Tampere University

Markus Mulkahainen

TEST CASE SELECTION AND PRIORITIZATION IN CONTINUOS INTEGRATION ENVIRONMENT

Faculty of Information Technology and Communication Sciences (ITC)
Master of Science Thesis
March 2019

ABSTRACT

MARKUS MULKAHAINEN: Test case selection and prioritization in continuous integration environment
Tampere University
Master of Science Thesis, 60 pages, 4 Appendix pages
March 2019
Master's Degree Programme in Information Technology
Major: Software Engineering
Examiners: Professor Kari Systä and professor Hannu-Matti Järvinen

Keywords: test case selection, test case prioritization, machine learning, incremental learning, regression testing, continuous integration, coverage, instrumentation

It is beneficial for continuous integration (CI), that building and testing a software happens as quickly as possible. Sometimes, when a test suite grows large during the lifecycle of the software, testing becomes slow and inefficient. It is a good idea to parallelize test executions to speed up testing, but in addition to that, test case selection and prioritization can be used. In this case study, we use incremental machine learning techniques to predict failing and passing tests in the test suite of existing software from the space industry and execute only test cases that are predicted failing. We apply such test case selection techniques to 35 source code modifying commits of the software and compare their performances to traditional coverage based selection techniques and other heuristics. Secondly, we apply different incremental machine learning techniques in test case prioritization and compare their performances to traditional coverage based prioritization techniques. We combine features that have been used successfully in previous studies, such as code coverage, test history, test durations and text similarity to separate passing and failing tests with machine learning. The results suggest, that certain test case selection and prioritization techniques can enhance testing remarkably, providing significantly better results compared to random selection and prioritization. Additionally, incremental machine learning techniques require a learning period of approximately 20 source code modifying commits to produce equal or better results than the comparison techniques in test case selection. Test case prioritization techniques with incremental machine learning perform significantly better than the traditional coverage based techniques, and they can outweigh the traditional techniques in the weighted average of faults detected (APFD) values immediately after initial training. We show that machine learning does not need a rigorous amount of training to outperform traditional approaches in test case selection and prioritization. Therefore, incremental machine learning suits test case selection and prioritization well, when initial training data does not exist.

TIIVISTELMÄ

MARKUS MULKAHAINEN: Testien valinta ja priorisointi jatkuvassa integraatiossa
Tampereen yliopisto
Diplomityö, 60 sivua, 4 liitesivua
Maaliskuu 2019
Tietotekniikan diplomi-insinöörin tutkinto-ohjelma
Pääaine: Ohjelmistotuotanto
Tarkastajat: professori Kari Systä ja professori Hannu-Matti Järvinen

Avainsanat: testien valinta, testien priorisointi, koneoppiminen, regressiotestaus, jatkuva integraatio, koodikattavuus, instrumentointi

Jatkuvan integraation toimivuuden edellytyksenä on, että ohjelmiston kääntäminen ja testaaminen tapahtuu mahdollisimman nopeasti. Ohjelmiston kehitystyön edetessä automaattisesti ajettavien testien määrä voi kasvaa suureksi. Tällöin on olemassa riski, että testaaminen hidastuu ja jatkuva integraatio kärsii sen seurauksena. Testejä voidaan nopeuttaa esimerkiksi rinnakkaistamalla testiajoja, mutta sen lisäksi testejä voidaan myös priorisoida tai testeistä voidaan valita vain pieni määrä ajettaviksi. Tässä tapaustutkimuksessa tutkimme testien valintaa ja priorisointia koneoppimisen avulla. Valitsemme ajettaviksi ainoastaan sellaiset testit, jotka koneoppimismallit ennustivat hajoaviksi. Koneoppimismallit päättelevät testien lopputulemia eri tietolähteitä yhdistelemällä. Näitä tietolähteitä ovat mm. koodikattavuus, testien ajohistoria, testien kestoaika ja testien ja koodimuutosten samankaltaisuus. Käytämme tällaista testien valintaa aineistoon, joka on kerätty avaruusteollisuuden ohjelmistoprojektista. Vertaamme koneoppimisen avulla saatuja tuloksia perinteisiin testien valintamenetelmiin ja heuristiikkoihin. Tapaustutkimuksessa vertailemme myös koneoppimisen avulla suoritettua testien priorisointia perinteisiin koodikattavuuspohjaisiin priorisointimenetelmiin. Tutkimuksen tulokset osoittavat, että tietyt testien valinta- ja priorisointimenetelmät tehostavat testaamista huomattavasti ja tuottavat merkittävästi parempia tuloksia kuin satunnaisuuteen perustuvat menetelmät. Tämän lisäksi tulokset osoittavat, että testien valinnassa koneoppimismenetelmät saavuttavat samankaltaisen tai paremman tuloksen kuin paras heuristiikka noin kahdenkymmenen koodimuutoksen jälkeen. Testien priorisoinnissa koneoppimismenetelmät tuottavat merkittävästi parempia tuloksia kuin vertailumenetelmät. Tutkimuksen tulokset osoittavat, että koneoppimismenetelmät eivät välttämättä tarvitse suuria määriä koulutusdataa, vaan voivat ennustaa pienelläkin määrällä koulutusdataa testien lopputulemia paremmin kuin vertailumenetelmät.

PREFACE

I am grateful for several people that contributed in this thesis. This master's thesis was created in co-operation with *Space Systems Finland*. From *SSF*, I want to thank Timo Latvala for the oppourtunity. Thank you Jaakko Hujanen for the support throughout the project. Thanks Ismo Toijala and Viorel Preoteasa. Thank you Tomi Räty from *VTT* for the discussions and emails. Thank you Árpád Beszédes, Benjamin Busjaeger and especially Helge Spieker for answering to my emails and clearing specific things out for me. Your insights and knowledge were invaluable to me. From my university, I want to thank professors Kari Systä and Hannu-Matti Järvinen. Thank you family members and friends. Thank you Tiina for the persistent support along these years. If you are still reading, thanks to you as well, dear reader.

In Helsinki, Finland, on 4 March 2019

Markus Mulkahainen

CONTENTS

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

TCS	Test case selection
TCP	Test case prioritization
CI	Continuous integration
ML	Machine learning
SUT	System under test
MCC	Matthews correlation coefficient
TSR	Test suite reduction
APFD	Weighted average of the percentage of faults detected
TP	Set of true positives
FP	Set of false positives
TN	Set of true negatives
FN	Set of false negatives
P	A program or software
P'	A modified version of P
T	Original test suite
T'	Reduced or prioritized test suite
F_T	Set of faults found by T
$F_{T'}$	Set of faults found by T'
f_T	Set of failing tests in T
$f_{T'}$	Set of failing tests in T'
T_{tmod}	Set of test cases selected with transitive dependency selection
T_{smod}	Set of test cases selected with any other TCS technique
$\forall$	For all
$\exists$	There exists
$\emptyset$	Empty set
$\in$	Is an element of
$\approx$	Is approximately equal to
$\neq$	Is not equal to
$\ll$	Is much less than
$\gg$	Is much greater than
$\subset$	Is a proper subset of
$\subseteq$	Is a subset of
$\cup$	Set union
$\cap$	Set intersection

1. INTRODUCTION

Continuous integration (CI) is a software engineering practice where a team of developers create software in frequent increments. The core principle is to integrate developers work often with a stable version of the software. The name, continuous integration, straightly refers to this activity of integrating new features and enhancements continuously to an existing software product. CI allegedly has many benefits. It pursues to ease the burden of developers by automating parts of the software code integration process and encouraging developers to commit their work more often. Small feature increments help to maintain the product functional at all times and to ensure that bugs show up quickly [16]. Additionally, CI can help to release software versions twice as often compared to projects without CI [20]. Applying CI practices and tools in any modern software project is therefore well justified.

Testing in CI should be automatic [42][16]. Firing off the test suite automatically after every change shifts the responsibility for testing from developers to the tools. Automatic testing ensures that each code increment is validated and checked for bugs without user intervention. Automatic testing speeds up the software product development process and makes it possible to ship a quality release when necessary, even every day [42].

A key element to successfully adopt CI is to have efficient and fast software building and testing processes [16]. Test suites tend to grow large during the lifecycle of software, and sometimes this can cause problems in keeping the test suites fast. The vast number of tests is not the only problem, but sometimes single test executions require long time periods to finish. In the context of *Space Systems Finland*, the problem culminates in validation tests, which exercise multiple features in an end-to-end manner. A validation test that runs for 20 minutes is not abnormal. Slow test cases and test suites reflect to CI pipelines and render them counterproductive. Test case selection and prioritization can be used in such situations to speed up and enhance testing.

Test case selection (TCS) selects a subset of the test suite to test a modified program for regressions. Ideally, the subset of tests finds the same number of faults with lesser effort compared to the original test suite. Quite often, however, the reduced test suite does not include all fault-revealing test cases, but TCS discards some of them. A somewhat safer approach is to apply test case prioritization (TCP) to the test suite. It does not reduce the number of tests in the suite, but merely re-orders them according to fault-revealing capabilities. The tests cases, that are considered to be more likely to find a fault, are moved to the top of the test suite allowing early test execution to reveal faults as soon as possible. TCP seeks to provide a maximum benefit to the tester, even if the test suite execution is halted at some point [44]. Even though such an interruption is possible, TCP executes the

whole test suite by default. The goal of TCP does not have to be early fault-detection, but it can be, for example, to maximize coverage as fast as possible instead [36].

The goal of this thesis is to explore and implement different TCS and TCP techniques in order to speed up testing and facilitate continuous integration. We found machine learning (ML) based techniques especially interesting. Machine learning was used in TCP already in 2006 [41], but during the past few years, it has become more popular in the domains of TCS and TCP [10][40][24][23]. These studies have shown promising results for the novel techniques, strengthening the viewpoint of ML being an efficient alternative for the traditional TCS and TCP approaches. The results of our thesis support the latest advancements in ML-based TCS and TCP and we propose ways to improve them. For example, we bring in more evidence that ML-based TCP surpasses the traditional coverage based approaches in the weighted average of percentage of faults detected (APFD), even without a rigorous amount of initial training data. We propose, that applying ML incrementally is beneficial in cases where no existing training data is available.

In the case study, we use test verdict history (e.g. information about past executions of a test), code coverage, test durations and text similarity to predict failing tests in the test suite. Our case study is perhaps closest to the study by Busjaeger and Xie [10], but in contrast to them, we show that with incremental learning we can speed up commissioning ML based TCP techniques, when no initial training data exists. While Busjaeger and Xie used machine learned ranking, we use different machine learning classifiers to classify test cases into categories of failing and passing using *scikit-learn* [31]. This is especially useful in TCS, where we can draw a decision boundary between the failing and passing tests as opposed to selecting top n tests from the prioritized list. Furthermore, we apply TCP by ordering test cases with class probabilities using the same classifiers as in TCS.

We separate the way how TCS and TCP handle commits that modify tests and source code. This has little effect in TCP results but has a more notable effect in TCS results. Every time a commit modifies only tests or their dependencies in the repository, we select all test cases that depend transitively on the changed file. For example, in C-language such a dependency is declared with *#include* directive. This technique is similar to techniques described by Yoo et al. [45] and Gligoric et al. [17]. While we show that this kind of transitive test case selection can reduce the number of tests significantly when applied to test modifying commits, we think that it unnecessarily complicates TCS and TCP schemes, especially in light of modern software engineering practices that encourage feature branching and merge requests, where tests and source code are modified together. As an alternative approach, we propose the dependency coverage score, introduced by Yoo et al. [45], which can help to overcome this complexity.

We assume, that the tests and source code are maintained in the same repository. For the sake of simplicity, we also assume, that tests exist in *test/* directory, and source code under *src/* directory. We consider source modifying commits being the ones that modify any file under *src/*, be it source code or configuration files, and test modifying commits the ones

that modify any file under *test/*, respectively. A commit can also modify both directories, *src/* and *test/*, and we call these commits ”source and test modifying commits”. While we use our techniques to select and prioritize validation tests, there is no restriction to use any of the techniques for integration and unit tests.

In this case study, we show that ML can be applied in TCS and TCS incrementally when no initial training data is available. We show that random forest classifier is among the best performing classifiers in TCS and TCP and that it produces higher scores and converges faster than multilayer perceptron, for example. We show that using features such as modification coverage, test history, and text similarity TCS can produce equal or better MCC scores than traditional coverage-based approaches in approximately 20 commits. Furthermore, we show that transitive dependency selection can significantly reduce the number of tests. Finally, we show that the incremental learning TCP techniques surpass coverage-based prioritization techniques in APFD scores even faster: almost immediately after initial training.

SSF is a Finnish software company specialized in industrial systems, including high-reliability software in the space domain. The system under test (SUT), i.e. the software on which we apply TCS and TCP techniques is a satellite instrument control software and will be part of the Meteosat Third Generation Sounder (MTG-S) satellites that are launched in the 2020s. The instrument, namely Sentinel-4/UVN, is a high-resolution spectrometer and will be used to monitor air quality parameters over Europe. The SUT is a command and control software for the Sentinel-4/UVN instrument. The software and its validation tests are programmed in Ada-language.

The research questions are the following:

RQ1 How big test suite reduction can code coverage based test case selection achieve?

RQ2 How effective is incremental learning based test case selection?

RQ3 How do the incremental learning based test case prioritization techniques compare to traditional coverage based prioritization techniques?

2. THEORETICAL BACKGROUND

This chapter covers certain topics that are needed to understand the later chapters. Our case study covers topics from regression testing, software engineering, machine learning, and information retrieval. We do not introduce each field thoroughly but instead skim through some important topics that help to understand our case study. We start with continuous integration in section 2.1. Then we discuss code coverage and instrumentation in section 2.2. We introduce test case selection, minimisation and prioritization in sections 2.3, 2.4 and 2.5. We introduce dependency coverage and transitive dependency selection in section 2.6. We discuss machine learning in section 2.7. Finally, we present a non-exhaustive list of related work in section 2.8.

2.1 Continuous integration

Continuous integration is a software engineering practice where developers merge and integrate their work several times a day into a single source code repository [38]. The repository should be equipped with automated building and testing tools [21]. CI is a combination of tooling and best practices where the tooling includes 1) building and 2) testing the software automatically [20]. Best practices include integrating early and not keeping code changes in the local workspace for too long [42]. The benefits of continuous integration are early detection of faults and speed in the software lifecycle from development to production [20].

Hilton et al. [20] investigated the use of continuous integration in 34,544 open source projects on Github. They found out, that 40% of the projects use continuous integration platforms such as Travis or CircleCI, and that the number is rising. Within the most popular open source projects, 70% of the projects relied on a continuous integration platform. Hilton et al. reported, that continuous integration helped the developers to release software versions twice as often and accept pull requests faster compared to projects that did not use continuous integration.

Zhao et al. [46] also studied the use of Travis in Github projects. They reported, that there is a positive trend in the number of closed pull requests over time in a project's lifecycle regardless whether CI is applied or not, but after adopting CI the positive trend actually flattens and becomes less steep. The results of Zhao et al. do not endorse the results of Hilton et al., but says quite the opposite: the positive trend in closed pull requests slows down after Travis is adopted. However, Zhao et al. identified other benefits for the use of Travis, such as increased number of merge commits and automated tests.

Shahin et al. [38] pointed out that approaches and tools that aim to reduce build and test

time support and facilitate continuous integration and other continuous practices. Test case selection and prioritization were seen as such supporting techniques. Indeed, one of the biggest motivation behind our case study was to enhance and speed up testing in order to enable efficient use of continuous integration.

2.2 Code coverage

In software testing, code coverage determines which parts of the system under test (SUT) are exercised by a test suite [15]. Code coverage can be expressed, for example, as the percentage of software statements exercised by the test cases in the test suite. The percentage of statements covered by a test suite indicates how completely the software is tested according to statement coverage criterion [47]. A well-tested program could yield a statement coverage of 100% indicating that the test suite exercises every statement in the software. However, 100% statement coverage does not imply that the software is well tested [4]. The precision on which the software code is surveyed is called the *coverage criterion*. Myers and Sandler [30] considers statement coverage as a weak criterion, and they introduce four stronger criterions: decision or branch coverage, condition coverage, decision-condition coverage, and multiple-condition coverage.

Decision or branch coverage is a stronger coverage criterion compared to statement coverage. It requires, that each decision, e.g. if-else, do-while or switch, is evaluated to both true and false at least once. For example, 100% branch coverage implies that the test suite exercises every if-statement in the software so, that both output branches, true and false, are taken. Generally, full branch coverage leads to full statement coverage, but according to Myers and Sandler, branch coverage does not satisfy statement coverage in certain special cases. An example of such a special case is a software code without any decisions. Therefore the definition of branch coverage has been expanded to require fulfilling statement coverage too. [30]

The code coverage information can be attained through code instrumentation [2]. Amman and Offutt [2] explain, that an *instrument* is an additional software code, that collects information from the software, but does not affect the functional behavior of the software. The statement coverage can be measured by placing an instrument between every statement in the software code and executing the test suite against the instrumented software. This results in a log of visited software statements, e.g. set of visited statements S_v. The statement coverage is then $\frac{|S_v|}{|S|} \times 100$ where S is the set of all software statements. Even though code instrumentation does not affect the functionality of the software, it can cause side-effects such as concurrency or timing issues [2]. In practical applications, statement coverage is often confused with line coverage [3].

2.3 Test case selection

Test case selection techniques are a group of regression testing techniques where a subset of the test suite is selected for execution. It reduces the number of tests to be run allowing

a shorter execution time, but at the same time risks neglecting *fault-revealing* test cases. Rothermel and Harrold [33] initially defined test case selection, but it was reformulated by Yoo and Harman [44] as following:

1. Given the program P, the modified version of it, P' and a test suite, T.
2. Find $T' \subseteq T$ to test P'.

According to Rothermel and Harrold [33], a test case $t \in T$ is *modification-revealing*, "if and only if it causes the outputs of P and P' to differ". Furthermore, Biswas et al. [7] specify that a modification-revealing test case t produces a different output for P and P'. We therefore interpret, that if t fails in P but succeeds in P', or t fails in P' but succeeds in P, t is modification-revealing.

Rothermel and Harrold [33] separate safe and unsafe regression test selection techniques. They explain that safe techniques include every modification-revealing test case from the test suite. Unsafe techniques respectively omit modification-revealing test cases from T'. *Inclusiveness* defined by Rothermel and Harrold [33] can be used to report the proportion of modification-revealing test cases in the reduced test suite. Inclusiveness is defined as [33]

$$\text{Inclusiveness} = \frac{m}{n} \times 100 \tag{2.1}$$

where m is the number of modification-revealing tests in the reduced test suite, and n is the number of modification-revealing tests in the original test suite. Rothermel and Harrold clarify, that any technique that provides 100% inclusiveness, is considered a safe regression test selection technique.

A test case is considered *modification-traversing*, if it fulfills one of the two criterions [44][33]:

- it executes new or modified code in the new version of the software, or
- it used to execute code that was deleted in the new version of the software

Test case selection techniques have been evaluated in the literature using metrics such as test suite reduction (TSR) and reduction in fault detection effectiveness [12][34][11]. Test suite reduction is expressed as [34]

$$\text{Test Suite Reduction} = \left(1 - \frac{|T'|}{|T|}\right) \times 100 \tag{2.2}$$

where T is the original test suite and T' is the reduced test suite. Reduction in fault detection effectiveness is given as [34]

$$\text{Reduction in Fault Detection Effectiveness} = \left(1 - \frac{|F_{T'}|}{|F_T|}\right) \times 100 \tag{2.3}$$

where F_T is the set of faults found by the original test suite T and $F_{T'}$ is the set of faults found by the reduced test suite T'. In the optimal situation, test case selection techniques provide a maximum reduction in test suite size, but at the same time reveal the same number of faults as the original test suite. It is therefore beneficial for test case selection techniques to have maximal test suite reduction and minimal reduction in fault detection effectiveness.

In our case study, F_T is unknown, e.g. we do not know the number of actual faults in the system. We only know which tests are failing, but a failing test does not always reveal one unique fault. One failing test can reveal 0, 1, 2 or even more actual faults. Because we do not know F_T, we cannot use reduction in fault detection effectiveness to measure the performance of TCS techniques. Instead, the closest we can get, is to assume that finding the failing test cases helps us to find the actual faults in the system. Therefore, we use TCS techniques to find the failing test cases, f_T, from T. We do not want the reduced test suite to contain anything else but the failing test cases. In addition to TSR, our objective becomes to maximize the proportion of test failures in the reduced test suite T':

$$\frac{|f_{T'}|}{|f_T|} \times 100 \tag{2.4}$$

where $f_{T'}$ is the set of failing tests in T', and f_T is the set of failing tests in original test suite T. The expression 2.4 is interesting. We can rewrite it with *true positives* (TP), *false positives* (FP), *true negatives* (TN) and *false negatives* (FN). We follow Knauss et al. [22], and describe true/false positives/negatives as following:

- TP: Test cases that were selected to the reduced test suite and failed (failed and predicted failing)
- FP: Test cases that were selected to the reduced test suite but passed (passed but predicted failing)
- TN: Test cases that were not selected to the reduced test suite and passed (passed and predicted passing)
- FN: Test cases that were not selected to the reduced test suite but failed (failed but predicted passing)

Removing the multiplication ($\times 100$), we can rewrite the expression 2.4 with TP, FP, TN and FN:

$$\frac{|f_{T'}|}{|f_T|} = \frac{TP}{TP+FN} \tag{2.5}$$

which is the same as *recall* in information retrieval theory [14][32]. We can do the same for test suite reduction, and rewrite it with TP, TN, FN and FP:

$$\begin{aligned} \text{Test Suite Reduction} &= 1 - \frac{|T'|}{|T|} = \frac{|T| - |T'|}{|T|} \\ &= \frac{(TN+FN+FP+TP) - (TP+FP)}{TN+FN+FP+TP} \\ &= \frac{TN+FN}{TN+FN+FP+TP} \end{aligned} \tag{2.6}$$

We now have two conflicting performance scores for TCS techniques: test suite reduction and recall. We can use these scores to measure 1) reduction in test suite size and 2) proportion of failing tests in T'. We would like to find a such TCS technique, that maximizes both of these scores. It is not easy, because increasing 1) potentially decreases 2) and vice versa. Furthermore, it is not easy to differentiate two almost equally performing techniques. Consider the following example with two TCS techniques $T1$ and $T2$:

- T1: Test suite reduction 0.95, recall 0.75
- T2: Test suite reduction 0.87, recall 0.82

It is hard to say which one of the two, $T1$ or $T2$, is better. We introduce another score, the Matthews correlation coefficient (MCC). We found MCC to be a representative surrogate for the combination of test suite reduction and recall. The MCC-score is a single value and gives us a more robust way to compare performances of TCS techniques. B.W. Matthews introduced the MCC-score in 1975 [28] and defined it as:

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}} \tag{2.7}$$

The highest possible MCC score is 1. It is achieved when $FP = \emptyset$, $FN = \emptyset$, $TP \neq \emptyset$ and $TN \neq \emptyset$. In such case, we do not have incorrect predictions. We correctly categorized all tests into passes and fails. Selecting only the failing predictions, we get the ”perfect selection”. The perfect selection never fully satisfies test suite reduction, e.g. $TSR \neq 1$, but

```
1 - unsigned int eeprom_address = 0x1234;
2 + unsigned int eeprom_address = 0x1235;
3
4 unsigned int get_eeprom_address(void)
5 {
6   return eeprom_address;
7 }
```

Program 2.1. *Code coverage based test case selection is unable to trace changes in global variables. The minus sign in front of the code line indicates that the line was deleted, the plus sign indicates that the line was added.*

always results in the maximum recall value of 1. In other words, $MCC = 1$ evaluates to highest possible test suite reduction for a recall of 1.

Knauss et al. [22] used F_1 score (equation 2.8), among precision (equation 2.9) and recall to measure the performance of a test case selection technique. In our case, the F_1 score was not a good option because we have a high class-imbalance in our dataset, e.g. less than 1% of the samples are positive. Boughorbel et al. [8] mention that F_1 score is sensitive to data imbalance, but MCC score handles class imbalance well. We considered the disadvantage of MCC score being that it values false positives and false negatives similarily, i.e. it is invariant to the changes in false positives and false negatives when their sum is constant. We argue, that having small amount of false negatives and greater amount of false positives is more beneficial than the contrary in test case selection. Therefore, we would have liked to add bias to the MCC score to penalize false negatives more than false positives, but we did not do this.

$$F_1 = \frac{2 \times \text{recall} \times \text{precision}}{\text{recall} + \text{precision}} \tag{2.8}$$

$$precision = \frac{TP}{TP + FP} \tag{2.9}$$

Code coverage based test case selection is not able to trace every kind of change in the codebase. These generally are the non-instrumentable parts of the code repository, such as meta- or configuration files, but also source code. For example, a global variable value change cannot be traced. Consider the program code 2.1. Even though the value of the *eeprom_address* variable is changed, it remains invisible to code coverage based test case selection.

2.3.1 Confusion matrix

The Figure 2.1 shows a confusion matrix. Confusion matrix in a binary classification problem is a two-by-two matrix, that contains the numbers of correctly and incorrectly

classified examples [39]. We separate these examples into bins of *true negatives*, *true positives*, *false positives* and *false negatives*. The confusion matrix lays out these bins in the four cells of the matrix.

Fawcett [14] describes each bin coherently: *True positives* are examples that are classified as positive, and their actual outcomes are positive. *True negatives* are classified as negative, and their actual outcomes are negative. *False positives* are classified as positive, but their outcomes are negative. Finally, *false negatives* are classified as negative, but their outcomes are positive.

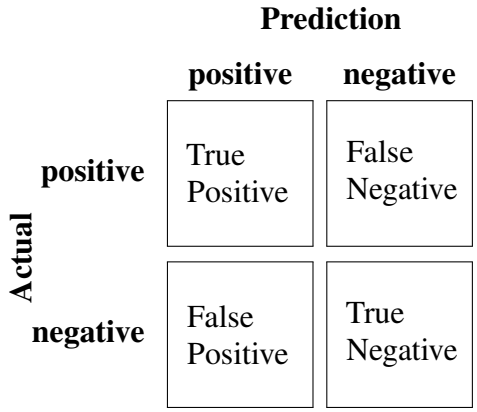

Figure 2.1. *A two-by-two confusion matrix can be used to evaluate binary classification (after Fawcett [14]).*

2.3.2 Coverage overlapping

Coverage overlapping happens when multiple test cases exercise the same areas in the SUT. If two or more test cases exercise e.g. the same software statements, we say that coverage overlapping is present. End-to-end tests tend to have a high coverage overlapping because of their high total coverage.

The Figure 2.2 illustrates test suite of 528 tests and their coverage over the SUT. In this particular software version, over 20 percent of the software statements are covered by more than 500 tests. If we modify the software in this area and then apply coverage-based TCS, the selection will most likely contain a lot of tests, e.g. $T' \approx T$. Test suite reduction will be therefore minimal. Generally, a modification in a densely covered area of the software can prevent TCS techniques to bring any benefit to the tester. Harrold et al. [19] pointed out a similar observation.

The Figure 2.2 reveals, that less than 20 percent of the software statements are covered by 1 to 10 test cases. A modification in this area of the software brings a large reduction in test suite size, e.g. $|T'| \ll |T|$. Generally, a modification in a loosely covered area of the software brings a great benefit to the tester in terms of test suite reduction.

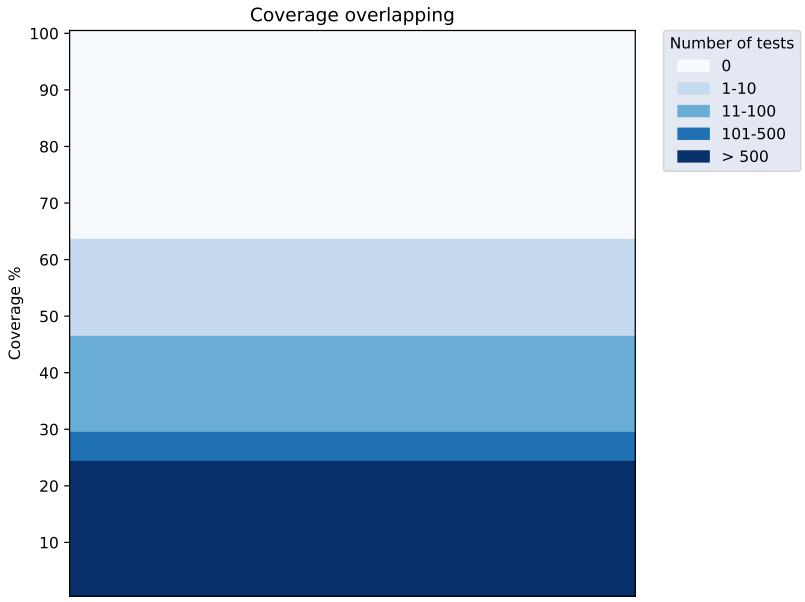

Figure 2.2. *Total statement coverage of a test suite. The darker the area, the greater number of tests exercise same part of the software. Test suite covers* 64% *of all software statements in this particular software version.*

The Figure 2.2 shows, that surprisingly big part of the software, 36%, is not covered by any test case. A modification in this part of the software yields 100% test suite reduction, e.g. $T' = \emptyset$. Such T' is not beneficial to the tester.

Wong et al. [43], Beena and Sarala [5] and Beszédes et al. [6] used priority-based test case selection to further reduce the test suite after applying TCS. Whenever a coverage-based TCS technique selected a high number of tests due to coverage overlapping, they applied an extra prioritization step to discard redundant test cases. For example, some of them applied additional coverage prioritization strategies to discard test cases with overlapping coverage. Such strategies can be beneficial when a TCS technique produces a small test suite reduction, e.g. when $T' \approx T$.

2.4 Test suite minimization

Even though test suite minimization is not studied in this thesis, it is an important regression testing technique and is thus shortly introduced. Test suite reduction problem definition was given by Harrold et al. [18], but it was later named to test suite minimization. Yoo and Harman [44] defines it as follows:

1. Given: A test suite, T, a set of test requirements $r_1, ..., r_n$, that must be satisfied to provide the desired 'adequate' testing of the program, and subsets of $T, T_1, ..., T_n$, one associated with each of the r_is such that any one of the test cases t_j belonging to T_i can be used to achieve requirement r_i.
2. Problem: Find a representative set, T', of test cases from T that satisfies all r_is.

Test suite minimization is similar to test case selection problem, but the key difference, according to Yoo and Harman [44], is whether the changes of the system are examined. Test case selection is interested in the changes of the system under test, while test suite minimization considers only a single version of the system. Test suite minimisation and selection use similar metrics for evaluation, namely test suite reduction (equation 2.2) and reduction in fault detection effectiveness (equation 2.3) [11].

2.5 Test case prioritization

Test case prioritization is a regression testing technique [44], that reorders the test cases in a test suite to maximize a specific *goal* [36]. The goal can be, for example, to detect faults as early as possible, to detect high-risk faults as early as possible, or to maximize the coverage of the system under test as fast as possible [36]. When the goal is early detection of faults, test case prioritization re-orders a test suite so, that the most potential tests to find a fault are executed first. This way, the tester will have faster feedback of failures and further execution of the test suite can be even halted if a failing test is found.

Arguably, early detection of faults is the most famous goal for prioritization, and many of the previous studies have used it as the goal [6][36][10][11]. This thesis does not make an exception but uses the same goal to differentiate well and poorly performing prioritization techniques. Test prioritization does not reduce the number of tests, and therefore test suite reduction and fault detection effectiveness remain unaffected. This lies in the assumption that the test suite execution is not halted.

The property on which the order is based on is called the *surrogate* [44]. The assumption is, that early maximization of the surrogate leads to maximization of the goal [44]. For example, if the surrogate was coverage, the test suite would be reordered descending by the number of statements covered. Previous studies have shown such surrogate to be significantly better than random ordering to maximize the goal of detecting faults early [11]. The surrogate does not have to be a single property, but it can also be a set of properties.

Formally, test case prioritization is defined as [44][36]:

1. Given: T, a test suite, PT, the set of permutations of T, and f, a function from PT to the real numbers.
2. Problem: Find $T' \in PT$ such that $\forall T'' \in PT$, $T'' \neq T'$: $f(T') \geq f(T'')$.

where f is a function that returns an *award value*, where a higher value represents a better ordering with respect to the goal. For example, if the goal was early detection of faults, f could be a function returning the weighted average of the percentage of faults detected (APFD) [36].

The APFD (equation 2.10) metric was first presented by Rothermel et al. [36], and it has been used ever since to evaluate and compare different prioritization techniques. APFD is

```
1  while (a) {
2    if (b) {
3      s1; // modified
     } else {
4      s2;
5      s3;
     }
6    if (c) {
7      return 0; // modified
     }
8    s4;
9    s5;
   }
```

Statement	test 1	test 2	test 3
1	X	X	X
2	X	X	X
3		X(M)	X(M)
4	X		
5	X		
6	X	X	X
7		X(M)	
8	X		X
9	X		X

Modifications	Total	Additional
No	t1,t3,t2	t1,t2,t3
Yes	t2,t3,t1	t2,t1,t3

Figure 2.3. *Comparison of test case prioritization strategies, see explanation in text (inspired by Rothermel et al. [36]).*

a decimal number between 0 and 1 where a higher value indicates a better ordering, e.g. that the fault finding tests were closer to beginning of the test suite. APFD consists of TF_i, n and m, where n is the number of test cases, m is the number of faults revealed by test suite T, and TF_i is the index of the first test case that reveals the ith fault.

$$APFD = 1 - \frac{TF_1 + ... + TF_m}{nm} + \frac{1}{2n} \tag{2.10}$$

Yoo and Harman [44] point out, that every fault and the test cases that revealed them must be known in order to calculate APFD. If this information was known beforehand, we would not need to prioritize anything. APFD can be used only for evaluation purposes after the prioritization has been carried out to measure its performance.

Using the coverage information as a surrogate for early fault detection has been studied in the past. Di Nardo et al. [11] compared the APFD values of four different coverage prioritization strategies, namely *total coverage*, *additional coverage*, *total coverage of modified code* and *additional coverage of modified code*. We will go through these strategies next, but introduce the Figure 2.3 shortly before that. The figure compares each of the strategies in a concise format. The figure shows a source code snippet, three test cases that cover the source code and a table that compares the orderings of the four strategies. *X* in the right-most table indicates that the test case covers the statement, and *M* inside the parentheses denotes that a statement has been modified. For example, *test 2* covers statements 1,2,3,6 and 7. Two of the statements, 3 and 7, were modified from the previous code.

Total coverage is the simplest of the strategies. It orders the test suite descending by a chosen coverage criterion [11][36]. The coverage criterion addresses the resolution of

the coverage information. This includes branch, statement, line and function coverage, to name a few [11][44]. The test case with the highest number of statements, lines or functions covered is executed first, and the least covering test case is executed last. The order is therefore *t1,t3,t2* in Figure 2.3. If two tests have similar coverage, the order can be chosen randomly [36]. Total coverage strategies belong to greedy algorithms [44].

Total coverage of modified code is a variation of total coverage, that considers only the changes introduced in the modified version of the software P'. Figure 2.3 presents the changed statements as *X(M)*. The test cases that exercise most of the modified code, are prioritized as first, and the least modification covering tests are prioritized as last. The order is therefore *t2,t3,t1* in Figure 2.3. The modification aware strategies generally assume that faults are found in the changed parts of the system.

Additional coverage strategy orders test cases descending by the coverage criterion. The difference to total coverage is that overlapping coverage is avoided - the test case that covers the greatest number of uncovered code is prioritized next [11]. For example, in figure 2.3 the order is *t1,t2,t3*, because initially *test 1* has the most comprehensive coverage. After removing the overlapping coverage in *test 2* and *test 3*, *test 2* has the highest coverage over uncovered code, and is therefore prioritized second. The additional coverage approaches belong to additional greedy algorithms [44].

Additional coverage of modified code is a variation of additional coverage, where only the changes introduced in the modified version of the software P' are considered. For example, in Figure 2.3 the order is given as *t2,t1,t3*. Initially, *test 2* covers most of the modifications and is therefore prioritized first. When the overlapping coverage in *test 1* and *test 3* are removed, there are no test cases left that covers yet uncovered and modified code. In this situation, the additional coverage approaches need a fallback strategy, e.g. total coverage. *Test 1* is selected next because it covers most of the statements according to the total coverage strategy.

Both of the additional coverage approaches need a fallback strategy when the saturation point is encountered. With saturation point, we mean the point during the prioritization where the selected test cases exercise the software so completely, that no other test case can increase test suite coverage. Rothermel et al. [35] used the total coverage strategy to prioritize the remaining test cases, similarly what was done in the example of the Figure 2.3. In a later study, Rothermel et al. [36] reset test cases to their initial coverage values, and then reapplied additional coverage strategy excluding the test cases that were already selected.

In their study, Di Nardo et al. [11] concluded that additional coverage strategies perform better than total coverage strategies. On the other hand, Rothermel et al. [36] did not find evidence that additional strategies perform better than total coverage, but their results were more mixed. Di Nardo et al. also concluded, that the modification information does not enhance test case prioritization, and therefore there is no reason to use the modification

aware prioritization strategies over the non-modification aware strategies. We made a similar notion in our case study as well.

2.6 Dependency coverage

When a modification is made to the software, the modification can break functionality in the modified module, but also in the modules that depend on the modified module [45]. This propagation of the modification can be troublesome as faults can show up in surprising components or sub-systems [45]. Executing the whole test suite for even a small change in the software is therefore justified to reveal the maximum number of regressions, but this is not always possible, due to restrictive time requirements, for example.

A rather safe way to reduce the number of tests, but to still to reveal faults in the dependent modules, is to recognize tests that exercise the modified modules and their dependents and execute them. Figure 2.4 presents a dependency graph with three test cases and six modules. A change in module *m1* would tag test *t1* affected. A change in module *m5* would tag tests $\{t2, t3\}$ affected. A combination of changes in modules $\{m5, m3\}$ would tag all test cases affected. Executing only the affected tests in the test suite possibly brings a reduction in the test suite size, but at the same time ensures that all dependent modules are tested. From now on, we call such TCS technique as *transitive dependency selection*.

Yoo et al. [45] applied multi objective search-based test suite selection in Google's test environment. They used information such as fault history, test execution time and dependency coverage to select and prioritize test cases from the test suite. Dependency coverage measures how big portion of the transitively affected modules are exercised by a reduced test suite. In other words, dependency coverage indicates the capability of a reduced test suite to exercise the affected modules and possibly reveal regression faults. Selecting such a set of tests that maximize the dependency coverage is therefore ideal. Yoo et al. formulated dependency coverage as follows:

$$\delta cov(T) = \frac{|\{m_i \in M : \exists t_j \in T \text{ s.t. } m_i \text{ is reached from } t_j\}|}{|M|} \tag{2.11}$$

where T is a subset of the full test suite $\mathscr{T}$ and M is the set of all transitively affected modules $\{m_1, m_2, ..., m_n\}$ for a modification. A test case t_j reaches module m_i if m_i is transitively dependant on t_j. For example, if we modify module *m3* in Figure 2.4, M equals to $\{m1, m2, m3, m4\}$, because these modules depend on *m3*. If we then choose a reduced test suite $T = \{t1, t2\}$, $\delta cov(T)$ equals to $\frac{|\{m1,m3\}|}{|\{m1,m2,m3,m4\}|} = \frac{2}{4} = 0.5$, indicating that the reduced test suite exercises half of the affected modules.

Yoo et al. [45] separates test and use dependencies, as pointed out in Figure 2.4. Test dependencies are between the test cases and the modules that they test. Use dependencies

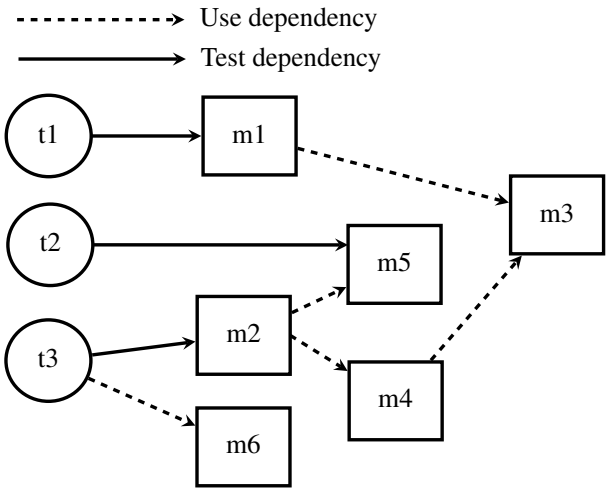

Figure 2.4. *An example of test cases and their dependencies (after Yoo et al. [45]).*

are between any two modules, respectively. A use dependency can be between a test and a functional module too if the test does not do assertations against the module. Such dependency is shown between $t3$ and $m6$ in Figure 2.4. In our case study, we are interested only in the use dependencies between the test cases and modules, for example helper packages and collections of utility functions.

2.7 Machine learning

Machine learning is a multidisciplinary field, that combines ”statistics, artificial intelligence, philosophy, information theory, biology, cognitive science, computational complexity, and control theory” [29]. The gist of machine learning is a piece of software, that is capable of improving its performance in a set of tasks, based on experience [29]. Any software that is able to use its previous knowledge to perform better at something without reprogramming the software, can be considered to realize machine learning.

Machine learning problems can be roughly categorized in *supervised*, *semi-supervised*, *unsupervised* and *reinforcement learning* problems [1]. In this case study, we are mostly interested in supervised learning, and especially in its sub-problem, *classification*. Supervised classification tasks are generally given *training data* and categorical *labels*, and the goal is to learn a mapping function from the training data to the labels [1]. The better the mapping function is, the more accurate results a machine learning *model* produces. For example, in an image classification task, the goal can be to recognize a certain animal in an image. The training data would consist of images of three animals, and the respective labels for the images would be one of $\{cat(0), dog(1), horse(2)\}$. Given the training data and the labels, the machine learning model would learn and modify its *parameters* so, that every time an image of a cat is fed to the model, the output of the model would equal to 0. In an ideal case, the machine learning model would not only have learned to classify images it has already seen in the training data but any arbitrary image of a cat or a dog. We say that a machine learning model is able to *generalize* well, if it is able to correctly

classify images outside from the training data [1]. On the contrary, if the model is only able to correctly classify the training samples and not any arbitrary image of a cat, it is possible that our model is *overfitting* due to a low number of training samples or too complex model [1].

In our case study, we classify test cases into categories of $\{fail(0), pass(1)\}$ using machine learning, similarly to the image classification task. We use a variety of ML models, ranging from neural networks and gradient boosting technique to simpler models such as logistic regression and naive bayes. These models work differently, i.e. their underlying algorithms are different, but they perform the same task: classification. As the models provide similar service, the implementation details can be hidden. The *scikit-learn* library [31] takes advantage of this and provides a similar interface to multiple machine learning models. In addition to *scikit-learn*, we use *xgboost* library that provides a gradient boosting technique for classification.

Online and incremental learning

Online learning is a branch of machine learning, where the training data is fed to the machine learning model iteratively instead of in one batch. The benefit of doing this is that the training data does not have to fit into the computer memory all at once, in case the training data is very large. The training data can be split into smaller chunks, and the model can be trained iteratively using the smaller chunks that fit into memory.

Another benefit of online learning is, that sometimes the full training data is not available, but is achieved over time. For example, in test case selection or prioritization, an execution of a test case increases training data by one sample. In these situations, it is useful for being able to update the current machine learning model on the fly.

In our case study, we make the same distinction between online learning and incremental learning as Saffari et al. [37]. Incremental learning is similar to online learning but is allowed to store a data sample for later use, whereas online learning has to discard the sample when the model is updated. Following this distinction, we apply incremental learning instead of online learning in our case study. We do this by accumulating the training data as more data samples become available after test executions and re-train the model with full training data in every iteration.

2.8 Related work

Spieker et al. [40] used reinforcement learning and multilayer perceptron to predict failing test cases based on test history. Their technique actualized both test case selection and prioritization in test suites. The idea was to 1) prioritize the test suite T, and 2) repeat selecting the topmost test from T as long as the summed duration of the selected tests go under a time threshold M. Such a technique can be considered as priority-based test

case selection. Spieker et al. used the normalized average percentage of faults detected (NAPFD) to measure the performance of their technique. Spieker et al. concluded that their technique needs approximately 60 CI cycles to perform equally or better compared to comparison techniques. The comparison techniques were a random technique, which ordered test cases randomly, a sorting technique, which ordered recently failed test cases with higher priority, and a weighing technique, which ordered test cases by a weighted sum of the test features. Spieker et al. were the first to apply reinforcement learning and possibly online learning in the fields of TCS and TCP.

Busjaeger and Xie [10] used supervised learning and pointwise ranking to prioritize test cases. They extracted testing results from an automation system worth three months, processed the data and divided it into training and test datasets. They then trained an SVM model with the training dataset and applied TCP to the test dataset. Busjaeger and Xie showed, that their supervised ML prioritization technique outperformed all comparison techniques. The comparison techniques included a random prioritization technique, a test history technique, and a coverage prioritization technique, among others. Furthermore, Busjaeger and Xie reported different recall values when a number of test cases are selected from the top of the prioritized list. Selecting 3% of the top tests from the prioritized test suite they achieved 75% recall. 3% selection size is the same as 97% test suite reduction.

Di Nardo et al. [11] applied TCS and TCP in an industrial system with real regression faults. They measured reductions in test suite sizes and fault detection effectivenesses with their coverage-based TCS techniques. Di Nardo et al. were barely able to reduce test suite sizes at all. The maximum test suite reduction they recorded, was 2%. Because of the small reductions, fault detection effectiveness was not compromised and the reduced test suites revealed the same number of faults as the original test suites. Di Nardo et al. discussed, that the small reductions in test suite sizes were likely caused by modifications to the core components of the software. Such parts were thought to be covered by a multitude of test cases. Additionally, as Di Nardo et al. examined only four different software versions, the modifications between subsequent versions were arguably large. Considering TCP, Di Nardo et al. showed that techniques based on additional coverage with fine-grained coverage perform significantly better than total coverage techniques. Furthermore, using modification information did not enhance coverage-based TCP techniques.

Beszédes et al. [6] used priority-based TCS to reduce test suite size in the WebKit web browser engine. In their initial experiments, they selected every test case that covered the modified procedures in the software, or that had failed previously. Using this initial selection, they witnessed a test suite reduction of $79,43\%$ with $95,08\%$ recall on average. In their study, Beszédes et al. used the term ”inclusiveness” instead of recall. They define: ”It is the ratio of the failing test cases included in the selection relative to the total number of failing test cases when executing the complete test suite.” This is the same as recall, as pointed out in section 2.3. Therefore, when Beszédes et al. mention ”inclusiveness”, we substitute it with ”recall”. When Beszédes et al. applied their selection technique in an actual live system where the setup was a bit more realistic, they witnessed a test

suite reduction of 51% with 75,38% recall on average. Beszédes et al. extended their selection technique with an extra prioritization step. The prioritization was based on coverage information. With this extra step, Beszédes et al. were able to further reduce the selection size. Using such a technique, they showed test suite reduction of over 90% with half the recall compared to the non-prioritized test suite. We interpret, that the recall was therefore approximately 38%. Comparing this result to result by Busjaeger and Xie [10], the ML-based TCS technique seems to have superior performance.

Harrold et al. [19] experienced fluctuating test suite reductions with their code-based regression-test-selection technique. Their TCS technique relied on code coverage information. Harrold et al. recorded test suite reductions from 0% to almost 100%. They discussed, that the large reductions were due to small modifications in the software, where only a few methods covered by a few tests were changed. Harrold et al. did not analyze thoroughly the reasons behind the small reductions but mentioned that the location of a change can affect test suite reduction. For example, ”a change in a startup code of a software causes each test case to be selected.” As Harrold et al. applied their technique over four different softwares with less than eleven software versions, it is possible that the modifications between two consecutive versions were still quite large. Applying TCS in such versions can bring no reduction in test suite size.

Gligoric et al. [17] used dynamic dependency tracking from tests to files to reduce the number of tests in the test suite. Their tool, ”Ekstazi”, can track any changes in files that are dependent on the tests, and execute only part of the test suite that is relevant for a set of file changes. The tool is capable of tracking source code files, but also configuration files. The tool monitors the execution of tests running on JVM and collects the accessed files using bytecode instrumentation and listening to all standard Java library methods that might open a file. After the collection of the dependent files is done, the tool can select a subset of tests to be executed for any change made in the dependent files. Gligoric et al. report, that their tool is capable to reduce end-to-end testing time by 32%.

Yoo et al. [45] used dependency coverage (equation 2.11) among other features to select and prioritize test suites. The optimization technique by Yoo et al. balanced three competing objectives: dependency coverage maximization, historical fault detection maximization and execution time minimization. Yoo et al. reported an average test suite reduction of 68% with their technique.

3. METHODOLOGY

We apply test case selection and prioritization techniques to existing software from the space industry to explore ways on how to enhance slow validation testing at *SSF*. We use a number of techniques in our case study and compare their performances. The selection techniques are introduced in section 3.1 and prioritization techniques in section 3.2. In section 3.3, we describe the data that each selection and prioritization technique has access to. In section 3.4, we describe how we select hyperparameters for the machine learning models. Finally, in section 3.5, we describe our case study setup and how we carry out the experiments.

3.1 Test case selection techniques

We chose eight test case selection techniques for comparison, where four are based on heuristics and four on machine learning. The heuristics are namely *Random*, *Coverage*, *Coverage(H)* and *Coverage(PH)*. The machine learning techniques are namely *RandomForest*, *RandomForest(U)*, *LogReg* and *XGBoost*. The Table 3.1 describes and summarizes each technique.

Half of the techniques include setting an upper limit for selection size. We set the limit to 2%, which guarantees at least 98% test suite reduction. There is no special reason why we set the limit to exactly 2%, but it was rather an arbitrary choice. We estimated, that 2% selection size reduces our test suite to approximately 10 test cases, which was thought to be reasonably quick to execute. An average test case takes 106 seconds to execute, and therefore 10 test cases take around 18 minutes to execute without test parallelization. For us, this is a more or less acceptable time waiting for the test results. For other projects where the committing frequency is higher, 18 minutes could have still been unacceptable.

The *Coverage(PH)* technique guarantees a 98% test suite reduction. It applies a similar selection technique to *Coverage(H)*, and selects every test case that fails in previous commit or covers changed parts in the SUT. Similarly to Beszédes et al.[6], a further prioritization step is taken if the optimal selection size is exceeded. The prioritization is based on a combined surrogate of test history and coverage, more precisely the average $\frac{M+FR+(LP-1)}{3}$, where M is *modification coverage*, FR is *failure rate* and LP is *latest pass* (see 3.3). The initially selected tests are ordered descending by this value, and only top tests from the ordered list are executed so that 2% selection size is satisfied.

RandomForest, *LogReg* and *XGBoost* also guarantee 98% test suite reduction. These techniques use the class probabilities, *predict_proba()* in *scikit-learn*, to sort the test cases descending by likelihood to fail when they exceed the initial selection size of 2%. They

Technique	Description	Selection size
Random	Select randomly n test cases from the test suite. n is a random number from 0 to $\|T\|$.	0 to $\|T\|$
Coverage	Select every test case that covers a modified statement.	0 to $\|T\|$
Coverage(H)	Select every test case that either 1) covers a modified statement or 2) failed in previous iteration.	0 to $\|T\|$
Coverage(PH)	Select every test case that either 1) covers a modified statement or 2) failed in previous iteration. Furthermore, if the selection size is greater than 2%, prioritize the selected tests and select n top test cases until 2% limit is satisfied. The prioritization step calculates the average of test history and coverage, sorts the test cases descending by this value and selects n top test cases from the sorted list.	0 to $0.02 \times \|T\|$
RandomForest	Select every test case that is predicted failing using random forest classifier from *scikit-learn*. The random forest implementation follows the Breiman's implementation [9]. Furthermore, if selection size is greater than 2% or less than 2, prioritize the test suite T using class probabilities and select 2% of the most promising tests.	2 to $0.02 \times \|T\|$
RandomForest(U)	Select every test case that is predicted failing using random forest classifier from *scikit-learn*. The random forest implementation follows the Breiman's implementation [9]. Furthermore, if selection size is less than 2, prioritize the test suite T using class probabilities and select 2% of the most promising tests. (U) denotes unlimited, i.e. this technique has no upper selection size limit.	2 to $\|T\|$
LogReg	Select every test case that is predicted failing using logistic regression classifier from *scikit-learn*. Furthermore, if selection size is greater than 2% or less than 2, prioritize the test suite T using class probabilities and select 2% of the most promising tests.	2 to $0.02 \times \|T\|$
XGBoost	Select every test case that is predicted failing using gradient boosting technique (XGBClassifier) from *xgboost*-library. Furthermore, if selection size is greater than 2% or less than 2, prioritize the test suite T using class probabilities and select 2% of the most promising tests.	2 to $0.02 \times \|T\|$

Table 3.1. *Test case selection techniques. The first four techniques are based on heuristics and the last four on machine learning. The machine learning techniques apply binary classification over the test case samples, and categorize the samples into bins of passing and failing. Part of the techniques guarantee* 98% *test suite reduction, namely Coverage(PH), RandomForest, LogReg and XGBoost.*

select a number of test cases from the prioritized list until 2% selection size is satisfied. Additionally, if the selection size was initially 0 or 1, these techniques apply the same protocol: prioritize and select 2% of the test cases. We did it this way because we found that in the beginning, the incremental machine learning techniques had the tendency to select only a small number of test cases, even 0 during many early commits.

Four techniques do not have upper limits for selection sizes. These are *Random*, *Coverage*, *Coverage(H)* and *RandomForest(U)*. They can select any number of tests to be executed. *RandomForest(U)* has a lower limit, i.e. it selects at least two test cases.

We evaluate the performances of each test case selection technique with MCC-score, confusion matrix, recall, and test suite reduction. We differentiate the well and poorly

performing techniques with MCC-score (equation 2.7). We use test suite reduction and recall to link and compare the results of this case study to previous studies. Confusion matrices provide additional details of the techniques, e.g. the number of false negatives, false positives, true negatives, and true positives.

3.2 Test case prioritization techniques

We evaluate and compare the performance of ten different test case prioritization techniques, where five are based on heuristics and five on machine learning. The heuristics are namely *Random*, *Coverage(T)*, *Coverage(A)* and *Coverage(AM)*. The machine learning techniques are namely *RandomForest*, *MLP*, *XGBoost*, *NaiveBayes* and *LogReg*. The Table 3.2 describes and summarizes each technique.

Technique	Description
Random	Prioritize test cases randomly.
Coverage(T)	Prioritize test cases according to total coverage strategy. Order test cases descending by *statement coverage* feature (see 3.3).
Coverage(TM)	Prioritize test cases according to total coverage of modifications strategy. Order test cases descending by *modification coverage* feature (see 3.3).
Coverage(A)	Prioritize test cases according to additional coverage strategy. Order test cases descending by number of unique covered statements. Test case that exercises the maximum number of yet undiscovered statements is prioritized next. Instead of the features in 3.3, this strategy needs to access the full coverage information for every test case.
Coverage(AM)	Prioritize test cases according to additional coverage of modifications strategy. Order test cases descending by number of unique covered and modified statements. Test case that exercises the maximum number of yet undiscovered and modified statements is prioritized next. Instead of the features in 3.3, this strategy needs to access the full coverage information for every test case. Needs also the modification information (e.g. git diff).
RandomForest	Prioritize test cases with random forest classifier from *scikit-learn* according to Breiman's implementation [9]. Sort test cases descending by class probabilities with *predict_proba()*.
MLP	Prioritize test cases with multi-layer perceptron classifier from *scikit-learn*. Sort test cases descending by class probabilities with *predict_proba()*.
XGBoost	Prioritize test cases with XGBClassifier from *xgboost*-library. Sort test cases descending by class probabilities with *predict_proba()*.
NaiveBayes	Prioritize test cases with gaussian naive bayes classifier from *scikit-learn*. Sort test cases descending by class probabilities with *predict_proba()*.
LogReg	Prioritize test cases with logistic regression classifier from *scikit-learn*. Sort test cases descending by class probabilities with *predict_proba()*.

Table 3.2. *Test case prioritization techniques. The first five techniques are based on heuristics and the last five on machine learning. The machine learning techniques apply pointwise ranking on the test case samples using class probabilities.*

The ML-based TCP techniques use class probabilities to prioritize test cases. The class probabilities are probability estimates for a test to fail and pass. More closely, given a test t, class probabilities is a tuple of $\{p1, p2\}$, where $p1$ is the probability estimate of t to fail and $p2$ is a probability estimate for t to pass. We sort the test cases descending by $p1$. In machine-learned ranking, such an approach is called pointwise ranking [26]. The other approaches are pairwise and listwise ranking, which usually outperform pointwise ranking [26]. In *scikit-learn*, we can get class probabilities with function *predict_proba()*.

If total coverage approaches find two equally important test cases, e.g. same statement coverage for two or more tests, Rothermel et al. [36] state that additional rules are necessary to order the equal test cases. We follow this suggestion and use *duration* as a secondary sorting rule for *Coverage(T)*, and *statement coverage* and *duration* for *Coverage(TM)*, respectively. Furthermore, in the literature, the additional coverage approaches also have used fallback strategies, when all software statements have been exercised at least once by a previous test case. For example, Rothermel et al. [35] used total coverage as a fallback strategy to additional strategies. We instead, do a similar thing as we do with total approaches: We use additional sorting rules. The secondary sorting rule for *Coverage(A)* is *duration*, and for *Coverage(AM)* the secondary rule is *statement coverage* and third rule is *duration*.

We measure the performance of each test case prioritization technique with weighted average percentage of faults detected (APFD), like in previous studies [11][35][36][24]. We differentiate the well and poorly performing techniques with APFD.

3.3 Test cases as feature vectors

Before we can apply machine learning techniques on test cases, we have to represent them as feature vectors. Feature vector describes an object and its characteristics as a vector of features. A feature can be numerical or categorical, such as a real number or a string. In our case study, all features are numerical. More closely, we use a total of seven features to represent test cases as feature vectors. Namely these features are *statement coverage*, *modification coverage*, *similarity score*, *duration*, *failure rate*, *latest pass* and *history length*. The features are similar to features used in the study by Busjaeger and Xie [10] but there are some differences. In the following section, we describe how we attain these features. But for now, let's not pay attention to where these features are coming from, but instead assume, that they are given to us.

Statement coverage is a floating-point number with a closed interval between 0 and 1. It is the total percent of statements (or lines) covered by a test case. For example, 1 would mean that a test case covers every line of the software and 0 would mean that not a single line is covered by the test case. The average statement coverage of all samples was approximately 0.39. The high value is explained by the nature of the validation tests, which can be enormous and exercise multiple features and requirements in an end-to-end manner. Unit tests, on the other hand, would arguably have a lower value. The statement coverage is updated every time the test case is executed and remains unchanged until the test case is re-executed. It is worth noticing, that even though statement coverage is said to be used in this thesis, in reality, it is the line coverage, and not statement coverage. The reason for this lies in the *gcov*-tool, which does not segregate lines and statements but interprets them the same.

Modification coverage is similar to statement coverage and has similar properties, but the coverage is calculated over the modified lines of a commit instead of all software code

lines. More closely, the modification coverage is calculated as $\frac{|C_t \cap M|}{|M|}$ where C_t is coverage of the test case and M is the set of modified lines. The test coverage, C_t, is produced by *gcov*-tool. We use *git diff* command to find M. Version control systems are often able to provide information about added lines too, but it is not easy to conclude whether they are covered or not. The modification coverage is different from any other feature because it combines information from history and the present. The full coverage of previous test executions is needed to deduce coverage over the modified lines, e.g. $|C_t \cap M|$. Therefore, the full coverage produced by *gcov*-tool needs to be persisted and transferred between any two commits.

Similarity score is the only feature that can be calculated dynamically without any knowledge of history. It is a similarity measure between a code change (git commit) and a test case, where a higher value means that certain keywords occur more often in both texts suggesting a higher similarity. The similarity score is calculated with TF-IDF transformation and cosine similarity [27]. TF-IDF ensures, that the keywords that occur in many test cases are discarded. The similarity score is a floating-point number with a closed interval between -1 and 1. Busjaeger and Xie [10] used the similarity score, but in addition, they first parsed the source code into abstract syntax tree and extracted identifiers such as method names, classes, and variables to build a content index. This case study does not use as thorough pre-processing but instead removes special characters from the test source codes before applying TF-IDF and cosine similarity.

Duration is test execution time of the test case in seconds. The average duration was approximately 106 seconds, shortest 10.51 seconds and longest 2160.24 seconds. Duration is updated every time the test case is executed.

Failure rate is a floating-point number with closed interval between 0 and 1 and it is calculated by $\frac{T_f}{T_p+T_f}$ where T_f is the total number of failures and T_p is total number of passes of single test case. Every test execution updates this value because either T_f or T_p is incremented.

Latest pass is a left-closed and right-unbounded discrete value from 1 to infinity. It is the number of failing test executions that precedes a passing test execution. Despite its' name, the latest pass is updated only if a test is executed as part of the test suite. Every failing test execution increments this value by one and passing execution resets the value back to 1. The initial value 1 is set because of the assumption that initially, every test case is passing.

History length is a left-closed and right-unbounded discrete value from 1 to infinity which denotes the number of executions for the test case. The initial value is 1 and each test execution increments the value by one.

3.4 Select hyperparameters for the ML models

Many machine learning models require setting a certain number of hyperparameters before they can be used. For example, the random forest model requires the user to specify

the number of decision trees in the forest and the maximum depth for each tree. These hyperparameters are usually set manually by the user, and they are not ”learned” in a way like the parameters of the model are. Finding a good set of hyperparameters can be hard, but there are a some techniques that can help. One of these techniques is a grid search. Grid search requires a set of candidate values for every hyperparameter. From the given candidate values, it tries to find the best combination of hyperparameters that produce the highest score. It trains the machine learning model with every combination and reports the set of hyperparameters that produce the highest score.

To search for the best hyperparameters for our binary classification machine learning models, we use grid search together with stratified 5-fold cross-validation. We use stratified cross-validation to make sure, that every fold has failing tests in them, because our data is highly unbalanced. We first collect all the samples and their respective labels for every test case, e.g. feature vectors consisting of statement coverage, modification coverage, similarity score, duration, failure rate, latest pass and history length and the label of 1 (passing) or 0 (failing). We then apply the *GridSearchCV* in *scikit-learn* over a small subset of these samples and labels using MCC as the score to find the best hyperparameters to classify unseen data. We select randomly 15% of the samples from the full dataset, because we want to minimize the data leakage from unseen data to later incremental learning. Also, we want our models to be able to maximize MCC as early as possible during incremental learning, and therefore there is no point using the whole dataset, as this would provide maximum performance only in the last commit. We applied the hyperparameter selection and obtained following hyperparameters for each classifier:

- Random forest
  - n_estimators: 150
  - max_depth: 7
  - criterion: gini
  - bootstrap: False
  - min_samples_split: 5
  - min_samples_leaf: 1
- XGBoost classifier (from *xgboost*-library)
  - n_estimators: 125
  - booster: gbtree
  - max_depth: 7
  - learning_rate: 0.75
  - subsample: 1
  - reg_alpha: 0
  - reg_lambda: 1
  - min_child_weight: 0
  - max_delta_step: 0

  - gamma: 0.1
  - colsample_bylevel: 0.75
  - colsample_bytree: 0.5
  - importance_type: gain
- Gaussian naive bayes (no hyperparameters)
- Logistic regression
  - penalty: l2
  - solver: liblinear
  - tol: 0.1
  - C: 0.2
  - max_iter: 1000
  - class_weight: balanced
- Multi layer perceptron
  - hidden_layer_sizes: (8,)
  - activation: relu
  - solver: adam
  - alpha: 1e-03
  - learning_rate_init: 0.01
  - tol: 0.001
  - max_iter: 1750
  - shuffle: True
  - early_stopping: False

We are aware, that by simply iterating through every test and commit, we use coverage and test history that would not be available yet in test case selection. We thought this to have a small effect on the hyperparameters, so we did not consider this being a problem. Another aspect is that we did not apply grid search separately for test case prioritization, but when we found good hyperparameters for test case selection, we used the same hyperparameters for prioritization.

3.5 Case study setup

Our task is to apply test case selection and prioritization in an existing software from the space industry to see whether we can enhance the slow testing process. Speeding up testing is important because it facilitates continuous integration and helps the developers to react faster to faults and failing tests. We apply TCS and TCP techniques from sections 3.1 and 3.2 to the software to gain insight which techniques are the most helpful for us in this sense.

We could have carried out the case study by checking out in the oldest commit in the version control history, and work our way up towards the newest commit while applying test case selection and prioritization techniques and recording APFD and MCC results on the fly. However, this approach would have given us little control over the setup, and we would have to repeat the process multiple times for different test selection techniques. Repeating the process multiple times was out of the question because the full test suite execution took us 17 hours. Instead, we wanted to collect coverage information, test verdicts and test durations only once for every test in every commit, and then apply test selection and prioritization to the collected data ”offline” to save time.

Additionally, we want to study the transitive dependency selection in our case study. We try to find out how big test suite reductions transitive dependency selection can produce among the test modifying commits. We, therefore, separate the case study in three phases: 1) data collection, 2) application of TCS/TCP techniques and 3) experimenting transitive dependency selection. The first two phases are connected to each other, and the third phase is its own independent experiment. In the next subsection, we describe how we collect the data. After this, we describe how we apply TCS and TCP techniques. Finally, in subsection 3.5.3, we describe how we carry out the transitive dependency selection experiment.

3.5.1 1st phase: Data collection

We collected statement coverage, test verdicts and test durations for every test case for as many commits as possible from the software. Because executing the full test suite took 17 hours, we knew that the data collection process was going to take a long time. In order to reduce the time, we decided to separate how source modifying commits and test modifying commits are handled. When a commit modifies only *test/* directory and not *src/* directory at all, we use transitive dependency selection to select only tests cases that are transitively affected by the modification. This indeed reduced the number of tests we had to execute during the data collection and made the data collection faster but also complicated our case study setup. We suggest another approach in the future work 5.4 how test case selection and prioritization can be used in projects, where the repository consists of both tests and source code. Our data collection algorithm is the following:

1. Checkout newest commit
2. Repeat:
   (a) If current commit has src/* modifications:
      i. Execute test suite
   (b) Else if current commit has test/* modifications:
      i. Find modified tests through transitive dependency selection (see 2.6)
      ii. Execute modified tests
   (c) Save executed test verdicts, coverage and durations
   (d) Checkout previous commit

The output of the algorithm above is an ordered set of tuples *D = {commit, tests}*, where *commit* is a commit's checksum and *tests* is a set of tuples *{verdict, coverage, duration}*. *verdict* is the output of a test: pass or fail, *coverage* is the full gcov-coverage for the test and *duration* is the test length in seconds. The first phase of the case study ends here. In the second phase, we use this data to perform test case selection and prioritization. Before that, we will describe the characteristics of our data, and how we preprocessed the version control history.

The version control history contained multiple instances of test/* modifying commits one after the other. We squashed these commits with *git rebase -i* command to save time during the data collection. This was done as a preprocessing step before running the data collection algorithm. Every commit that had no source or test modifications was also removed since they had no effect on the functionality of the software.

The step 2ai, full test suite execution, lasted 17 hours on average. The algorithm was continuously being executed for approximately two months for the preprocessed version control history. 87 commits ended up in the dataset, where 45 commits had only source code modifications, 17 commits had both test and source code modifications and 25 commits had only test modifications. Unfortunately, the test suite contained many non-deterministic test cases due to differences in the test environments and we decided to remove these tests from the dataset. As a result, a portion of the commits ended up having no faults. Even though part of the commits had no longer failing tests, we did not remove these commits from the dataset. We decided to keep them because we want our setup to correspond to a real environment as closely as possible. In the second phase, we apply test case selection and prioritization techniques in these commits, but we do not calculate performance scores in them (APFD, MCC, TSR, recall, etc).

The characteristics of the collected dataset is shown in Table 3.3. The dataset contained 36 source modifying commits, 14 both source and test modifying commits and 11 test modifying commits that included at least one failing test case. The 36 source commits contained 142 failing and 18000 passing tests. Out of the 142 failing tests, 32 were normal failures, and 110 either failed to build or had runtime errors. Note, that the build or execution failures in the Table 3.3 are test cases that had passed at least once before. We had to remove every test case that was recently added and had build or execution failures because it was impossible to gather coverage information for them. As soon as the removed tests passed again in the following commits, they were added back to the test suite. The oldest commit in the dataset did not luckily contain any failing tests after the non-deterministic tests were removed.

Surprisingly, many of the test cases failed because of build or execution errors. The reason behind this was not thoroughly studied, but it could possibly relate to differences between the test environments used in this thesis and the real one. It is also possible, that the developers were aware of these build failures all along, and they had no intention to fix them.

Modifications	Commits	Commits with at least one failing test case	Failing tests	Normal failures	Build or execution failures	Passing tests
Source code	45	36	142	32	110	22590
Source and test code	17	14	124	66	58	8513
Test code	25	11	119	39	80	7109

Table 3.3. *Charasteristics of the data used in this thesis.*

Figure 3.1 illustrates the version control history with commit specific test execution information. The shapes connected with lines represent commits in the version control history, where the shape in bottom-left corner equals to the oldest commit. The commits are divided in three types, source code modifying, test modifying and both source and test modifying commits. There are no test modifying commits next to each other, because consecutive test commits were squashed. The number on the left side of the commit is the number of tests executed, excluding tests that were non-deterministic or newly introduced and had build failures. For source and source&test modifying commits the number equals to the size of the test suite. The number inside the circle shape denotes the *n*th measurement point, e.g. measurement of APFD, test suite reduction or recall. Out of the 87 commits 35 were used as measurement points. Generally, all source modifying commits were measured, unless they had zero test failures. The star (*) inside the first commit denotes the initial commit, where code coverage, test durations and test history were initialized. The following commit (T) refers to *training*, where the machine learning models for test prioritization and selection were initially trained with the full test suite information. Unfortunately, the training commit contained only one failing test, and therefore the initial training was thin. Figure 3.1 does not display number of test executions and failing tests for every commit. Whenever the numbers are not shown, it implies that the number has same value as the commit before it.

3.5.2 2nd phase: Apply test case prioritization and selection

We apply test case selection and test case prioritization techniques to the collected data in section 3.5.1. In the 1st phase, we collected an ordered set of tuples D = *{commit, tests}*, and in the 2nd phase, we iterate through this data and apply different selection and prioritization techniques to it. The algorithm below presents how we do it. We execute the algorithm for every test case selection technique t and for every tuple $d \in D$:

1. If $d.commit$ has test/* modifications:
   (a) Let T_{tmod} be the tests selected with transitive dependency selection
2. If $d.commit$ has src/* modifications:
   (a) Let T_{smod} be the tests selected with t according to current knowledge C
3. Let $T' = T_{tmod} \cup T_{smod}$

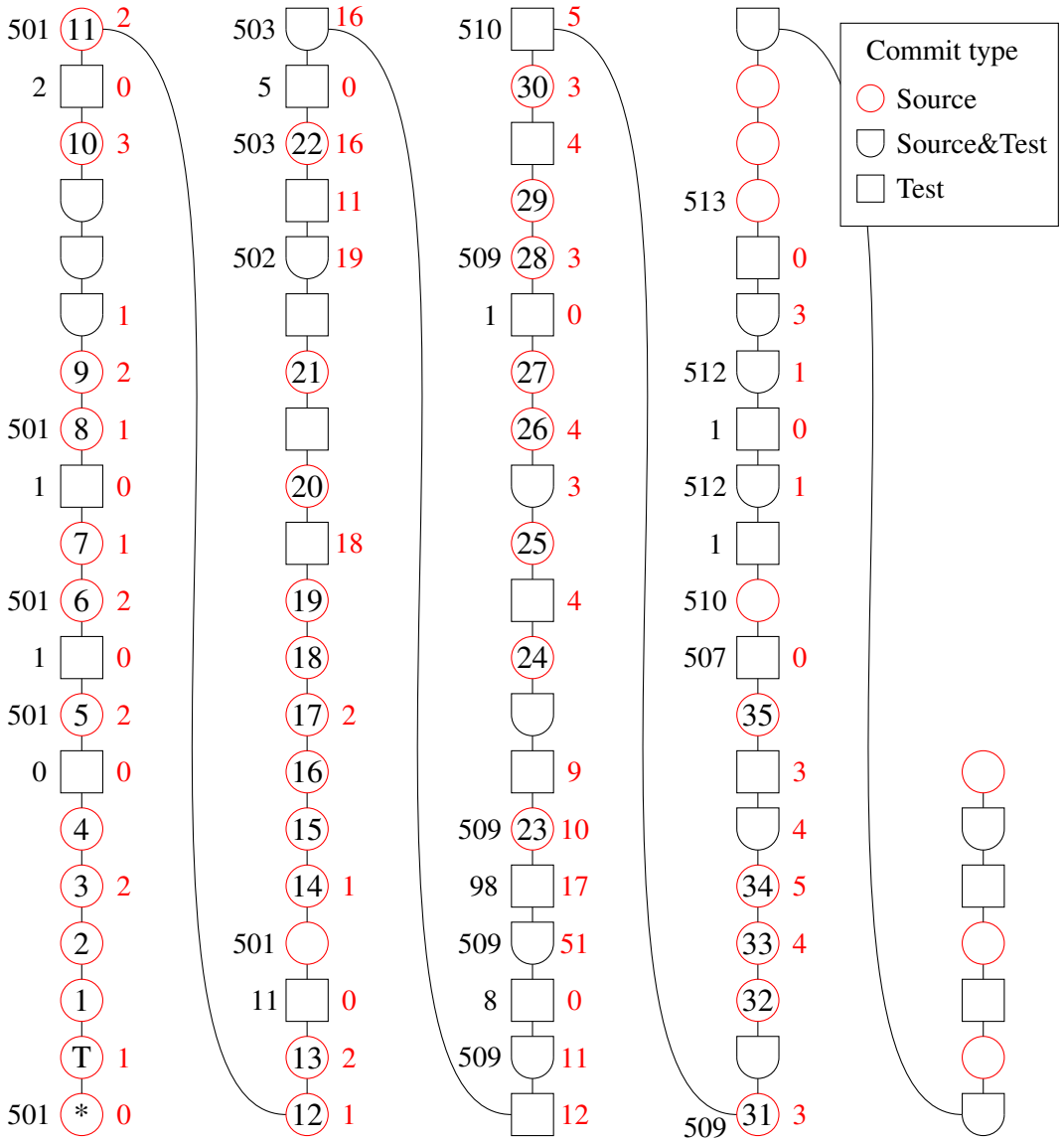

Figure 3.1. *Version control history along with commit specific information, such as number of executed tests on the left, index of source code modifying commit inside the shape and the number of failing tests on the right. The star (*) denotes the initial commit where coverage, test durations and verdicts are collected for the first time. The next commit, tagged as T, is used to train the machine learning models. The indices, e.g. numbers inside the shapes denotes nth measurement point, where test suite reduction, recall and MCC are calculated for test case selection techniques and APFD for test case prioritization techniques. The commit types are separated into source modifying commits (modifies src/*), source&test commits (modifies src/* and test/*) and test commits (modifies test/*).*

4. Simulate the execution of T'
5. Update current knowledge C

The C represents the current knowledge we know about the test cases. This includes the coverage, durations and test verdict histories (history of passes and fails) for every test case. In the first commit, we do not have this information, and therefore one commit is

needed to initialize the test case selection techniques. During the first commit, we collect the initial coverage, durations, and test verdicts. This commit is denoted as * in Figure 3.1. Furthermore, as the machine learning techniques also need initialization, we use the next commit to train the machine learning models with full knowledge provided in $d.tests$. This is denoted as T in Figure 3.1. After these two consecutive commits have passed, we can start applying the algorithm as planned.

In the first step, we apply transitive dependency selection to the $d.tests$, if $d.commit$ type is ”test” or ”source&test”. In the second step, we apply the TCS technique t to $d.tests$ using the current knowledge C. We save the selected tests in T_{smod}. In the third step, we combine the transitively affected tests T_{tmod} and the selected tests T_{smod}. T_{tmod} is empty, if $d.commit$ type is ”source”. T_{smod} is empty, if $d.commit$ type is ”test”, respectively. If $d.commit$ type is ”source&test”, both T_{tmod} and T_{smod} can contain test cases, but not the same test cases.

In the fourth step, we do not have to execute the reduced test suite T', because we already did it in the 1st phase. Instead, we imagine the execution of T', and update our current knowledge about test histories, coverages and durations with the test cases selected from $d.tests$. We discard the information for $d.tests$ that were not selected in T', because, in reality, we would not have their information. In the fifth step, we save and persist the current knowledge C for the next iteration, d_{i+1}.

There are still a few things we have to clarify because the algorithm is abstracted and hides some details. For example, it does not show how we accumulate training data for the machine learning models, or where we re-train the models. Let us explain this next.

In addition to updating C in step 5, we turn the selection T_{smod} into feature vectors. We do this only if $d.commit$ type is ”source”. Using the coverage, duration and verdict we produce the feature vector *{statement coverage, modification coverage, similarity score, duration, failure rate, latest pass}* for every test case. We do this for all tests in T_{smod}, and save them for the next iteration d_{i+1}. We apply this idea in every source commit, and eventually, our training data accumulates and grows larger. The training dataset is a set of $\{T_{smod_1}, ..., T_{smod_{n-1}}, T_{smod_n}\}$, where n is an index of a source commit. During every iteration, we re-train the machine learning model with this training dataset. Accumulating the training dataset like this can become infeasible in the long run, and online learning solutions should be used instead. We have stated this in the future work 5.4.

The last hidden detail about the algorithm is that we calculate MCC, recall and test suite reduction between the steps 4 and 5 if the $d.commit$ is a source commit and the commit has at least one failing test case. If there are no failing tests, the output of MCC is undefined, recall is zero, and test suite reduction would be the only indicator worth measuring. We, therefore, skip measuring performances when the commit has no failing tests. In addition to non-faulty commits, we do not measure performances in ”test&source” commits or ”test” commits either. We consider the problems of this next.

Modern software engineering practices encourage the use of feature branches and merge

requests. The project used in this thesis, however, uses a single branch without merge requests, partly because the project repository was migrated from *svn* to *git*. While it is just an implementation detail whether to apply test case selection over one commit or over a merge request consisting of multiple commits, the types of the changes matter more. When test case selection is applied over merge request, it is more likely, that the changes include both test and source code modifications. If there are many test modifications, there is a risk that transitive dependency selection selects the majority of the tests. Therefore, separating the way how source and test modifications are handled, can be questioned. We realized this issue earlier during the case study, but only figured out a candidate solution to it too late considering the limited time resources given for this thesis. If the dependency coverage (see 2.6) was added as a feature, there would not be a need to separate the handling of source and test modifications. Instead, we could have applied the same test case selection technique t in ”source”, ”source&test” and ”test” commits, and we could have measured the performances regardless of the commit type. We address this issue in section future work 5.4.

The following algorithm presents how we apply test case prioritization to the data collected in section 3.5.2. The algorithm is executed for every test case prioritization technique t and for every $d \in D$:

1. If $d.commit$ has test/* modifications:
   (a) Let T_{tmod} be the tests selected with transitive dependency selection
2. If $d.commit$ has src/* modifications:
   (a) Let $T_p = t.\text{prioritize}(T - T_{tmod}, C)$
3. Let $T' = T_p \cup T_{tmod}$
4. Simulate the execution of T'
5. Save test suite information

where T is the original test suite. In the first step, we select a subset of tests with transitive dependency selection from $d.tests$ if the commit type is ”source&test” or ”test”. In the second step, we exclude T_{tmod} from T, and apply test case prioritization to it if the commit type is ”source&test” or ”source”. T_{tmod} is empty for source commits. The C inside the *prioritize*-function represents the current knowledge of the test cases, i.e. the coverage information from previous test runs, test verdict histories, and durations. In the third step, we combine T_p and T_{tmod}. In the fourth step, the simulate the execution of T', and pull the new information from $d.tests$, namely *coverage*, *duration* and *verdict* for every test case. We do not need to execute T', because the test suite was already executed in the 1st phase. If the $d.commit$ type is ”source” and there is at least one failing test, we measure APFD for T_p, and report it for later inspection. In the fifth step, we update the current knowledge C, vectorize T_p, add it to the training dataset, and re-train the current machine learning model.

3.5.3 3rd phase: Transitive dependency selection

We carry out an additional experiment related to transitive dependency selection. This third phase of the case study is not related to the two earlier phases but is its own experiment. In this phase, we try to conclude whether transitive dependency selection can reduce test suite sizes. We apply transitive dependency selection to test modifying commits only. Since other TCS techniques can be applied to source commits, we try to find out if transitive dependency selection could be a sufficient TCS technique among test commits. We introduced transitive dependency selection in section 2.6.

This phase is quite straightforward:

1. Checkout to the previous test commit in version control history
2. Apply transitive dependency selection
3. Measure test suite reduction

We apply the above algorithm for as many commits as possible. Notice, that this algorithm is not depending on the data collected in section 3.5.1. We apply the algorithm in the non-preprocessed *git*-repository.

In the second step of the algorithm, we apply transitive dependency selection. We will now describe how we do it. We first create the dependency graph for the current test suite and their use dependencies, as in Figure 2.4. The use dependencies are effectively all Ada packages that are recursively imported with ”with” keyword from tests T. We recursively go through all file imports starting from the test cases, and this way create the dependency graph. After we have the dependency graph, we mark the changed modules in it with the help of *git diff* command. We then traverse the dependency graph from right to left to find out the test cases that depend on the marked modules. Once we have found the test cases, we mark this selection as the reduced test suite T'. In the third step of the algorithm, we measure the test suite reduction, e.g. $\left(1 - \frac{|T'|}{|T|}\right) \times 100$.

4. RESULTS

In this chapter, we go through the results attained from the TCS and TCP experiments. Furthermore, we display results for transitive dependency selection in test modifying commits.

4.1 Test case selection

Test case selection techniques were compared using Matthews correlation coefficient (equation 2.7) values. We consider Matthews correlation coefficient to be a sufficient surrogate for test suite reduction and recall. The boxplot in Figure 4.1 displays MCC-scores for each test case selection technique over 35 commits. The green triangle is mean and the orange line is median. The box presents values from lower to upper quartile. The whiskers display the range of the data, and the dots are flier points. The *Coverage(PH)* technique has the highest median and mean MCC-score, and *Random* technique the lowest.

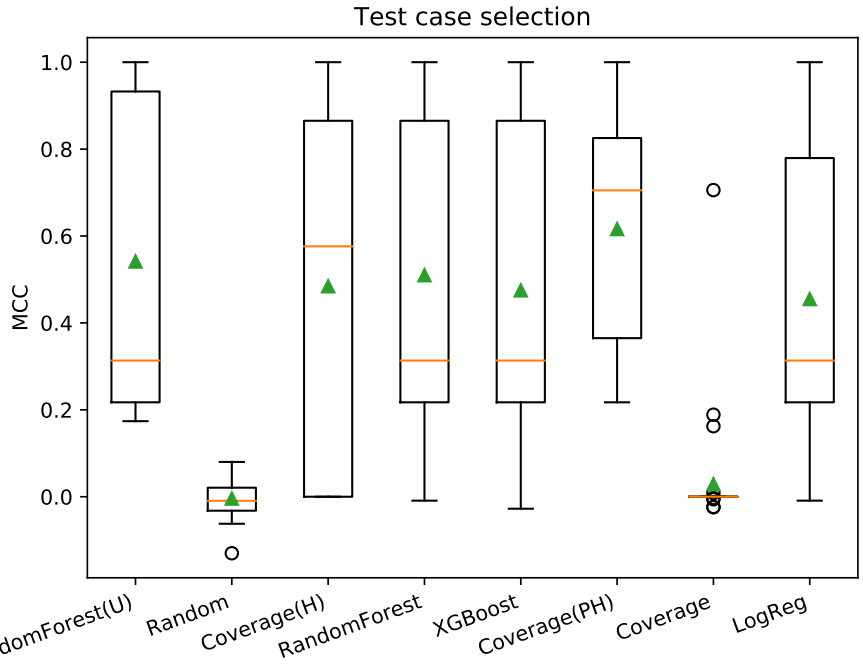

Figure 4.1. *Matthews correlation coefficient of each test case selection method over 35 commits.*

In order to examine the significance of the techniques, we carried out a Kruskal-Wallis test for the MCC scores across 35 commits. The result showed H-statistic of $117,9$ and the p-value of $2,07 \cdot 10^{-22}$ allowing the rejection of the null hypothesis (medians of the groups are equal). In order to find which of the groups were different, a pairwise post-hoc test

was carried out using Dunn's test with Bonferroni adjustment. The pairwise comparison is shown in Figure 4.2.

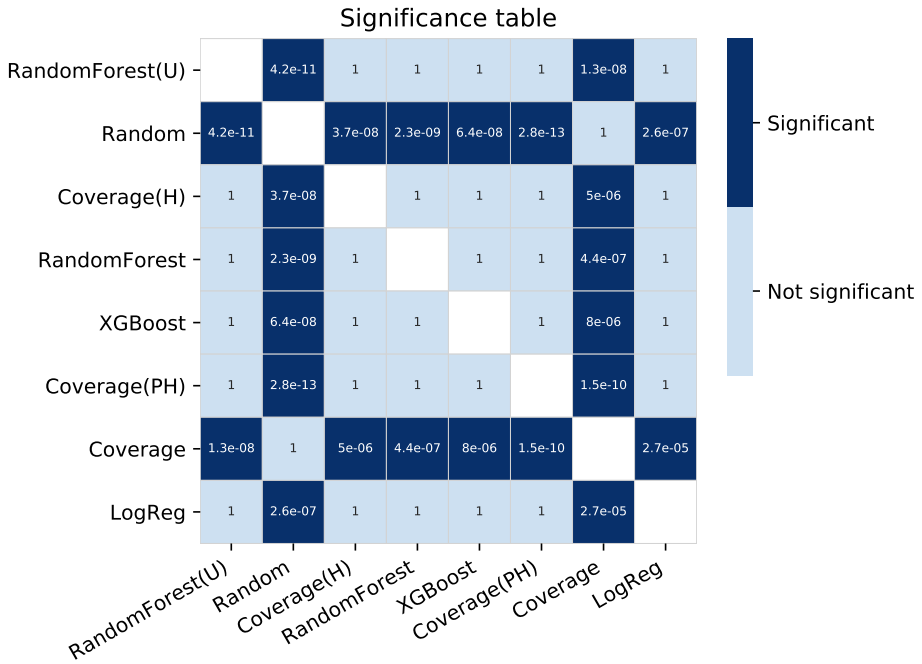

Figure 4.2. *Pairwise significance analysis using Dunn's test with Bonferroni adjustment. Any value below 0.05 indicate significant difference in Matthews correlation coefficient.*

Figure 4.3 shows MCC-trend for each heuristic (top) and each machine learning technique (bottom) across 35 source modifying commits. All machine learning techniques have fairly low MCC-values during the first 19 commits. Towards the end, the machine learning techniques have improved, producing higher MCC-values.

The Table 4.1 shows average recall and test suite reduction for each test case selection technique. The averages are calculated over all 35 source code modifying commits, and they do not praise the results of machine learning techniques. The machine learning techniques had a bad performance in the beginning due to low number of training samples, and these results are included in the averages.

The *Coverage(PH)*, *LogReg*, *RandomForest* and *XGBoost* techniques had an explicit limit for test suite reduction, and at least 98% test suite reduction was guaranteed for them. The *Coverage(H)* technique had the highest mean for recall (91.4%) and *Random* the lowest (39,5%). The Figures B.1 and B.2 in appendix B (page 63) illustrate the relation of recall and test suite reduction in more detail.

The Figures A.1 and A.2 in appendix A (page 61) show confusion matrices for each of the test case selection techniques across all 35 source code modifying commits. *RandomForest(U)* correctly predicted the highest number of failing tests. It was able to correctly predict 116 out of 141 failing tests. The technique with the highest average of MCC, *Coverage(PH)*, predicted correctly 97 out of 141 failing tests. The false positive values

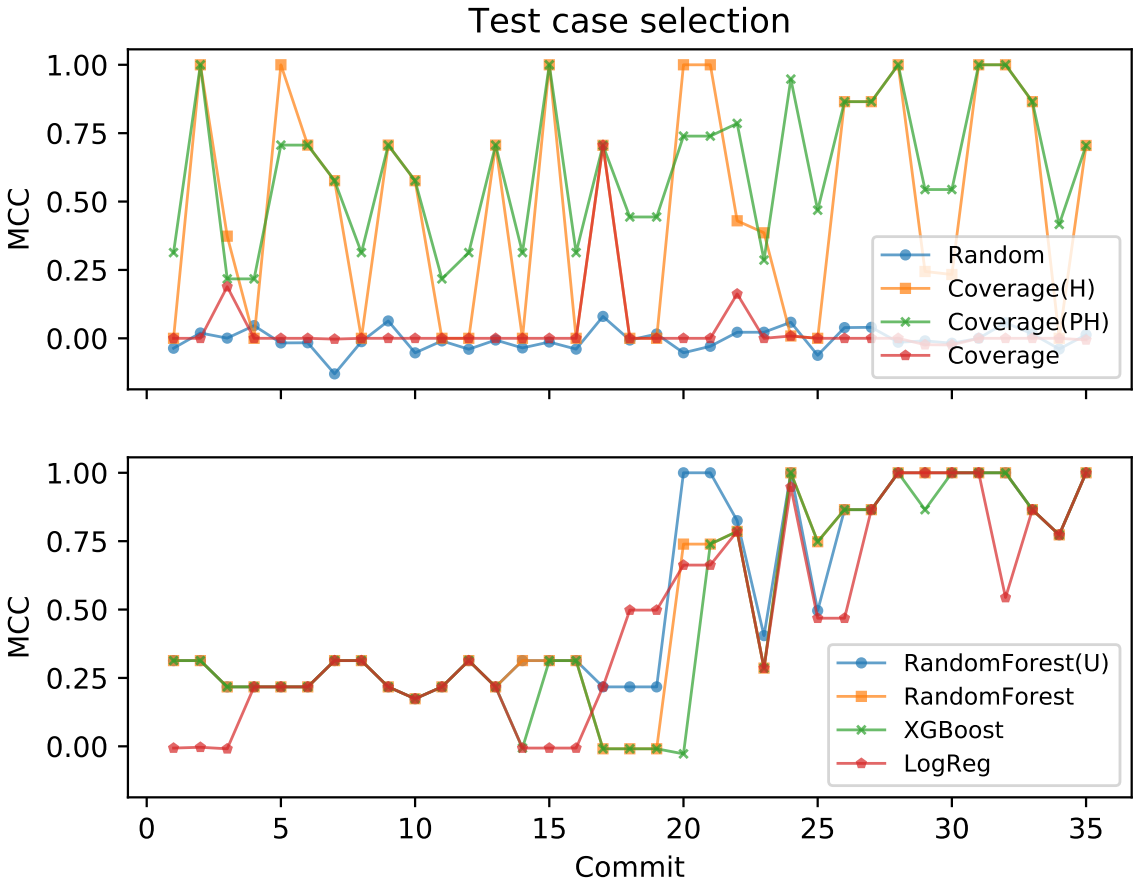

Figure 4.3. *MCC per method and commit. Trends for heuristics are shown in the top plot, and for machine learning techniques in the bottom plot.*

Technique	Recall	Test suite reduction
Random	0,395	0,561
Coverage	0,398	0,647
Coverage(H)	0,914	0,640
Coverage(PH)	0,785	0,987
LogReg	0,582	0,985
RandomForest	0,699	0,984
RandomForest(U)	0,779	0,982
XGBoost	0,655	0,984

Table 4.1. *Average recall and test suite reduction of each test case selection technique over 35 source code modifying commits.*

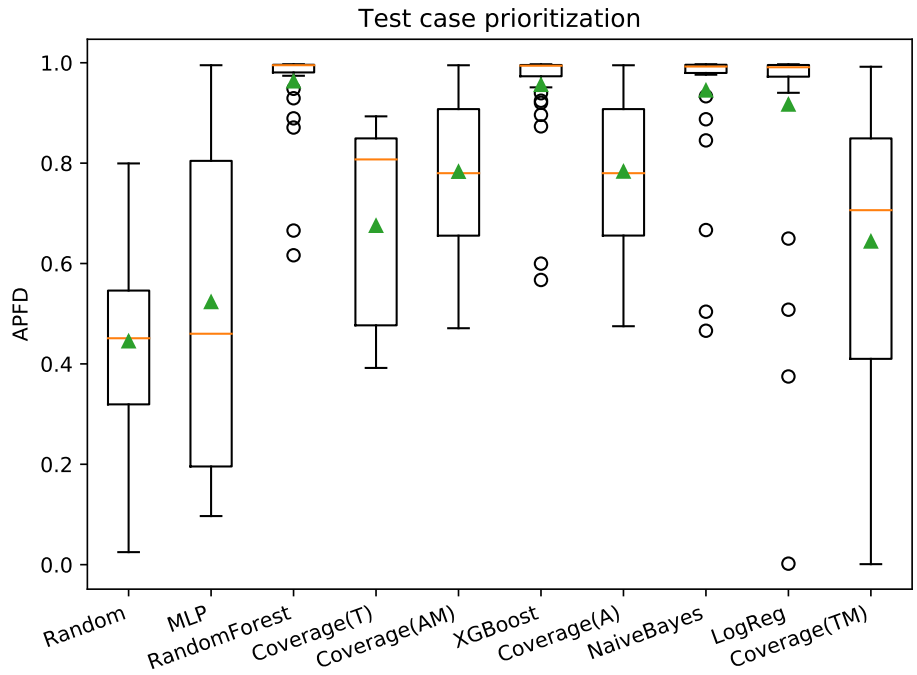

Figure 4.4. *APFD of each test case prioritization method over 35 commits.*

were 209 for *RandomForest(U)* and 131 for *Coverage(PH)*, *Coverage(PH)* being notably better.

4.2 Test case prioritization

We compared ten test case prioritization techniques and found out that four machine learning techniques surpassed the rest of the techniques. The Figure 4.4 shows APFD boxplots of all techniques. There was no statistical difference between the machine learning based prioritization techniques, except with multilayer perceptron, which was outweighed by other machine learning techniques and had an equal performance with random prioritization across 35 commits. Table 4.2 shows mean APFD values for each technique.

	APFD	
Technique	**Mean**	**Median**
Random	0,445	0,451
Coverage(T)	0,675	0,807
Coverage(TM)	0,644	0,706
Coverage(A)	0,783	0,780
Coverage(AM)	0,783	0,780
MLP	0,523	0,460
RandomForest	0,963	0,995
XGBoost	0,956	0,994
NaiveBayes	0,945	0,992
LogReg	0,917	0,991

Table 4.2. *Weighted average of faults detected (APFD) for each test case prioritization technique over 35 source code modifying commits.*

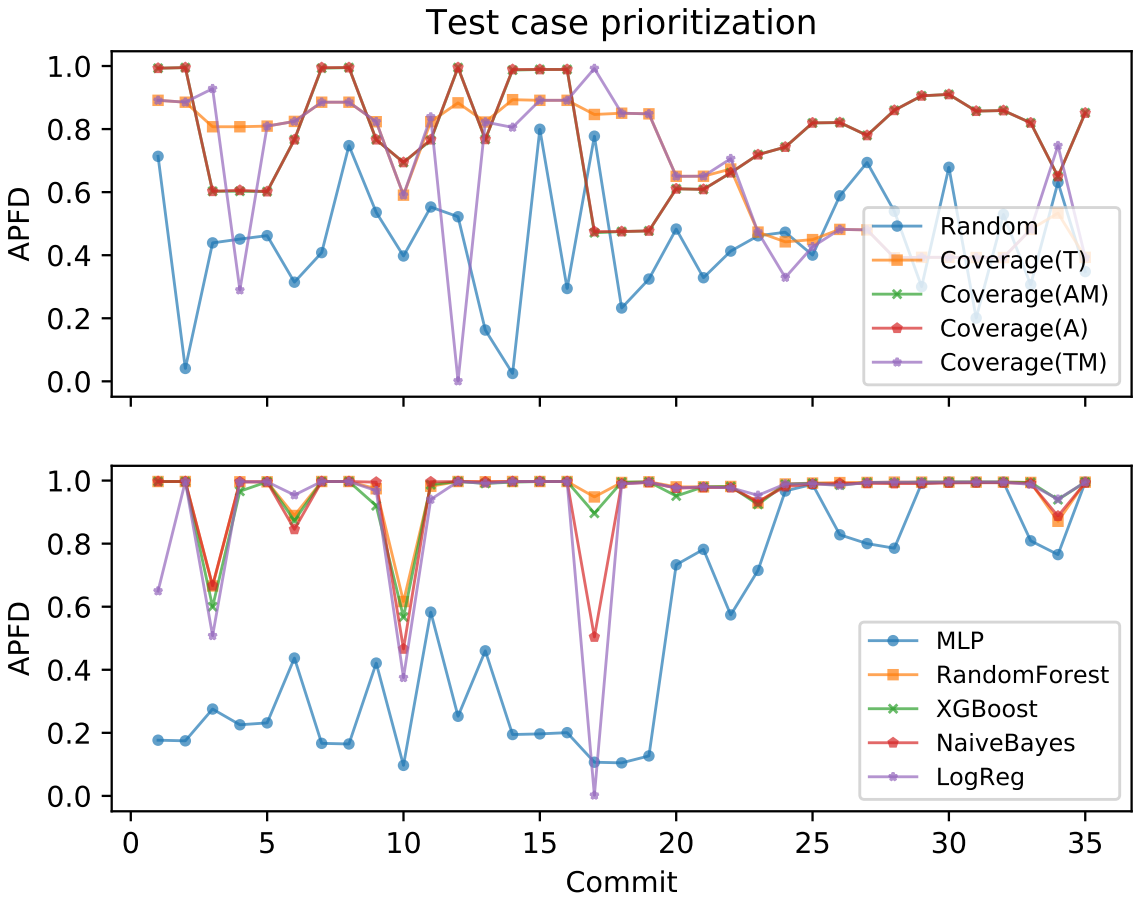

Figure 4.5. *APFD per commit for each test case prioritization technique. Trends for heuristics are shown in the top plot, and for machine learning techniques in the bottom plot.*

Figure 4.5 shows APFD trend for each prioritization technique. The APFD values are fairly high for all machine learning techniques starting from the first commit, except for multilayer perceptron. The APFD values also stay fairly high throughout the commits, but there are a number of commits where the APFD values suddenly drop. For the heuristics, APFD values fluctuate more. Interestingly, *Coverage(AM)* and *Coverage(A)* produce similar curves.

We carried out Kruskal-Wallis test for the APFD values showing H-statistic of $184,9$ and the p-value of $4,7 \cdot 10^{-35}$. This allowed the rejection of the null hypothesis and indicated that the median of at least one group had a significant difference. To find which of the groups were different, we applied a post-hoc test using Dunn's test with Bonferroni adjustment, similar to what we did in test case selection. The Figure 4.6 shows the pairwise comparison of each method.

4.3 Transitive dependency selection

Transitive dependency selection was executed for 347 test modifying commits. This is more commits than was collected in data collection section 3.5.1. Applying transitive

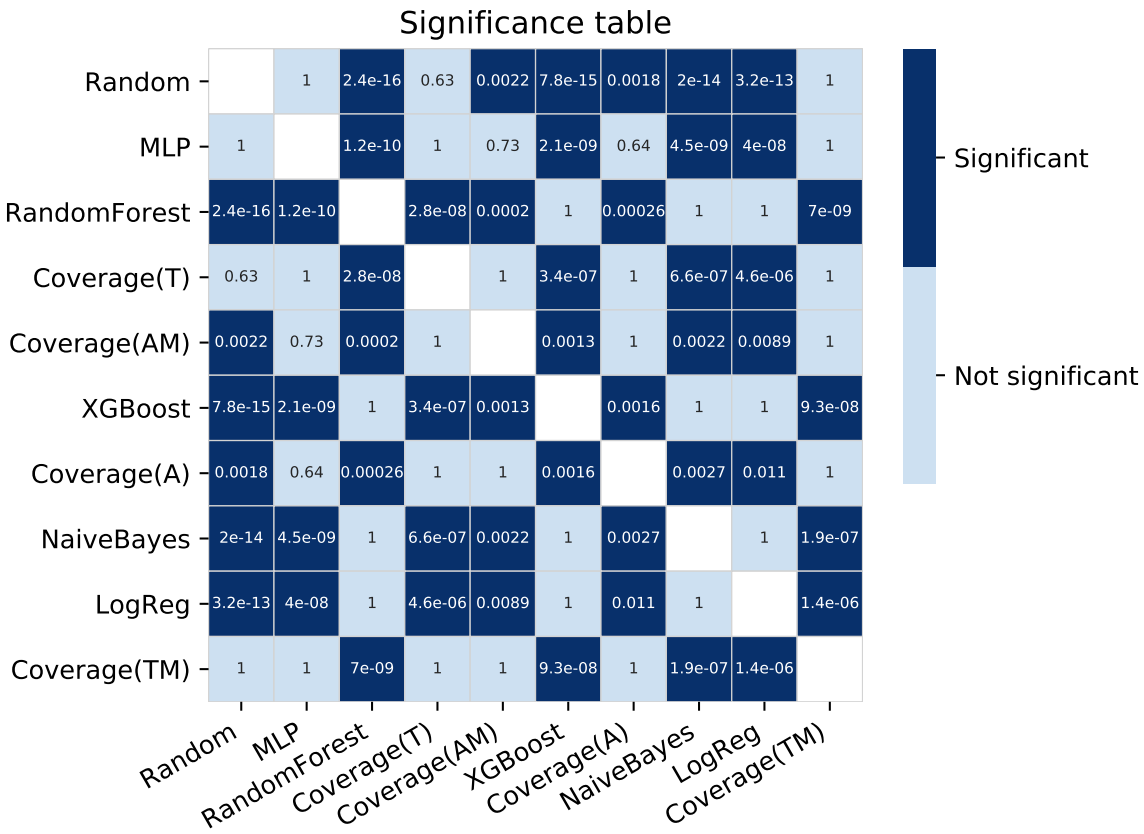

Figure 4.6. *Pairwise APFD significance analysis using Dunn's test with Bonferroni adjustment. Any value below 0.05 indicate significant difference in the means of APFD.*

dependency selection does not require executing test suites, and therefore we did not limit our analysis to the 27 test modifying commits. The disadvantage of not executing test suites is that we do not know how many failing tests (e.g. recall) transitive dependency selection was able to find. This could be interesting future work.

The histogram 4.7 shows results for test suite reduction of transitive dependency selection. The average reduction over 347 commits was $78,8\%$ and the median was $99,6\%$. Interestingly, there are two visible peaks on both ends of the histogram. Most of the time the reduction was over 95%, indicating that transitive dependency selection has the potential to reduce test suite significantly. The reduction was never either 0% or 100%.

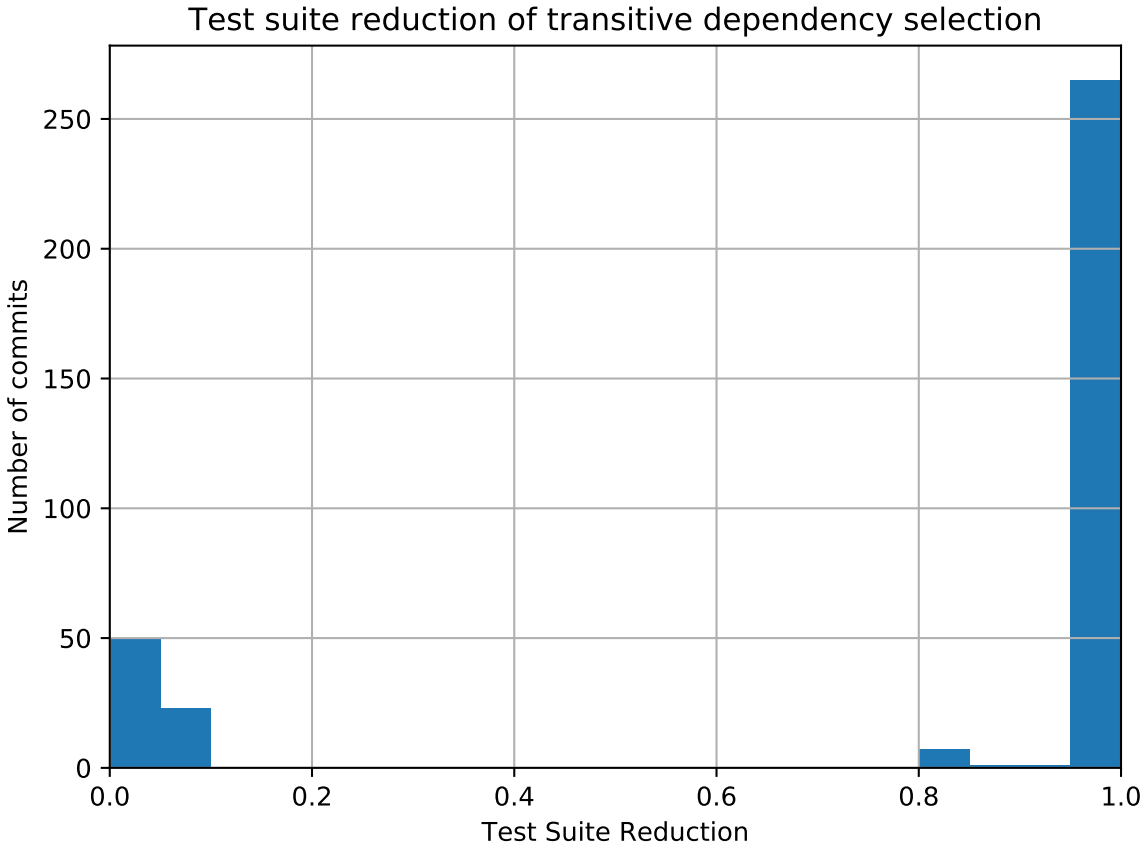

Figure 4.7. *Transitive dependency selection produced test suite reduction over 95% for most of the test modifying commits.*

5. DISCUSSION

This chapter discusses the results of different test case selection and prioritization techniques that we applied to the data collected in section 3.5.1. We compare our results to previous studies and analyze if our techniques could enhance automatic testing and facilitate continuous integration.

5.1 Test case selection

In our test case selection case study, we compared the performance of eight test case selection techniques. Four of the techniques were based on heuristics, and the rest four were based on machine learning. We measured three different performance indicators for each technique, namely test suite reduction, recall, and Matthews correlation coefficient. The MCC-score was used to differentiate the well and poorly performing techniques in a form of significance analysis using Dunn's test with Bonferroni adjustment (Figure 4.2).

5.1.1 Heuristics

The significance analysis reveals, that *Coverage* and *Random* test case selection techniques are outweighed by the rest of the techniques. Interestingly, these two techniques do not have a statistical difference in their performances. To understand why the simple coverage technique did not perform better than random technique, we must carefully look at the numbers of test suite reduction provided by the coverage technique. In 16 source modifying commits, *Coverage* technique provided 100% test suite reduction, meaning that it did not select any test case for execution and therefore recall and MCC scores were 0. This implies, that the modified parts were not covered by any test case with the statement coverage criteria. The high number of commits (16 out of 35) can be due to the outdated coverage information, but arguably the total test suite coverage and uninstrumentable parts of the software are the main factors for our findings. We performed an analysis of the test suite coverage in one software version (see 2.3.2), and found total test suite coverage of 64%. We can deduce, that randomly modifying one statement in this software version has 36% chance to cause 100% test suite reduction. The chance is greater if we also consider modifications to the uninstrumentable parts of the software, such as header files that contain only variable definitions. We think, that at least these three factors: total test suite coverage, uninstrumentable parts of the software and outdated coverage information were the reasons why *Coverage* technique did not select any test cases in a high number of commits. To answer why *Coverage* did not outperform *Random* technique, we also have to consider an important finding in our study: the same tests quite often fail in consecutive

commits. Therefore, coverage over modifications is not a strong test failure predictor in our case study.

The other extreme in *Coverage* selection technique is to select every test case for execution and to provide no reduction in test suite size at all. This happened in 11 out of 35 source modifying commits. We thought part of the reason for this being in the overlapping coverage (Figure 2.2). In our total test suite coverage analysis, we found out that approximately 25% of the software is covered by 500 or more test cases (out of 528 test cases). It seems, that the commits quite often tend to modify these ”hot spots”, yielding no reduction in coverage based test case selection. In these situations, 100% recall in ensured, but MCC score is always 0. Combining the two extremes, where *Coverage* either selects every test or not a single test case, happens in 27 out of 35 commits, and therefore the general performance of *Coverage* is not good. The Figures 4.3 and B.1 visualize this problem.

RQ1: How big test suite reduction can code coverage based test case selection achieve?

The coverage based test case selection (*Coverage*) achieved test suite reduction of $64,7\%$ while having a recall of $39,5\%$ on average. The MCC scores have no significant differences from *Random* technique, and therefore the two techniques have equal performance. This indicates, that the simple coverage selection technique is not sufficient to speed up testing and facilitate continuous integration, but other techniques should be used instead. The test suite reduction is often either 100% or 0%, e.g. $T' = \emptyset$ or $T' = T$. In both of these situations, TCS does not provide any help to the tester.

We were not the first to measure fluctuations in test suite reductions. Harrold et al. [19] also experienced fluctuating test suite reductions in their coverage-based TCS technique. They recorded 0% test suite reduction in 6 out of 32 software versions. We recorded a 0% reduction in 11 out of 35 commits. It is a good question, why we experienced this more often than Harrold et al. We think, that the reason for this is behind our validation tests and their tendency to have high coverage overlapping. Harrold et al. also measured high reductions, such as 98%, but never as high as 100%. Our results are therefore more polarized than results of Harrold et al. We think, that Harrold et al. had larger modifications between software versions than we had between consecutive commits. This could explain why Harrold et al. never experienced 100% test suite reduction, but we did.

Di Nardo et al. [11] applied coverage-based TCS techniques in an industrial system. In most cases, Di Nardo et al. were unable to produce any reduction in test suite sizes. The maximum test suite reduction they recorded, was 2%. Di Nardo et al. compared only four consecutive software versions. This strengthens the viewpoint, that large number of modifications between software versions cause smaller reductions. We also experienced a lot of small reductions (0%), but they were due to coverage overlapping rather than large modifications in the software.

The *Coverage(H)* technique was able to resolve part of the issues of *Coverage*, providing significantly better results. It had an additional way to predict a test failure, namely the

latest pass. It selected every test case that either covered a modification or failed in the previous commit. *Coverage(H)* yielded 91,4% recall and 64% test suite reduction on average, prominently increasing the recall value from the simple technique and never providing 100% reduction. This technique did not, however, fix the other problem of the simple coverage method, but still executed the full test suite in the same 11 commits. To overcome this problem, the *Coverage(PH)* used an extra prioritization step to further select a number of tests. Wong et al. [43] were the first to apply prioritization after test case selection, and later others used it too [5][6].

Beszédes et al. [6] used priority-based test case selection to reduce test suite size in the WebKit web browser engine. In their initial experiments, they selected every test case that covered the modified procedures in the software, or that had failed previously. This technique is equivalent to our *Coverage(H)* technique, the coverage criteria being the only difference. Using this initial selection, Beszédes et al. witnessed a test suite reduction of 79,43% with 95,08% recall on average. Later on, Beszédes et al. applied their selection technique in an actual live system with a more realistic setup. They witnessed a test suite reduction of 51% with 75,38% recall on average.

The results of Beszédes et al. and us are roughly similar, but the differences can be partly explained by the coverage criteria. Beszédes et al. used a procedure level coverage, while we used statement (or line) coverage. Fine-grained coverage can yield a bigger test suite reduction and more efficient fault detection rates. On the other hand, procedure-level coverage information can stay up to date longer compared to statement coverage as time passes, and therefore coarser granularity can select test cases with more confidence even with outdated coverage information.

When Beszédes et al. [6] extended their selection technique with an extra prioritization step, they witnessed test suite reduction of over 90% with 38% recall. Their technique resembles our *Coverage(PH)* technique, but we used the test history information to prioritize the test cases in addition to coverage information. We were able to produce 98,7% test suite reduction with 78.5% recall on average with *Coverage(PH)*. It is surprising how much greater recall we were able to get by simply using test history information as one of the prioritization criteria. The TCS performances were roughly similar, but using test history information in the prioritization step can make a big difference in recall.

The *Coverage(PH)* technique is indeed our most promising test case selection technique among the heuristics. This technique uses priority-based test case selection over modification coverage and test history. It selects every test case that covers a change or failed in the previous commit. If the selection size is still too large, it reduces the selection by prioritizing the selected tests using failure rate, latest pass, and modification coverage.

5.1.2 Machine learning

When incremental learning is applied in test case selection, the assumption is, that the performance of the machine learning models gradually increases as tests are being executed and new labeled data samples are accumulated in the training dataset. Because the MCC scores assumingly increase over time, using the significance table to examine the performance overall commits may not be that interesting. It is more interesting to know, that do the machine learning techniques eventually reach the same performance as the heuristics, and if so, then how long time does it take to reach a similar performance? To investigate this, Figure 4.3 shows the performance of each technique over time. Indeed, every machine learning technique shows a positive trend for the MCC scores, where the techniques perform better in the end than in the beginning.

Looking at Figure 4.3, we can see that the performances of the machine learning techniques gradually increase, and towards the end, they perform equally or better than the heuristics. There are small differences between the performances of the techniques, but they seem to roughly follow a similar trend. It looks like, that at commit number 20 all of the techniques gain a positive boost and they perform better than in commit 19. We figured out that the reason behind this lies in test history. There is a squashed test commit between commits 19 and 20 (see Figure 3.1). The transitive dependency selection executed 501 tests in this specific commit, where 18 tests failed. This caused the test cases to update their history, most importantly the *latest pass*. The machine learning techniques had so far learned to affiliate test history with currently failing tests, and therefore the techniques selected all the previously failed tests for execution in commit 20, rendering a high MCC score.

It can be questioned whether such a big test suite is permitted in the squashed test commit since we try to maximize the reduction. However, recall that the test commits are squashed. In reality, there are many test modifying commits next to each other. Most of the time, selecting transitively affected tests cause only a small portion of tests to be selected, as we showed in section 4.3. Even though it seems that, as depicted in Figure 3.1, the test commit executes the almost full test suite, it is more likely that there are many test commits next to each other that over time execute the full test suite instead. The way how we separated test and source code modifying commits, can be distracting, and we discuss this matter more in section 5.4.

During the first 19 commits, the machine learning techniques have fairly low MCC scores possibly due to the low amount of negative samples in the training data. Between commits 20 and 35 however, the machine learning techniques seem to perform better. The Table 5.1 collects the recall and test suite reduction values of each technique between the commits 20 and 35. By focusing on the differences in this table and Table 4.1, we can examine which techniques were able to improve their performances.

Compared to Table 4.1, the machine learning techniques have increased their recall, but also a bit of test suite reduction. *RandomForest(U)* technique outperforms *Coverage(PH)*

	Commits 1-19			Commits 20-35		
Technique	**Recall**	**TSR**	**MCC**	**Recall**	**TSR**	**MCC**
Random	0,316	0,588	−0,010	0,489	0,528	0,003
Coverage	0,553	0,524	0,047	0,215	0,793	0,007
Coverage(H)	0,886	0,523	0,387	0,947	0,778	0,600
Coverage(PH)	0,781	0,988	0,515	0,790	0,986	0,736
LogReg	0,412	0,982	0,189	0,783	0,987	0,771
RandomForest	0,623	0,980	0,220	0,790	0,990	0,854
RandomForest(U)	0,702	0,980	0,255	0,871	0,983	0,881
XGBoost	0,570	0,980	0,203	0,755	0,990	0,798

Table 5.1. *Average recall, test suite reduction and Matthews correlation coefficient for each test case selection technique. Most of the techniques had higher MCC score between commits 20-35, indicating superior performance to performance between commits 1-19.*

in recall with a slightly lesser test suite reduction. The rest of the techniques also provide competitive results to *Coverage(PH)*. *Coverage(H)* still remains the technique with the highest recall.

RQ2: How effective is incremental learning based test case selection?

The best performing machine learning model was an unlimited random forest (*RandomForest(U)*), which achieved a test suite reduction of $98,2\%$ and recall of $73,1\%$ on average. Towards the end the recall was notably higher, rendering MCC-score also higher. After twenty commits the machine learning techniques started to perform notably better and reached a similar performance to the best heuristics. This is a promising result for incremental learning based test case selection and shows that machine learning techniques have the capability to outperform heuristics in a relatively small number of commits. Therefore, incremental learning is suitable in test case selection and can be used to speed up testing and to facilitate continuous integration.

A distracting detail of the machine learning techniques is that at commit 17 the faulty tests were explained by modification coverage alone (Figure 4.3). Some of the machine learning techniques apparently had no previous knowledge of associating failing tests with coverage, and the failing tests were missed. Because the failing tests were not executed as part of the test suite, their test history remained unaffected, leading to the unfortunate situation of missing the same tests in commits 18 and 19 as well. Logistic regression and the unlimited random forest were the only techniques to select some of these failing tests.

The inconvenience of commit 17 was not repeated. A similar situation happened in commit 22, where a portion of test failures was explained by modification coverage alone. Fortunately, the machine learning models had learned to associate modification coverage with failing tests, and the failing tests were correctly predicted. It is not however safe to think, that eventually all associations are discovered. It is possible, that some associations are never found.

It is possible, that outlier detection or another unsupervised learning method could have spotted the failing test in commit 17, because the *modification coverage* was prominently

high for the failing tests. This could have been registered as an anomaly. Spotting and executing the tests that have odd or exceptional features could supplement test case selection. Using outlier detection to figure out yet undiscovered patterns would be an interesting future study.

Spieker et al. [40] used reinforcement learning to select and prioritize test cases, and their technique required 60 consecutive commits to perform equally or better than comparison techniques. The test case selection results in our case study suggest, that our method needs approximately 20 source code modifying commits to provide similar results with the comparison techniques. The machine learning techniques provide similar or better MCC scores compared to the *Coverage(PH)* technique after 20 commits. This could indicate, that using a different model (e.g. random forest classifier instead of multilayer perceptron), accumulating training data and re-training the machine learning model in every iteration, and using more features in addition to test histories, such as coverage information and text similarity scores, can help to reach the saturation point faster. The results we achieved in our case study, are not outright comparable to results with Spieker et al., because our experimentation setups were different, the comparison methods were different and the used measures were different, namely NAPFD and MCC. We also did not validate our results with other projects but applied our techniques to a single software project only. Therefore, more investigation would be required to compare results more reliably with Spieker et al. and us.

Busjaeger and Xie [10] used supervised learning and pointwise ranking to prioritize test cases. Using their prioritization technique, they were able to select 3% of the topmost test cases and provide 75% recall. Such selection equals to 97% test suite reduction. Our results are approximately similar, but we achieved the results with less training data. The results in our case study suggest, that if initial training data does not exist, incremental learning can eventually achieve similar performance to supervised batch-learning in TCS. The saturation point where we reached a similar performance to Busjaeger and Xie, was at around 20th commit.

5.2 Test case prioritization

In our test case prioritization experiment, we compared the performance of ten different prioritization techniques. Five of these techniques are based on heuristics, e.g. on coverage information, and they have been studied in the past [11][36]. The other five techniques are based on incremental machine learning. In order to compare the performances of these techniques, we measured the weighted average of the percentage of faults detected (APFD) for each technique. Then we applied Dunn's test with Bonferroni adjustment to the APFD values (Figure 4.6) to differentiate the well and poorly performing techniques.

5.2.1 Heuristics

The Figure 4.6 shows the significance table for ten prioritization techniques. Considering the heuristics (*Random, Coverage(T), Coverage(TM), Coverage(A), Coverage(AM)*), the random prioritization was outperformed by both additional coverage prioritization techniques. Surprisingly, the total coverage prioritization techniques did not have a statistical difference to the random technique, even though the mean and median values for APFD were prominently higher for total techniques compared to random technique, as depicted in Figure 4.4. Apparently, the deviation in performances of the total techniques was too large.

The additional coverage prioritization techniques have been shown to outperform total coverage prioritization techniques in the past [11]. Based on the significance analysis, we were unable to support the claim. According to our results, the total coverage techniques perform equally well with additional coverage techniques. However, based on the boxplot 4.4, we would still choose additional coverage prioritization techniques over total techniques, because the average APFD is higher, and the lower and upper quartiles are higher and closer to each other for additional techniques. Against the findings of Di Nardo et al., Rothermel et al. [36] found total statement coverage technique being equally or better performing than additional statement coverage technique. Our results do not support this either.

Di Nardo et al. [11] produced mean APFD value of 74.2% and a median of 74.1% with additional coverage prioritization technique. Total coverage approaches with the block-coverage criterion produced mean APFD of 59.7% and a median of 59.7%, respectively. Our results are slightly higher, means and medians being 78.3%, and 79% for the additional coverage approach, and 69% and 80% for the total approach. The differences can be related to the different coverage-criterion used, to the data, or to the way how APFD was calculated. Di Nardo et al. used real regression faults to calculate APFD in their experiment, but we assumed that one failing test reveals one unique fault in the system, and calculated APFD accordingly. This can indicate, that our prioritization results are slightly over-positive, and if we applied our techniques in a real system with real regression faults in interest, the APFD values would not be as high. The same assumption, that a failing test reveals a single unique fault, has been made in other studies too [40][10].

Di Nardo et al. [11] showed in their study, that the modification information does not improve TCP results. We support this claim. According to Figure 4.6, there is no statistical difference between *Coverage(T)* and *Coverage(TM)*, and neither between *Coverage(A)* and *Coverage(AM)*, pointing out that using modification information to prioritize tests does not improve APFD values. Additionally, modification aware techniques require extra work to find out the modifications between P and P', e.g. with *git diff* command. According to results by Di Nardo et al. and us, there is no reason to do this extra work. The modification unaware techniques perform equally well compared to modification aware techniques.

A surprising detail of the two additional prioritization techniques is, that their performances are almost exactly the same, as shown in Figure 4.5. They produce the same APFD values in 27 out of 35 commits, and in 8 commits the values are close to each other. We found out, that in 26 out of 35 commits both of the techniques even orders the test cases similarly, producing exactly the same prioritized test suite. We did not thoroughly study the reason why the orders are similar so many times, but we think that it relates to the fallback method of the modification aware technique. Whenever it is satisfied including all the code modifications, it proceeds with the basic additional coverage technique. Generally, the commits tend to modify only a small number of statements, and therefore the fallback to basic additional coverage happens early.

The commit number 17 in Figure 4.5 shows that modification aware prioritization technique correctly ordered the test suite while the additional coverage with modifications did not. The disadvantage of the additional technique is that if a modification breaks multiple tests, it is unlikely that the broken tests are ordered next to each other. This was shown in commit 17. On the other hand, since all the broken test reveal the same fault, the order of the rest test cases do not matter as long as the first failing test is executed as early as possible. We do not consider the actual faults in this experiment but treat a failing test as a link to a unique fault, which arguably distorts the results.

5.2.2 Machine learning

Similarly to test case selection, the assumption is that the performance of incremental learning prioritization techniques gradually increase when test suites are executed and more training data is being accumulated in the training dataset. Therefore, we would expect to see an ascending curve for APFD values in Figure 4.5. Surprisingly enough, the figure shows a rather stable curve for most of the machine learning techniques, with a number of downward spikes at commits 3, 6, 10, 17 and 34. The techniques also seem to perform unnaturally well starting from the first commit, and the performance is close to 100% in many commits. Only the multilayer perceptron seems to behave more or less the way what we would expect to see, starting from low APFD values and gradually improving its performance and producing higher APFD values on the way.

We explored the reason why APFD was so high in the beginning and found out that it is due to test history being a good predictor on the test failures. Commits 1 and 2 did not introduce any new test failures, but instead, the same tests failed in the commits 0, 1 and 2. The commit 0, denoted as T in Figure 3.1, is used to initially train the machine learning techniques. Apparently, the machine learning techniques had learned to affiliate test history with failing tests in the training phase at commit 0, and therefore prioritizing the previously failing tests first immediately produced high APFD scores. We found similar cases elsewhere.

We investigated further the issue and found out that commits 3, 6, 9, 10, 11, 13, 14, 17, 20, 22, 23, 25, 26, 33, and 34 introduce new test failures compared to their previous commit.

Conversely, commits 1, 2, 4, 5, 7, 8, 12, 15, 16, 18, 19, 21, 24, 27, 28, 29, 30, 31, 32 and 35 do not introduce any new test failures compared to their previous commit, but all of the failing tests failed in the previous iteration as well. Therefore, most of the time we would have got a high APFD score by ordering the test cases by the latest pass feature. We realize, that it would have been a good idea to add such a technique into the pool and compare its performance to e.g. machine learning techniques, but in this study, we were more interested to investigate how the traditional coverage-approaches compare to machine learning techniques. Busjaeger and Xie [10] has already shown, that ML-based prioritization outperforms history based prioritization. How many consecutive commits incremental learning requires to outperform history based prioritization, remains an open question.

A note we can make about our version history and its test failures is that the same tests quite often fail in consecutive commits. We think that the reason for this is, that the testers did not execute the whole test suite in every commit, and therefore the failing tests remained unexposed to the testers and the failing tests were not immediately fixed. This is the reason, why the project was a good target for test selection and prioritization after all. If we had used any of the ML prioritization techniques in the actual project, the testers would probably have fixed the failing tests earlier. Many times, the consecutive failures in the version control history were build or execution failures. It is possible, that the testers were aware of the failing tests all along, and there was no intention to fix the failures right away.

The Figure 4.5 shows negative spikes at commits number 3, 6, 10, 17 and 34 for the machine learning prioritization techniques. These spikes happen at commits where new failing test cases were introduced. The commit 3 introduced a code modification that broke the tests. The modification was detectable by *modification coverage*, but none of the techniques had affiliated modification coverage with failing tests so far. Commit number 6 introduced a new test case with a build failure, but we do not know the reason for the failure. The commit before it tampered tests (see Figure 3.1), and arguably something in this commit broke the test while the transitive dependency selection was unable to trace the change. The commit number 10 introduced two new failing tests, and the failures were caused by a change in a constant value. Because coverage does not editorialize global variables and constants, modification coverage could not detect the change and the failures were missed. The failures of commit 17 were explained by modification coverage alone. The random forest had already learned to associate modification coverage with failing tests and therefore produced a high APFD, but the rest ML-techniques did not perform so well. The commit number 34 modified a part of the source code, that was covered by many test cases, and apparently, there were a lot of failure candidates, but only one test case failed due to this change.

The significance analysis (Figure 4.6) reveals, that all of the machine learning techniques excluding multilayer perceptron outperforms random prioritization and all of the coverage-based prioritization techniques. This is an interesting result and shows that even without a

rigorous amount of training data, the ML-based test prioritization produces better APFD values compared to the traditional approaches. Our results suggest, that ML prioritization techniques can produce high APFD values starting from the first commit. Therefore, when initial training data does not exist, incremental learning can be used to speed up the commission of machine learning based test case prioritization. The expectation is, that the APFD values are higher compared to APFD values produced by traditional approaches starting from the first commit.

Interestingly, multilayer perceptron did not differentiate from random prioritization. It is possible that the multilayer perceptron requires more negative samples, higher learning rate or resampling strategies to converge faster. The Figure 4.5 shows a positive trend for MLP and suggests that eventually, it performs as well as the other ML techniques, but it requires more commits and data to gain similar performance. Towards the end, it performed almost equally well with the other machine learning techniques. Neural networks have been used in many machine learning tasks lately, even in test case prioritization [40], but our study suggests that in test case prioritization, there are possibly better model alternatives when incremental learning is applied, for example, random forests and boosting techniques. Our approach was to fully re-train the machine learning model when new data arrives, and therefore we do not know if e.g. truly online random forests outperform online neural networks. Lakshminarayanan et al. [25] were able to provide a comparable performance to periodically re-trained random forests with Mondrian forests, and therefore applying such techniques in test case prioritization would be an interesting future study.

RQ3: How do the incremental learning based test case prioritization techniques compare to traditional coverage based prioritization techniques?

According to the significance analysis (Figure 4.6), the incremental learning techniques outperform traditional statement coverage based prioritization techniques in fault detection rates, when a failing test is assumed to reveal one unique fault. Only the multilayer perceptron did not show a statistical difference to the traditional techniques in the inspection range from commit 1 to commit 35 in APFD values. Towards the end, it performed equally or better than the traditional techniques. The rest of the ML techniques produce high APFD values starting from the first commit. This is a surprising result because the assumption for incremental learning is that the performances are initially low and they gradually increase. The reason for the good initial APFD performance was thought to be due to test history being a good predictor on test failures. In our dataset, many same tests failed in consecutive commits, and therefore prioritizing the test cases that failed in the previous commit produce high APFD values. The ML techniques seemingly had learned to affiliate test history with test failures during the initial training phase at commit 0. However, the ML techniques did not have a stable performance. The ML techniques produced abrupt low APFD values at a number of commits, e.g. at commits 10 and 17 (Figure 4.5). The abruptly low APFD values are possibly justified by the fact that the selected features were unable to explain the test failures (commit 10), or that the ML techniques had not yet learned to affiliate certain features with test failures (commit 17). There was no statistical difference between the four

best-performing machine learning techniques, but naive Bayes, logistic regression, random forest and gradient boosting technique from xgboost-library had an equal performance in fault detection rates. The random forest classifier, however, was possibly the best technique to handle the surprising ”special” commits at 10 and 17.

5.3 Transitive dependency selection

We searched all transitively affected test cases from 347 commits to reduce the number of tests that need to be executed during a commit that modifies *test/** directory only. We did not consider dependencies between *test/* and *src/* directories, but only the tests and their dependencies under *test/* directory. We applied other test case selection and prioritization techniques for the source code modifying commits. We recorded an average reduction of $78,8\%$ and a median of $99,6\%$ with executing all transitively affected test cases. The Figure 4.7 shows, that most of the time the reduction was over 95%, and some times below 10%. Surprisingly, we did not record any reductions between these two extremes. Our result suggests, that selecting transitively affected tests reduce the number of tests significantly for commits that modify tests or their use dependencies.

Gligoric et al. [17] used their tool, ”Ekstazi”, to track changes in dependant files. With their tool, Gligoric et al. reduced end-to-end testing time by 32%. This is not the same as test suite reduction, and therefore our results are not directly comparable, but arguably the time and size reduction are roughly following each other which allows a coarse comparison.

Our technique is slightly different compared to Gligoric et al. We track all the dependent Ada-files using static analysis on the recursive file imports. Using this technique, we possibly got a higher test suite reduction than Gligoric et al. A higher test suite reduction is expected because our technique selects only Ada-files, and not the configuration or other metafiles at all. Additionally, Gligoric et al. used their technique for every kind of changes in their system, while we considered only modifications under *test/* directory. We don’t know how much this affects the results, but arguably it either increases or decreases the test suite reduction that we reported. Thirdly, Gligoric et al. validated their results with 615 revisions from 32 open-source projects, while we used 347 revisions from a single project. Considering these three points, the 32% testing time reduction compared to 78.8% test suite reduction still sounds like a big difference, and therefore we argue that our results need more verification from other projects.

Yoo et al. [45] used dependency coverage (equation 2.11) among other features to select and prioritize test suites. Yoo et al. reported an average test suite reduction of 68% with their technique. This strengthens the viewpoint that the results produced by our transitive dependency selection among test modifying commits require further verification.

5.4 Future work

In any further test case selection or prioritization studies, we suggest the use of dependency coverage (equation 2.11) in addition to the features explained in section 3.3. The dependency coverage was introduced by Yoo et al. [45]. The dependency coverage could be especially helpful in projects, where tests and source code are maintained in the same repository. A modification of a test case or any of its dependencies can cause the test to fail, and therefore dependency coverage could be a strong predictor for a test failure and a nice addition to the features described in 3.3.

Furthermore, the dependency coverage could have fixed our complex case study setup in section 3.5. We separated how *test/** and *src/** modifying commits are handled, and this made our setup complex. With dependency coverage, we could have handled every commit similarily, i.e. use the same test case selection or prioritization algorithm for every commit regardless of the commit type. We consider this as a benefit in any future studies.

In our case study, we accumulated training data and fully re-trained the machine learning models in every iteration. This produced good results, but in reality, this can be infeasible when the training data grows large. We propose the online Mondrian forest technique introduced by Lakshminarayanan et al. [25] for future studies instead. We showed, that random forest outweighs multilayer perceptron in the range of 1-35 commits in test case prioritization, and therefore propose Mondrian forest as an online alternative for the random forest.

We applied transitive dependency selection over 347 test modifying commits. We measured test suite reductions, but not recall, because we did not have full test execution information for these commits. Finding the proportion of failing tests in the reduced test suites would have been an interesting case study. We will possibly study this in the future.

During our case study, we thought unsupervised learning to be an interesting case study in the areas of test case selection and prioritization. For example, outlier detection could spot tests with surprising behaviors ahead of time and reveal yet undiscovered patterns. We also propose exploiting semi-supervised learning in test case selection, where the training data accumulation can be rather slow. Semi-supervised learning could be used to fill the gaps in the labels of the training data.

Another idea we had during our case study, was that machine learning could be used to spot non-deterministic test cases. Any test case without an apparent reason for failure could be interpreted as a non-deterministic test case. The false negatives are an interesting category in this sense. According to Martin Fowler [13], spotting and eradicating the non-deterministic test cases from the test suite early is important, because they can have negative and infectious effects on developers attitude towards failing tests.

5.5 Threats to validity

There are many threats to validity. Firstly, we deleted all the non-deterministic tests from the test suite before the experiments. This arguably distorts the test selection and prioritization results. However, we argue that test history features, such as the latest pass and failure rate described in section 3.3 are the key features to explain even the non-deterministic test case failures.

We used a different test environment during the data collection (section 3.5.1) than what is used in the actual project. These two test environments are similar, but they use a different amount of hardware simulation. This could have brought excessive discrepancies on test verdicts between the test environments.

We do not know the actual faults in the software and we treated a failing test as a link to a unique fault. Unfortunately, this means that the weighted average percentage of faults detected (APFD) values are just estimates. Previous studies have made the same assumption, that a failing test reveals one fault [6][40].

Because of separating how src/* and test/* commits are handled, the experiment setup became complex. We reported MCC, recall and test suite reduction of source modifying commits only, and ignored the values for the rest commit types. Using dependency coverage as a new machine learning feature could have fixed this issue. This was already pointed out in future work section 5.4.

We found out that coverage information produced by *gcov*-tool is not accurate when a statement contains line breaks. In such situations, the first line is only detected by gcov, and the rest of the lines are ignored. Our software code contained abundance number of statements that split to multiple lines. Therefore, the coverage based selection and prioritization techniques could have been affected.

Finally, we applied test case selection and prioritization to a small amount of data from one software project only. This suggests that external validity can be affected. We had planned to include another project if there was time left, but this would have surpassed the scope of a master's thesis.

6. CONCLUSION

It is beneficial in continuous integration, that building and testing a software happens as quickly as possible. Because CI aims to provide rapid feedback to the developers, slow testing can be harmful [16]. As softwares evolve, the test suites become large and at some point, they can no longer be executed in a short time. We tried to find ways to enhance or speed up testing in order to facilitate CI. We found out, that test case selection techniques can be used to reduce the time required for testing. Test case prioritization also helps to decrease the time for getting feedback from test executions. We found incremental machine learning especially interesting for its capability to eventually outperform comparison heuristics.

We used incremental machine learning to predict failing validation tests out of the test suite using information such as test history, code coverage, and modifications introduced in a commit. With these predictions, we were able to effectively select a small number of test cases for execution when a new commit was made to the software repository. We found out, that the incremental machine learning based test case selection techniques eventually perform equally well or better than the best heuristic. Similar results have already suggested by Spieker et al. [40], who used reinforcement learning and neural networks to select a subset of tests based on test history. Their technique required 60 consecutive CI cycles to perform equally well or better than the comparison techniques in NAPFD values. Our evaluation was based on the MCC score, and the ML techniques produced equal or better MCC scores than the best heuristic after 20 source code modifying commits. Our results support the results of Spieker et al. and brings in more evidence that when initial training data does not exist, machine learning can be applied incrementally to eventually produce as good or better results as comparison techniques. In addition to that, our results give a cautious hint, that accumulating training data and re-training the models in every iteration, using more features such as code coverage and similarity score and using a different classifier, e.g. random forest, can make the models learn faster and predict failing tests correctly earlier.

We compared a number of test case prioritization techniques. Half of the techniques were based on incremental machine learning and the other half on code coverage. A random technique was also included. The coverage based techniques have already been studied in the past [11][36]. We found out, that the incremental learning based techniques outperform rest of the techniques in APFD values. The machine learning techniques produced significantly better APFD values compared to the coverage and random techniques in the range of 1-35 source modifying commits. Only the multilayer perceptron did not perform better than the comparison heuristics but required a longer training period to catch up with a

similar performance to other techniques. Towards the end, multilayer perceptron produced almost equally good APFD values with the rest machine learning techniques. However, the results suggest that incremental machine learning has a lot of potential in test case prioritization.

We explored how transitive dependency selection affects test suite reduction when a commit modifies tests or any of their ”use dependencies”. Transitive dependency selection produced an average test suite reduction of $78,8\%$ over 347 test modifying commits. This shows that transitive dependency selection can reduce test suite substantially.

Despite our positive results in favor of using machine learning in test case selection and prioritization, we think that our results need further verification. We used only a single software project in our case study, and therefore external validity is risked. Secondly, we fully re-train the machine learning models in every commit, which can become infeasible when the training data increases. Thirdly, our experiment setup was complex because of how we treated commits that modify *src/** and *test/** directories. We have plans to continue the investigation of incremental learning in test case selection and prioritization at *SSF*, and we are going to pay attention to these issues in the future.

REFERENCES

[1] E. Alpaydin, Introduction to Machine Learning, 2nd ed., The MIT Press, 2010.

[2] P. Ammann, J. Offutt, Introduction to Software Testing, 1st ed., Cambridge University Press, New York, NY, USA, 2008.

[3] C.S. Arapidis, Sonar Code Quality Testing Essentials, Packt Publishing Ltd, Olton, 2012. ID: 1019538. Available: http://ebookcentral.proquest.com/lib/tut/detail.action?docID=1019538

[4] O. Banias, The drawbacks of statement code coverage test case prioritization related to domain testing, in: 2016 IEEE 11th International Symposium on Applied Computational Intelligence and Informatics (SACI), May, 2016, pp. 221–224.

[5] R. Beena, S. Sarala, Code Coverage Based Test Case Selection and Prioritization, ArXiv e-prints, Dec. 2013.

[6] Á. Beszédes, T. Gergely, L. Schrettner, J. Jász, L. Langó, T. Gyimóthy, Code coverage-based regression test selection and prioritization in webkit, in: 2012 28th IEEE International Conference on Software Maintenance (ICSM), Sept, 2012, pp. 46–55.

[7] S. Biswas, R. Mall, M. Satpathy, S. Sukumaran, Regression test selection techniques: A survey, Informatica (Ljubljana), Vol. 35, Jan. 2011.

[8] S. Boughorbel, F. Jarray, M. El-Anbari, Optimal classifier for imbalanced data using matthews correlation coefficient metric, PLOS ONE, Vol. 12, Iss. 6, June 2017, pp. 1–17.

[9] L. Breiman, Random forests, Machine Learning, Vol. 45, Iss. 1, Oct, 2001, pp. 5–32.

[10] B. Busjaeger, T. Xie, Learning for test prioritization: An industrial case study, in: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, New York, NY, USA, 2016, ACM, FSE 2016, Seattle, WA, USA, pp. 975–980.

[11] D. Di Nardo, N. Alshahwan, L. Briand, Y. Labiche, Coverage-based regression test case selection, minimization and prioritization: a case study on an industrial system, Software Testing, Verification and Reliability, Vol. 25, Iss. 4, pp. 371–396.

[12] E. Engström, P. Runeson, M. Skoglund, A systematic review on regression test selection techniques, Information and Software Technology, Vol. 52, Iss. 1, 2010, pp. 14 – 30.

[13] Eradicating non-determinism in tests, https://martinfowler.com/articles/nonDeterminism.html. Accessed: 2019-02-13.

[14] T. Fawcett, An introduction to roc analysis, Pattern Recognition Letters, Vol. 27, Iss. 8, ROC Analysis in Pattern Recognition, 2006, pp. 861 – 874.

[15] M.M. Fawzy, M.S. El-Mahallawy, H. El-Deeb, Enhanced code coverage approach for regression testing, in: 2015 International Conference on Control, Instrumentation, Communication and Computational Technologies (ICCICCT), Dec, 2015, pp. 438–442.

[16] M. Fowler, M. Foemmel, Continuous integration, Thought-Works) http://www.thoughtworks. com/Continuous Integration. pdf, Vol. 122, 2006, p. 14.

[17] M. Gligoric, L. Eloussi, D. Marinov, Ekstazi: Lightweight test selection, in: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, May, 2015, Vol. 2, pp. 713–716.

[18] M.J. Harrold, R. Gupta, M.L. Soffa, A methodology for controlling the size of a test suite, in: Proceedings. Conference on Software Maintenance 1990, Nov, 1990, pp. 302–310.

[19] M.J. Harrold, J.A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S.A. Spoon, A. Gujarathi, Regression test selection for java software, SIGPLAN Not., Vol. 36, Iss. 11, Oct. 2001, pp. 312–326.

[20] M. Hilton, T. Tunnell, K. Huang, D. Marinov, D. Dig, Usage, costs, and benefits of continuous integration in open-source projects, in: 2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), Sep., 2016, pp. 426–437.

[21] J. Humble, J. Molesky, Why enterprises must adopt devops to enable continuous delivery, Vol. 24, Aug. 2011, pp. 6–12.

[22] E. Knauss, M. Staron, W. Meding, O. Söder, A. Nilsson, M. Castell, Supporting continuous integration by code-churn based test selection, in: 2015 IEEE/ACM 2nd International Workshop on Rapid Continuous Software Engineering, May, 2015, pp. 19–25.

[23] R. Lachmann, Machine learning-driven test case prioritization approaches for black-box software testing, in: European Test and Telemetry Conference ettc2018, Jun, 2018, pp. 300–309.

[24] R. Lachmann, S. Schulze, M. Nieke, C. Seidl, I. Schaefer, System-level test case prioritization using machine learning, in: 2016 15th IEEE International Conference on Machine Learning and Applications (ICMLA), Dec, 2016, pp. 361–368.

[25] B. Lakshminarayanan, D.M. Roy, Y. Whye Teh, Mondrian Forests: Efficient Online Random Forests, arXiv e-prints, June 2014, p. arXiv:1406.2673.

[26] H. Li, Learning to Rank for Information Retrieval and Natural Language Processing, Second Edition, Vol. 4, Apr. 2011.

[27] C.D. Manning, P. Raghavan, H. Schütze, Introduction to Information Retrieval, Cambridge University Press, New York, NY, USA, 2008.

[28] B. Matthews, Comparison of the predicted and observed secondary structure of t4 phage lysozyme, Biochimica et Biophysica Acta (BBA) - Protein Structure, Vol. 405, Iss. 2, 1975, pp. 442 – 451.

[29] T.M. Mitchell, Machine Learning, 1st ed., McGraw-Hill, Inc., New York, NY, USA, 1997.

[30] G.J. Myers, C. Sandler, The Art of Software Testing, John Wiley & Sons, Inc., USA, 2004.

[31] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay, Scikit-learn: Machine learning in Python, Journal of Machine Learning Research, Vol. 12, 2011, pp. 2825–2830.

[32] D. Powers, Evaluation: From precision, recall and f-factor to roc, informedness, markedness & correlation, Mach. Learn. Technol., Vol. 2, Jan. 2008.

[33] G. Rothermel, M.J. Harrold, Analyzing regression test selection techniques, IEEE Transactions on Software Engineering, Vol. 22, Iss. 8, Aug, 1996, pp. 529–551.

[34] G. Rothermel, M.J. Harrold, J. Ostrin, C. Hong, An empirical study of the effects of minimization on the fault detection capabilities of test suites, in: Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272), Nov, 1998, pp. 34–43.

[35] G. Rothermel, R.H. Untch, C. Chu, M.J. Harrold, Test case prioritization: an empirical study, in: Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for Business Change' (Cat. No.99CB36360), Aug, 1999, pp. 179–188.

[36] G. Rothermel, R.H. Untch, C. Chu, M.J. Harrold, Prioritizing test cases for regression testing, IEEE Transactions on Software Engineering, Vol. 27, Iss. 10, Oct, 2001, pp. 929–948.

[37] A. Saffari, C. Leistner, J. Santner, M. Godec, H. Bischof, On-line random forests, in: 2009 IEEE 12th International Conference on Computer Vision Workshops, ICCV Workshops, Sep., 2009, pp. 1393–1400.

[38] M. Shahin, M.A. Babar, L. Zhu, Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices, IEEE Access, Vol. 5, 2017, pp. 3909–3943.

[39] M. Sokolova, G. Lapalme, A systematic analysis of performance measures for classification tasks, Information Processing & Management, Vol. 45, Iss. 4, 2009, pp. 427 – 437.

[40] H. Spieker, A. Gotlieb, D. Marijan, M. Mossige, Reinforcement learning for automatic test case prioritization and selection in continuous integration, in: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, New York, NY, USA, 2017, ACM, ISSTA 2017, Santa Barbara, CA, USA, pp. 12–22.

[41] P. Tonella, P. Avesani, A. Susi, Using the case-based ranking methodology for test case prioritization, in: 2006 22nd IEEE International Conference on Software Maintenance, Sep., 2006, pp. 123–133.

[42] M. Virmani, Understanding devops amp; bridging the gap from continuous integration to continuous delivery, in: Fifth International Conference on the Innovative Computing Technology (INTECH 2015), May, 2015, pp. 78–82.

[43] W.E. Wong, J.R. Horgan, S. London, A.P. Mathur, Effect of test set minimization on fault detection effectiveness, in: Proceedings of the 17th International Conference on Software Engineering, New York, NY, USA, 1995, ACM, ICSE '95, Seattle, Washington, USA, pp. 41–50.

[44] S. Yoo, M. Harman, Regression testing minimization, selection and prioritization: A survey, Softw. Test. Verif. Reliab., Vol. 22, Iss. 2, Mar. 2012, pp. 67–120.

[45] S. Yoo, R. Nilsson, M. Harman, Faster fault finding at google using multi objective regression test optimisation, in: European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'11), 2011.

[46] Y. Zhao, A. Serebrenik, Y. Zhou, V. Filkov, B. Vasilescu, The impact of continuous integration on other software development practices: A large-scale empirical

study, in: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), Oct, 2017, pp. 60–71.

[47] H. Zhu, P.A.V. Hall, J.H.R. May, Software unit test coverage and adequacy, ACM Comput. Surv., Vol. 29, Iss. 4, Dec. 1997, pp. 366–427.

APPENDIX A: CONFUSION MATRICES

Figures A.1 and A.2 show confusion matrices for every test case selection technique.

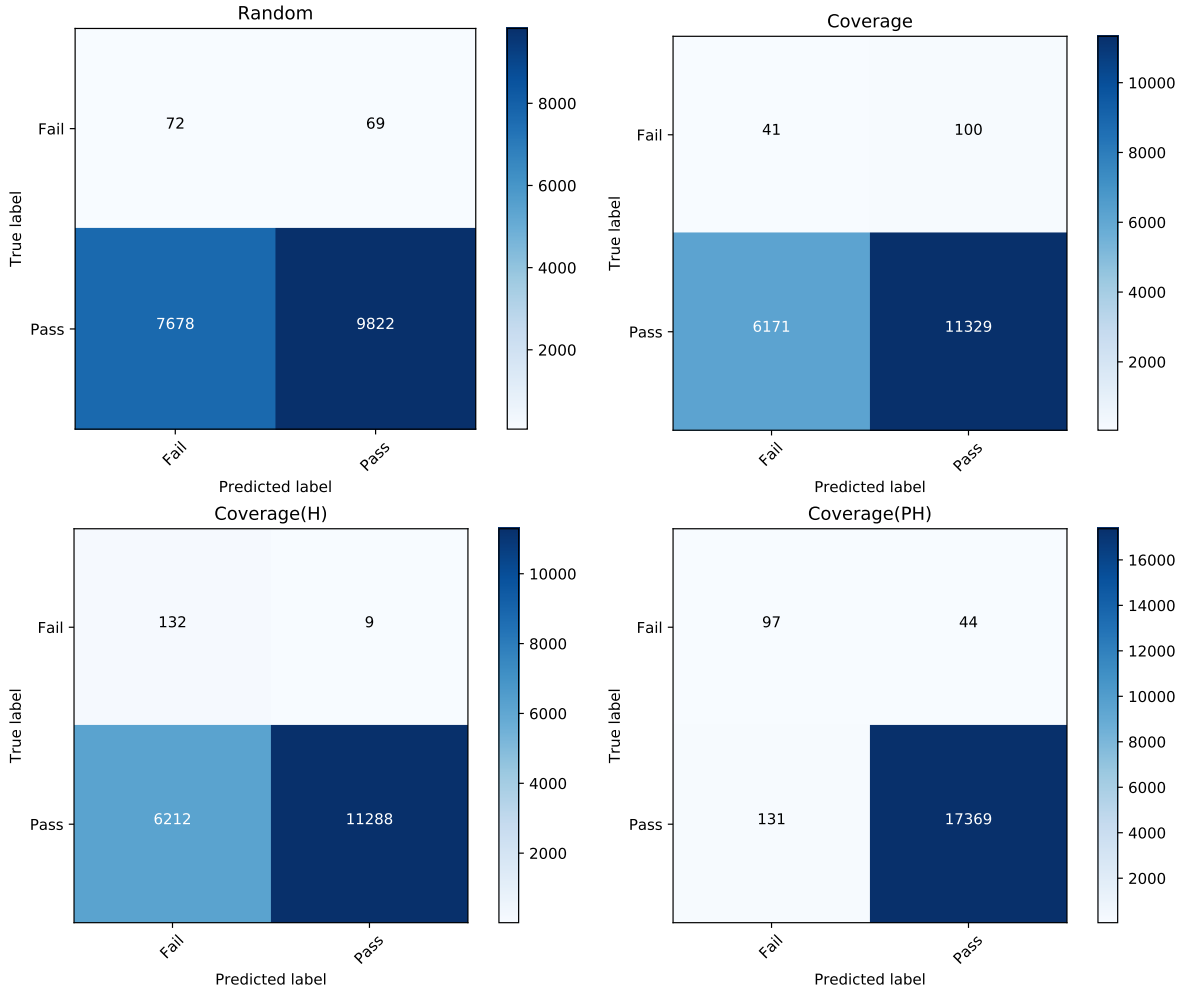

Figure A.1. *Summed confusion matrices of coverage based test case selection methods over 35 commits.*

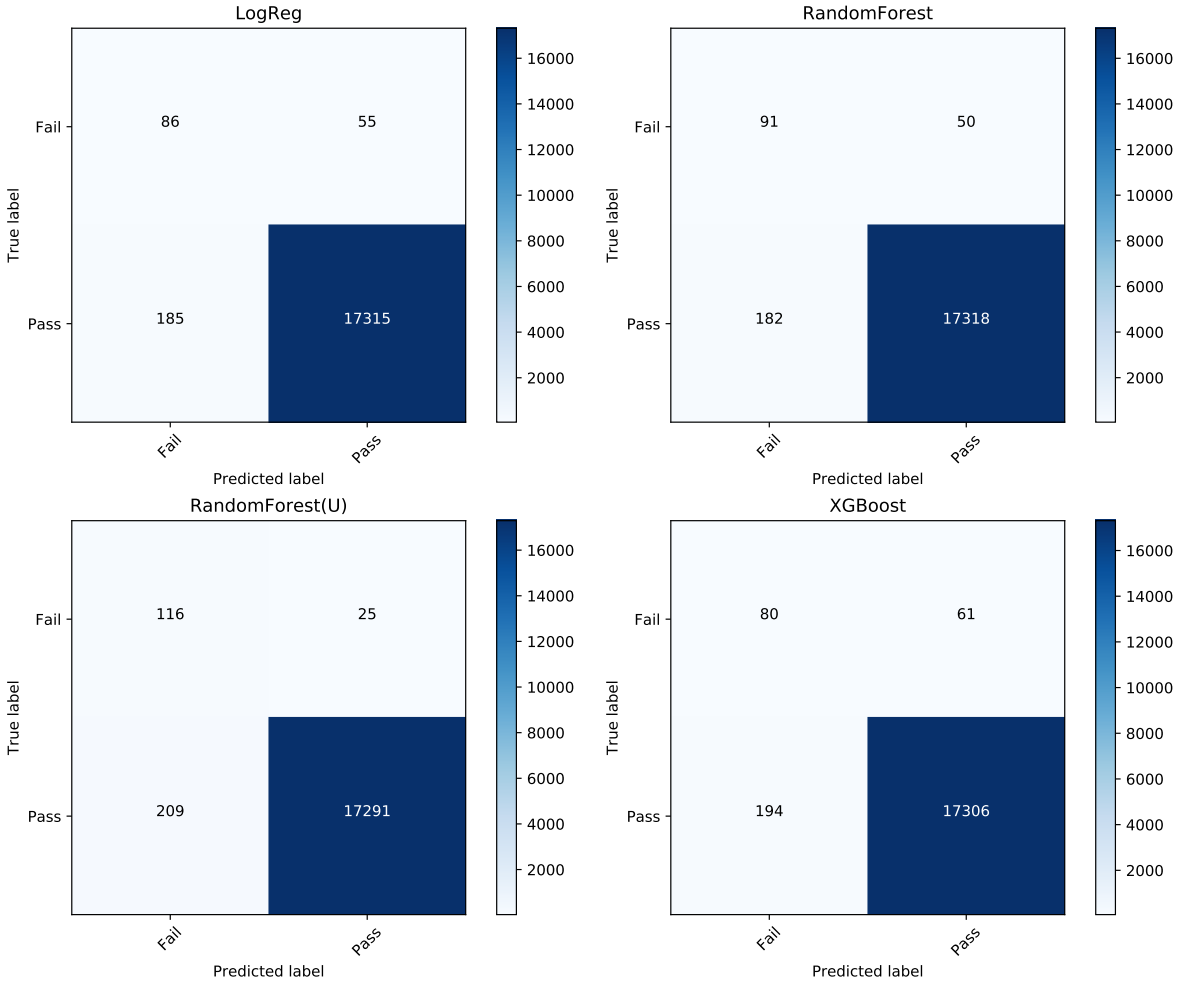

Figure A.2. *Summed confusion matrices of machine learning based test case selection methods over 35 commits.*

APPENDIX B: HISTOGRAMS

Figures B.1 and B.2 show the relation of recall and test suite reduction for every test case selection technique in 35 source code modifying commits. One dim square in the plot corresponds to a single commit. The color intensity explains the number of commits. The best scenario is where recall is close to 1, and test suite reduction is close to 1. For example, the *Coverage(PH)* technique has eighteen such commits. We wish to see as many commits as possible in the upper-right corner of the 2d histogram.

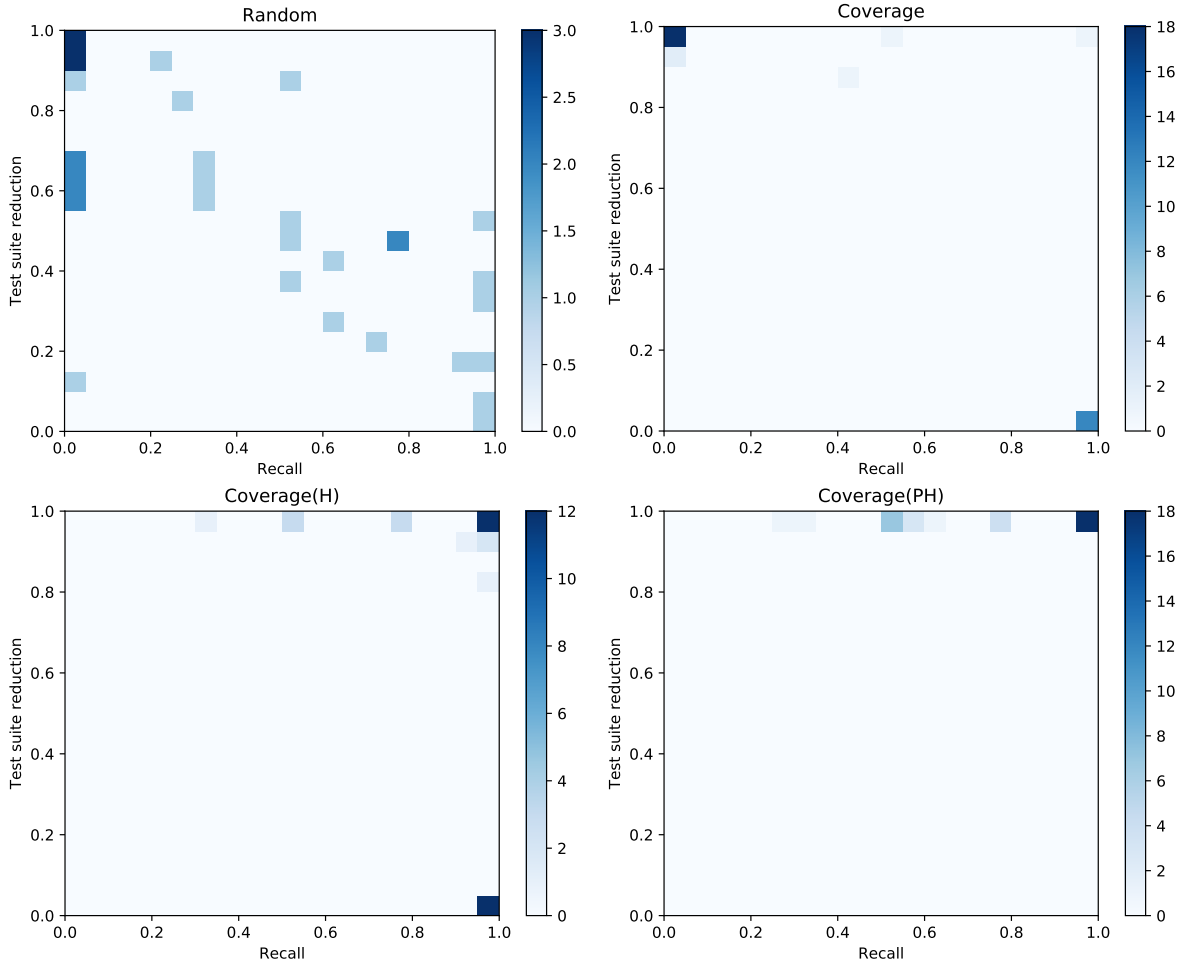

Figure B.1. *Recall and test suite reduction of the coverage based test case selection techniques.*

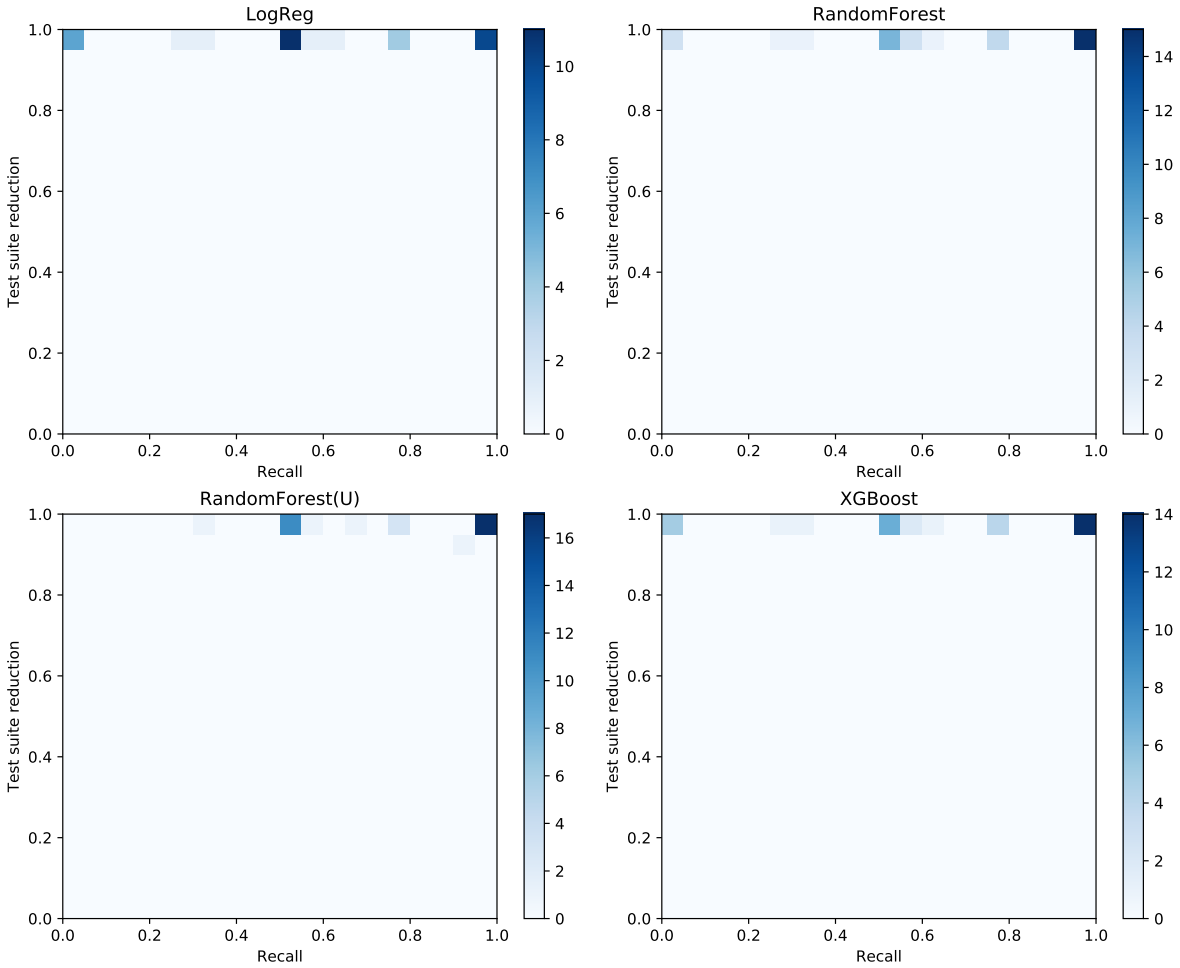

Figure B.2. *Recall and test suite reduction of the machine learning test case selection techniques.*