the rforest and the ctree classifiers have similar scores. Furthermore, these two classifiers are the best in the table although their scores are in comparison to other tables low.

Classifier	ACC	PREC	RECA	F1	MCC	AUC
lda	0.9970	0.0	0.0	0.0	-0.0007	0.4986
qda	0.0002	0.0002	1.0	0.0004	0.0	0.5
gnbayes	0.9950	0.1	0.0	0.0	-0.0010	0.4976
logreg	0.9997	1.0	0.0	0.0	0.0	0.5
ctree	0.9997	0.3833	0.05	0.0757	0.0931	0.5249
rforest	0.9997	0.3833	0.0333	0.0472	0.0523	0.5166
adaboost	0.9997	0.9	0.0	0.0	-9.5786e-06	0.4999
xgboost	0.9997	1.0	0.0	0.0	0.0	0.5

Table 4.6: Stratified 10-fold cross-validation of own identfied flaky tests for machine B with metrics differences.

The following table 4.7 shows the cross-validation results based on the absolute JVM metrics and with the positive class labels from the iDFlakies data set. The logreg classifier has the worst RECA, F1, MCC and AUC scores. The scores for the parametric classifiers all below 20% for RECA, F1 and MCC. The non-parametric classifier perform similar or better than the parametric classifiers. The rforest and the ctree classifiers have the highest RECA, F1, MCC and AUC scores. The xg-

boost classifier perform best according to the PREC but the scores for RECA, F1 and MCC are lower than 36%. The adaboost perform for all scores worse than xgboost.

Classifier	ACC	PREC	RECA	F1	MCC	AUC
lda	0.9834	0.1419	0.1074	0.1222	0.1152	0.5502
qda	0.9782	0.1167	0.1566	0.1337	0.1243	0.5719
gnbayes	0.9641	0.0599	0.1595	0.0871	0.0818	0.5661
logreg	0.9892	1.0	0.0	0.0	0.0	0.5
ctree	0.9928	0.7012	0.5890	0.6401	0.6391	0.7931
rforest	0.9931	0.7225	0.5835	0.6455	0.6458	0.7905
adaboost	0.9901	0.8943	0.0933	0.1686	0.2862	0.5466
xgboost	0.9906	0.9185	0.1427	0.2469	0.3598	0.5713

Table 4.7: Stratified 10-fold cross-validation of flaky tests identified by iDFlakies for machine B.

The table 4.8 shows the cross-validation results based on the differences of the JVM metrics with the positive class labels from the iDFlakies data set. In this table we see again as from the previous results that the rforest and the ctree classifiers perform best. In particular they have the best RECA, F1, MCC and AUC scores. The results show that the parametric classifiers have for the RECA, F1 and MCC scores no value over 25%. The xgboost classifier has for all scores low values and the adaboost classifier is also here even worse than the xgboost classifier. These results also show that the rforest and the ctree classifier have similar scores but the rforest is overall slightly better by 1%.

Classifier	ACC	PREC	RECA	F1	MCC	AUC
lda	0.9827	0.0618	0.0452	0.0522	0.0443	0.5189
qda	0.9516	0.0595	0.2425	0.0956	0.1016	0.6008
gnbayes	0.9525	0.0558	0.2203	0.0891	0.0924	0.5903
logreg	0.9895	0.6797	0.0081	0.0160	0.0699	0.5040
ctree	0.9907	0.6077	0.3582	0.4500	0.4619	0.6779
rforest	0.9910	0.6284	0.3653	0.4615	0.4747	0.6815
adaboost	0.9892	0.1342	0.0007	0.0014	0.0037	0.5002
xgboost	0.9898	0.9002	0.0370	0.0708	0.1784	0.5185

Table 4.8: Stratified 10-fold cross-validation of flaky tests identified by iDFlakies for machine B with metrics differences.

For both execution environments machine A and B the rforest and the ctree classifiers perform in general best followed by the xgboost classifier. In particular we also see that the scores are better if we use the absolute JVM metrics and not the differences. This observation is made by machine A and B. If we compare the impact of the different memory size of machine A and B then we also see that machine B with 128GB of RAM have in general better scores than the cross-validation results of machine A with 16 GB of RAM.

Chapter 5

Discussion

Since flaky tests are a big issue in the industry it is important to investigate different mitigation strategies. Our findings show that there are certain cluster structures of flaky tests based on JVM metrics. Such clusters can enhance the predictability of different machine learning models. Indeed, this thesis has shown that flaky tests can be predicted by machine learning models. This approach is quite different in comparison with the current methods for dealing with flaky tests like *Rerun* and *DeFlaker*. It requires more research to optimize a machine learning based on JVM metrics. Furthermore, a deep learning approach could perform differently but it was not a part of this thesis. In general, such classifiers can be integrated in a custom Java test runner to indicate also a passing test as possibly flaky. With such a test runner there would be no need to have only failing tests to do a rerun since we know based on the JVM metrics that also a passing test can be flaky on later runs. In the context of having a continuous integration pipeline such a test runner can improve the developers' efficiency. If the test runner shows that a certain test case is likely to be flaky and the test fails then the developer can rather try to fix the flakiness instead of fixing a nonexistent defect in the code under test. Such a tool will probably have a larger impact on big software projects so that no unnecessary costs of fixing nonexistent defects can be minimized.

The results of this thesis give no essential information why flakiness occurs or what additional root causes of flakiness are which are already discussed in previous related work. But the results supports the results and findings from the Google blog post by *Listfield* [7], that shows a relationship between memory consumption and the probability of flakiness of a test. Furthermore, this thesis uses data with different metrics of different JVM memory areas. This gives a more detailed and specialized results. An important observation is that the general predictability of flaky tests is better with data that uses metrics of the whole JVM as absolute values. Data that uses the difference between the metrics of after a test execution and the metrics before a test execution lead to a poorer predictability. With the differences a test only related data set is used which neglect the absolute values of the JVM.

The findings of this thesis shows that flaky tests can be predicted based on JVM metrics. Future research shall also focus on native applications that are not developed in the Java ecosystem. Furthermore, it would be interesting how flaky tests behave in scripting languages like Python and Javascript. Can flaky tests in those languages also be predicted by machine learning models and memory related data? Since there are also root causes that are network related an extension of the data with network related metrics can be interesting. For example besides having a data set with memory related metrics two additional variables that give some measures about the upload and download sizes of the current test execution. This thesis has also shown that there is a difference between the predictability of flaky tests of machine A and machine B that have different memory sizes. It is worth to investigate more the impact of memory sizes on the flakiness of tests. This

thesis used only two different machines that give not yet a reliable answer if larger memory really minimizes the flakiness of tests. Another aspect which was not investigated so far is the impact of parameter optimization of the classifier on the predictability. In this thesis we used the default parameters for the different classifiers. In machine learning it is usually the case that some improvement can be done by optimizing the parameters of the models.

Another interesting approach for further research would be to investigate the impact of bytecode frequency of flaky tests. Likewise for native applications outside of the Java ecosystem the opcodes frequencies can be used for flaky test classification. This would be a similar approach as how some malware detection software work. If the development of better dynamic strategies for dealing flaky tests progress then the costs of rerun the failing tests in test suite can be lowered. On a large scale project this could have a significant impact by minimizing the useless reruns. As software project become more complex in the future software developer shall give more attention to the impact of flaky tests. This thesis gives a motivation to do further research in predicting flaky tests and to develop more mitigation strategies with its insights.

Chapter 6

Conclusion

In this thesis we investigated the impact of JVM metrics on the predictability of flaky tests. Furthermore, we were interested in if there is a cluster structure in the JVM metrics among flaky tests. The data set of the *iDFlakies* project gives the basis information of flaky tests on 82 open-source projects. In order to collect JVM metrics we developed a toolchain that executed the test suites of the projects with a modified code base so that JVM metrics are collected.

To understand the structure of flaky tests in the data a PCA with a biplot was done. Since there are too many data for giving a plot we needed the project with the most identified flaky tests. The *hadoop* had the most flaky tests and the biplot indicates clearly that the flaky tests are grouped together in a dimension reduced space based on the first and second principle components on absolute JVM metrics.

The following paragraphs will answer the research questions:

RQ1 First the non-parametric models perform in general better than the parametric ones. Especially the random forest and the classification tree models performed best. The extreme gradient boosting model performs third best.

RQ2 Furthermore, the predictability is better by using absolute JVM metrics than only the differences used by a single test.

RQ3 An interesting observation is also that different sizes of RAM also impact the predictability. Data of machine B with 128 GB of RAM generally performed better.

RQ4 The performance of prediction is lower if we used the flaky labels from the *iDFlakies* data set. To use class labels from different data sources will lead to the problem that the execution environment is likely not identical. As mentioned above we see that the RAM size has also an impact.

The insights of this thesis allows to predict flaky tests at a certain degree. The prediction allows to know if a test is flaky although the test passes. It is not necessary anymore to have failing test beforehand in order to identify flaky tests. This thesis should enable future research on predicting flaky tests by machine learning models. There are still many things which are not considered so far (e.g., network usage, flaky tests of non-Java projects, scripting languages, etc.). These new insights can give more motivation in developing new tools for mitigating flaky tests. Such tool have the potential to increase the productivity during development and to lower the costs.

Bibliography

[1] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie, "idflakies: A framework for detecting and partially classifying flaky tests," in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pp. 312–322, 2019.

[2] B. Daniel, V. Jagannath, D. Dig, and D. Marinov, "Reassert: Suggesting repairs for broken unit tests," in *2009 IEEE/ACM International Conference on Automated Software Engineering*, pp. 433–444, 2009.

[3] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, (New York, NY, USA), p. 643–653, Association for Computing Machinery, 2014.

[4] M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli, "Understanding flaky tests: The developer's perspective," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, (New York, NY, USA), p. 830–840, Association for Computing Machinery, 2019.

[5] S. Thorve, C. Sreshtha, and N. Meng, "An empirical study of flaky tests in android apps," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 534–538, 2018.

[6] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "Deflaker: Automatically detecting flaky tests," in *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, (New York, NY, USA), p. 433–444, Association for Computing Machinery, 2018.

[7] J. Listfield, "Where do our flaky tests come from?," in *Google Testing Blog*, 2017.

[8] https://subversion.apache.org/.

[9] https://www.apache.org/.

[10] "Rerun failing tests." http://maven.apache.org/surefire/maven-surefire-plugin/examples/rerun-failing-tests.html. Accessed: 2020-07-01.

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

[12] https://github.com/.

[13] https://sites.google.com/view/flakytestdataset.

[14] https://www.zi.uzh.ch/en/teaching-and-research/science-it/infrastructure/sciencecloud/.

[15] https://docs.oracle.com/javase/1.5.0/docs/api/java/lang/instrument/package-summary.html.

[16] E. Bruneton, R. Lenglet, and T. Coupaye, "Asm: A code manipulation tool to implement adaptable systems," in *In Adaptable and extensible component systems*, 2002.

[17] https://bytebuddy.net/.

[18] http://javaparser.org/.

[19] https://metrics.dropwizard.io/.

[20] B. Ghotra, S. McIntosh, and A. E. Hassan, "A large-scale study of the impact of feature selection techniques on defect classification models," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pp. 146–157, 2017.

[21] M. A. Hall and L. A. Smith, "Subset selection : A correlation based filter approach," 1997.

[22] https://xgboost.readthedocs.io/en/latest/index.html.

[23] J. Micco, "The state of continuous integration testing at google." http://aster.or.jp/conference/icst2017/program/jmicco-keynote.pdf, 2017.

[24] L. Qian, G. Zhou, F. Kong, and Q. Zhu, "Semi-supervised learning for semantic relation classification using stratified sampling strategy," in *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing*, (Singapore), pp. 1437–1445, Association for Computational Linguistics, Aug. 2009.

Appendix

Metric	Explanation
heap.committed	The amount of memory in bytes that is committed for the Java virtual machine to use
heap.init	The amount of memory in bytes that the Java virtual machine initially requests from the operating system for memory management
heap.max	The maximum amount of memory in bytes that can be used for memory management
heap.usage	Ratio used / max
heap.used	The amount of used memory in bytes
non-heap.committed	The amount of memory in bytes that is committed for the Java virtual machine to use
non-heap.init	The amount of memory in bytes that the Java virtual machine initially requests from the operating system for memory management
non-heap.max	The maximum amount of memory in bytes that can be used for memory management
non-heap.usage	Ratio used / max
non-heap.used	The amount of used memory in bytes
pools.Code-Cache.committed	The amount of memory in bytes that is committed for the Java virtual machine to use
pools.Code-Cache.init	The amount of memory in bytes that the Java virtual machine initially requests from the operating system for memory management
pools.Code-Cache.max	The maximum amount of memory in bytes that can be used for memory management
pools.Code-Cache.usage	Ratio used / max
pools.Code-Cache.used	The amount of used memory in bytes
pools.Compressed-Class-Space.committed	The amount of memory in bytes that is committed for the Java virtual machine to use
pools.Compressed-Class-Space.init	The amount of memory in bytes that the Java virtual machine initially requests from the operating system for memory management
pools.Compressed-Class-Space.max	The maximum amount of memory in bytes that can be used for memory management
pools.Compressed-Class-Space.usage	Ratio used / max
pools.Compressed-Class-Space.used	The amount of used memory in bytes
pools.Metaspace.committed	The amount of memory in bytes that is committed for the Java virtual machine to use
pools.Metaspace.init	The amount of memory in bytes that the Java virtual machine initially requests from the operating system for memory management
pools.Metaspace.max	The maximum amount of memory in bytes that can be used for memory management
pools.Metaspace.usage	Ratio used / max
pools.Metaspace.used	The amount of used memory in bytes

Table 6.1: Explanation of detailed memory metrics part 1

Metric	Explanation
pools.PS-Eden-Space.committed	The amount of memory in bytes that is committed for the Java virtual machine to use
pools.PS-Eden-Space.init	The amount of memory in bytes that the Java virtual machine initially requests from the operating system for memory management
pools.PS-Eden-Space.max	The maximum amount of memory in bytes that can be used for memory management
pools.PS-Eden-Space.usage	Ratio used / max
pools.PS-Eden-Space.used	The amount of used memory in bytes
pools.PS-Eden-Space.used-after-gc	
pools.PS-Old-Gen.committed	The amount of memory in bytes that is committed for the Java virtual machine to use
pools.PS-Old-Gen.init	The amount of memory in bytes that the Java virtual machine initially requests from the operating system for memory management
pools.PS-Old-Gen.max	The maximum amount of memory in bytes that can be used for memory management
pools.PS-Old-Gen.usage	Ratio used / max
pools.PS-Old-Gen.used	The amount of used memory in bytes
pools.PS-Old-Gen.used-after-gc	
pools.PS-Survivor-Space.committed	The amount of memory in bytes that is committed for the Java virtual machine to use
pools.PS-Survivor-Space.init	The amount of memory in bytes that the Java virtual machine initially requests from the operating system for memory management
pools.PS-Survivor-Space.max	The maximum amount of memory in bytes that can be used for memory management
pools.PS-Survivor-Space.usage	Ratio used / max
pools.PS-Survivor-Space.used	The amount of used memory in bytes
pools.PS-Survivor-Space.used-after-gc	
total.committed	The amount of memory in bytes that is committed for the Java virtual machine to use
total.init	The amount of memory in bytes that the Java virtual machine initially requests from the operating system for memory management
total.max	The maximum amount of memory in bytes that can be used for memory management
total.used	The amount of used memory in bytes

Table 6.2: Explanation of detailed memory metrics part 2

Metric	Explanation
PS-MarkSweep.count	The total number of collections that have occurred
PS-MarkSweep.time	The approximate accumulated collection elapsed time in milliseconds
PS-Scavenge.count	The total number of collections that have occurred
PS-Scavenge.time	The approximate accumulated collection elapsed time in milliseconds

Table 6.3: Explanation of garbage collector metrics

Metric	Explanation
blocked.count	Current number of threads in "blocked" state in this JVM
count	Current number of threads in this JVM
daemon.count	Current number of live daemon threads in this JVM
deadlock.count	Current number of deadlocked threads in this JVM
deadlocks	Collection of information about the currently deadlocked threads
new.count	Current number of threads in "new" state
runnable.count	Current number of threads in "runnable" state in this JVM
terminated.count	Current number of threads in "terminated" state
timed_waiting.count	Current number of threads in "timed_waiting" state in this JVM
waiting.count	Current number of threads in "waiting" state in this JVM

Table 6.4: Explanation of thread metrics