Learning from Failure: A History-based, Lightweight Test Prioritization Technique Connecting Software Changes to Test Failures

Falco Dürsch, Patrick Rein, Toni Mattis,
Robert Hirschfeld

Technische Berichte Nr. 145

des Hasso-Plattner-Instituts für
Digital Engineering an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für
Digital Engineering an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für
Digital Engineering an der Universität Potsdam | 145

Falco Dürsch | Patrick Rein | Toni Mattis | Robert Hirschfeld

Learning from Failure: A History-based, Lightweight Test Prioritization Technique Connecting Software Changes to Test Failures

Universitätsverlag Potsdam

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available on the Internet via http://dnb.dnb.de/.

Universitätsverlag Potsdam 2022
http://verlag.ub.uni-potsdam.de/

Am Neuen Palais 10, 14469 Potsdam
Phone: +49 (0)331 977 2533 / Fax: 2292
Email: verlag@uni-potsdam.de

The series **Technische Berichte des Hasso-Plattner-Instituts für Digital Engineering an der Universität Potsdam** is edited by the professors of the Hasso Plattner Institute for Digital Engineering at the University of Potsdam.

ISSN (print) 1613-5652
ISSN (online) 2191-1665

This work is protected by copyright.
Layout: Tobias Pape
Print: docupoint GmbH Magdeburg

ISBN 978-3-86956-528-6

Also published online on the publication server of the University of Potsdam:
https://doi.org/10.25932/publishup-53755
https://nbn-resolving.org/urn:nbn:de:kobv:517-opus4-537554

Regression testing is a widespread practice in today's software industry to ensure software product quality. Developers derive a set of test cases and execute them frequently to ensure that their change did not adversely affect existing functionality. As the software product and its test suite grow, the time to feedback during regression test sessions increases, and impedes programmer productivity: Developers wait longer for tests to complete and delays in fault detection render fault removal increasingly difficult.

Test case prioritization addresses the problem of long feedback loops by reordering test cases, such that test cases of high failure probability run first, and test case failures become actionable early in the testing process. We ask, given test execution schedules reconstructed from publicly available data, to which extent can their fault detection efficiency be improved, and which technique yields the most efficient test schedules with respect to APFD?

To this end, we recover 6,200 regression test sessions from the build log files of TRAVIS CI, a popular continuous integration service, and gather 62,000 accompanying change lists. We evaluate the efficiency of current test schedules and examine the prioritization results of state-of-the-art lightweight, history-based heuristics. We propose and evaluate a novel set of prioritization algorithms, which connect software changes and test failures in a matrix-like data structure.

Our studies indicate that the optimization potential is substantial because the existing test plans score only 30 % APFD. The predictive power of past test failures proves to be outstanding: simple heuristics, such as repeating tests with failures in recent sessions, result in efficiency scores of 95 % APFD. The best-performing matrix-based heuristic achieves a similar score of 92.5 % APFD. In contrast to prior approaches, we argue that matrix-based techniques are useful beyond the scope of effective prioritization and enable a number of use cases involving software maintenance.

We validate our findings from continuous integration processes by extending a continuous testing tool within development environments with means of test prioritization and pose further research questions. We think that our findings are suited to propel the adoption of (continuous) testing practices and that programmers' toolboxes should contain test prioritization as an existential productivity tool.

Regressionstests sind in der heutigen Softwareindustrie eine weit verbreitete Praxis, um die Qualität eines Softwareprodukts abzusichern. Dabei leiten Entwickler von den gestellten Anforderungen Testfälle ab und führen diese wiederholt aus, um sicherzustellen, dass ihre Änderungen die bereits existierende Funktionalität nicht negativ beeinträchtigen. Steigt die Größe und Komplexität der Software und ihrer Testsuite, so wird die Feedbackschleife der Testausführungen länger und mindert die Produktivität der Entwickler: Sie warten länger auf das Testergebnis und die Fehlerbehebung gestaltet sich umso schwieriger, je länger die Ursache zurückliegt.

Um die Feedbackschleife zu verkürzen, ändern Testpriorisierungs-Algorithmen die Reihenfolge der Testfälle, sodass Testfälle, die mit hoher Wahrscheinlichkeit fehlschlagen, zuerst ausgeführt werden. Der vorliegende Bericht beschäftigt sich mit der Frage nach der Effizienz von Testplänen, welche aus öffentlich einsehbaren Daten rekonstruierbar sind und damit, welche anwendbaren Priorisierungs-Techniken die effizienteste Testreihenfolge in Bezug auf APFD hervorbringen.

Zu diesem Zweck werden 6,200 Testsitzungen aus den Logdateien von TRAVIS CI, einem oft verwendeten Dienst für Continuous Integration, und über 62,000 Änderungslisten rekonstruiert. Auf dieser Grundlage wird die Effizienz der derzeitigen Testpläne bewertet, als auch solcher, die aus der Neupriorisierung durch leichtgewichtige, verlaufsbasierte Algorithmen hervorgehen. Zudem schlägt der vorliegende Bericht eine neue Gruppe von Ansätzen vor, die Testfehlschläge und Softwareänderungen mit Hilfe einer Matrix in Bezug setzt.

Da die beobachteten Testreihenfolgen nur 30 % APFD erzielen, liegt wesentliches Potential für Optimierung vor. Dabei besticht die Vorhersagekraft der unmittelbar vorangegangen Testfehlschläge: Einfache Heuristiken wie das Wiederholen von Tests, welche kürzlich fehlgeschlagen sind, führen zu Testplänen mit einer Effizienz von 95 % APFD. Matrix-basierte Ansätze erreichen eine Fehlererkennungsrate von bis zu 92.5 % APFD. Im Gegensatz zu den bisher bekannten Ansätzen sind die matrix-basierten Techniken auch über den Zweck der Testpriorisierung hinaus nützlich und in der Softwarewartung anwendbar.

Zusätzlich werden die Ergebnisse der vorliegenden Studie für Continuous Integration Systeme im Kontext integrierter Entwicklungsumgebungen validiert, indem ein Tool für Continuous Testing um Testpriorisierung erweitert wird. Dies führt zu neuen Forschungsfragen. Die Untersuchungsergebnisse sind geeignet die Einführung von Continuous Testing zu befördern und untermauern, dass Werkzeuge der Testpriorisierung für produktive Softwareentwicklung essenziell sind.

Contents

1. Regression Testing in Continuous Integration Environments **14**

2. Background and Related Work **20**
2.1. Regression Test Prioritization 20
2.2. Regression Test Selection 23
2.3. Continuous Testing 23
2.4. Summary 24

3. Matrix-based RTP **25**
3.1. Research Gap 25
3.2. File-to-Failure Matrix 27
3.3. Prioritization Algorithms 31

4. Dataset **36**
4.1. Continuous Integration at Travis CI 37
4.2. Build Linearization and Learning 37
4.3. Recovering Test Execution Histories from Public Data 38
4.4. Changesets 41

5. Evaluation **42**
5.1. Method 42
5.2. Regression Testing on Open-source Software Projects 47
5.3. Baseline Performance 52
5.4. Matrix Prioritization Performance 52
5.5. Discussion 60
5.6. Threats to Validity 62

6. Exploratory Study on Test Prioritization for Continuous Testing **65**
6.1. Introduction to *AutoTDD* 65
6.2. Extensions of *AutoTDD* 66
6.3. Proposed Research Directions 66

7. Future Work **69**
7.1. Extensions to Matrix-based Test Prioritization 69
7.2. Software Development Process Improvements 70
7.3. Test Maintenance Support 71

8. Conclusion **73**

A. Source Code **75**

Bibliography **81**

List of Figures

1.1. Branch-based Development Workflow 15
1.2. Build Failure Rate of Merge Requests 17
1.3. Relative Failure Rate Given Merge Request Size 17

2.1. Regression Test Selection versus Prioritization 20

3.1. Approach Overview 26
3.2. File-to-failure Matrix Example 28
3.3. Matrix Schema 29
3.4. Combining Failure Matrices 30
3.5. Matrix to Decision Tree Conversion 32
3.6. Matrix of the Running Example 34
3.7. Conditional Probability Example 35

4.1. Dataset Schematic 36
4.2. Build and Job Schematic 38

5.1. Example APFD Plots 44
5.2. Comparison of Untreated to Optimal Test Ordering 50
5.3. Detailed APFD Trend for all Projects 51
5.4. Baseline Performance 56
5.5. Failure History Gantt Chart 57
5.6. Matrix Performance 58
5.7. Sketch of Historic Relevance 58
5.8. Fault Detection Efficiency on Repeated Test Failures 59
5.9. Failure Distance Distribution 59

6.1. *AutoTDD* 65
6.2. Prioritization Performance Visualization 66

8.1. Perceived Immediacy during Regression Test 73

List of Tables

3.1. Weighting Function Overview 31

4.1. Recovered Test Attributes 39
4.2. Study Subject Statistics 40
4.3. Changeset Statistics 41

5.2. Comparison of Optimal Schedules 49

6.1. Prioritization Performance of *AutoTDD* 67

Listings

5.1. Prioritization Strategy Trait and Parameter Object. 47

A.1. Recently Failed 75
A.2. Matrix Data Structure 76
A.3. Unit Matrix Construction 76
A.4. Reducer: Combining Matrices 77
A.5. Strategy: *conditional-prob* 78
A.6. Strategy: *recently-changed* 79
A.7. Strategy: *file-similarity* 80

1. Regression Testing in Continuous Integration Environments

Regression testing is a widespread technique in today's software industry to ensure software product quality. A set of test cases is derived from an initial set of requirements, which exercise the software system with known inputs and check for correct outputs. As software development carries on, re-execution of all test cases prevents developers from introducing faults in existing functionality, called regressions. Therefore, the size of the test set can be assumed to be ever increasing: As new requirements arrive, new test cases are added, or as developers identify error conditions (bugs), they introduce additional checks for corner cases. Typically, software developers are hesitant to remove test cases, as they fear future detrimental changes may go undetected, and product managers rarely decrease the software's feature set significantly.

To catch detrimental changes, or out-of-date test oracles[1], development teams increasingly rely on the software development practice of Continuous Integration (CI) [16]. Within this setting, programmers integrate code changes frequently (up to several times a day) by creating a commit in the version control system and publishing it to a central repository. From there on, a CI server, which is oftentimes a separate machine or can be a third-party service, runs through the entire build and verification process of the software, which includes execution of the regression test suite. In many development processes, the failure to complete a CI cycle represents an obstacle to integrating the current set of changes into mainline development, also known as pre-merge check. This way, the check sets a minimum quality bar for proposed changes, which are then required to be syntactically correct, to pass all tests, and to cope with changes in the program's environment.

Software development teams often benefit from an established CI process, because the necessary infrastructure and the required degree of (test) automation shorten release cycles or the time to fault localization [25]. While the initial CI setup might start small and builds may last only several minutes, build times increase as the project gains in complexity and size. Subsequently, developer productivity is at risk because programmers require more time to integrate their changes, even if those are of minor nature.

A common development model for making use of CI is built around version control branches (see Figure 1.1). Within this model, developers require a CI build,

[1] A *test oracle* describes a mechanism to determine whether a test case passed or failed. The techniques range from hard-coding known outputs to verifying system properties through comparing two competing implementations, for example.

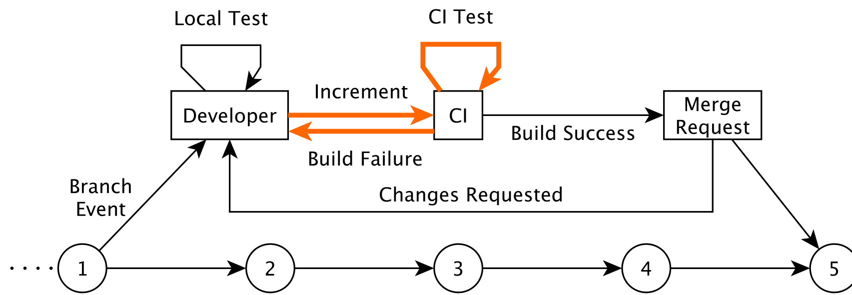


Figure 1.1.: In a branch-based development workflow, developers fork the development mainline and spent multiple iterations on a single increment. They validate their changes using a CI through the main feedback loop, highlighted in bold.

before their code change can enter the review phase. While a single green build would in principle suffice to integrate the change in the main line of development, in reality, this feedback loop is entered many times over. When developers engage in pre-submit testing, they may be confronted with an unstable build process, integration of recent changes from the trunk, or several (possibly mutually dependent) program faults. Typically, addressing any of these issues requires restarting the build process from the very beginning. As a result of increased build times, developers spend a larger fraction of their time waiting for the CI build to complete, as even very minor modifications undergo the same rigid build process. Eventually, developers might start multi-tasking and begin to work on other features or bugs. However, due to the mental overhead of switching between tasks and the natural upper limit to the capability to handle different tasks in parallel, this is poor compensation for long build times. To worsen the problem, we observe a tendency to skip local testing altogether and to increasingly rely on CI systems for most of integration tasks [2]. In tandem with long build times and frequent builds due to high code churn, CI services are subject to congestion [14, 37]. In consequence, the time to feedback not only depends on the build duration but also needs to factor in the size of the build backlog. In summary, we make the case that even marginal increases in build time can negatively impact the efficiency of software development.

For these reasons, a desirable CI workflow entails a build time that is as short as possible to maintain developer productivity. However, speedy software builds do not appear to be the norm since 90 % of study respondents report actively spending time on reducing build time [24]. While only experts can optimize general infrastructure, such as compilers or dependency management tools, product-specific test suites resemble areas of great control to software engineers. Despite the fact that programmers can easily change them, test suites grow in size, duration, and complexity. The fact that most test automation tools employ the “retest all” strategy, in which all tests execute in a fixed order regardless of their fault detection efficiency, diminishes the chances of early feedback.

As a result of this development model, raised test cost, and growing code churn, large software organizations increasingly focus on optimization of verification efforts as a means to achieve faster feedback and shorter build times [9, 29, 35, 56]. For instance, they invest into *regression test selection* techniques. Based on program analysis, an automated system schedules only the subset of test cases that traverse the changed code for execution and therefore all newly introduced faults. Alternatively, those organizations implement means of *regression test prioritization* that prioritize test cases of higher failure probability, which therefore run first. Either way, the time required to inform developers of failed test cases, which mandate fixing production code or adjusting the test oracle, shortens.

The topic of resource congestion and prolonged feedback cycles does not only affect large software companies but also extends to open-source projects. A substantial build overhead may negatively impact the time required to deliver a critical bug fix; can frustrate existing developers, which oftentimes participate in their spare time; or discourage new contributors. We find evidence of long build times in open-source projects in a preliminary study of a dataset of a popular continuous integration service [4]. Up to two-thirds of builds fail to meet the rule of the ten-minute build [16] and their total time exceeds 40 minutes on average. In accordance, reports of larger organizations, testing also contributes significantly to prolonged feedback loops. On average, open source projects dedicate 40 % of build time to testing activities. Also, we quantify the impact of checking a code change using a test suite and a CI system, and find that 11.8 % of all merge requests are affected by test failures.

Getting hold of these test failures as soon as possible appears critical, because they severely impede integration of the code change in many instances. Within a merge request that possesses at least one test failure and one successful build during the time frame of the observation, Figure 1.2 shows that 50 to 55 % of builds fail due to tests, with another peak at 30 to 35 % percent. For individual projects with a focus on test-driven development, up to two third of merge request builds fail due to tests. Also, Figure 1.3 shows that the failure rate is independent of the length of the merge request. Apparently, while still working on the same feature, developers tend to continuously break tests, as they try to make the code compliant to the feedback of the reviewer. In summary, the test suite is a suitable vantage point for optimizing the build process and for shortening the feedback cycle, such that developers can proceed to fix faults earlier in the testing process.

In response to the above constraints and challenges, we propose to deepen the understanding of regression testing in a continuous integration environment by studying open-source software and the extent to which test case prioritization affects small- to medium-sized projects. To this end, we require a new dataset of test execution histories of open-source projects to determine which existing prioritization heuristics cope well with the projects' diversity. Additionally, we investigate if extending existing heuristics with further inputs provides better prioritization performance.

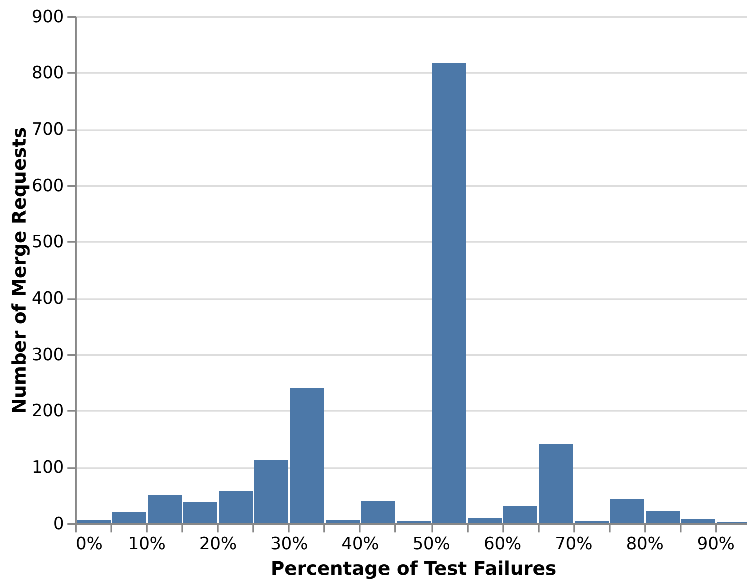


Figure 1.2.: On average, half of the builds of a merge request break due to test failures

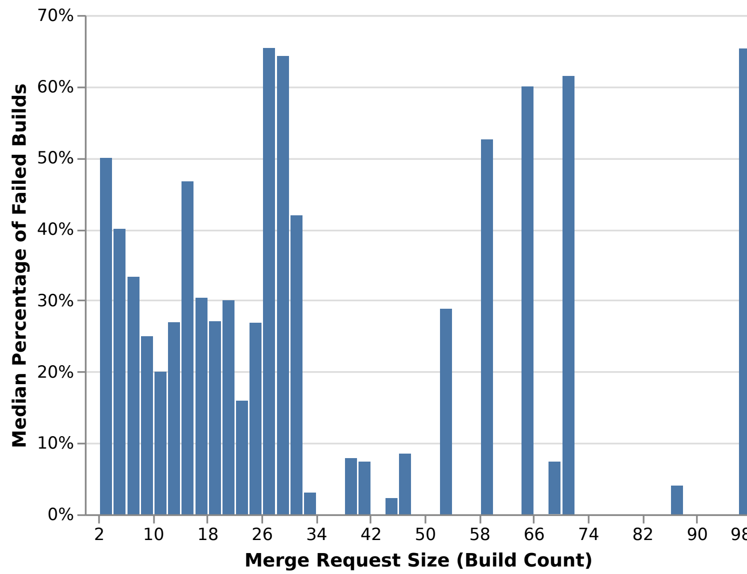


Figure 1.3.: Contrary to our expectation, longer merge requests (in terms of build count) sustain similar build failure rates

List of Contributions We make the following contributions to the field of regression test prioritization, listed in no particular order:

1. *Dataset* To study a diverse set of projects, we present a newly derived dataset of more than 6,000 regression test sessions from twenty open-source projects. To this end, we analyze log files of TRAVIS CI, a popular continuous integration service, for traces of test execution and recover over 62,000 accompanying change lists. This real-world dataset closes the gap between sets of small, diverse academic study subjects and industry-scale, homogeneous software development artifacts.

2. *Baseline* To shed light on the state of regression testing on open-source projects, we extensively evaluate prior work on the novel dataset. For reproducibility, we investigate the performance of the observed test order and three more baseline techniques of earlier works in the field of history-based regression testing. To further quantify the potential impact of cost-aware test prioritization, we compare test plans of different optimality criteria: detected faults per test case run and detected faults per cost.

3. *Matrix-based Regression Test Prioritization* We extend prior history-based techniques to include the current change of the software revision as the input signal. To this end, we connect software changes to test failures in a matrix-like data structure. This approach compensates the static prediction, a major shortcoming of existing heuristics, which assign test priorities regardless of the software change. We report on the prioritization performance of one algorithm using a probabilistic interpretation and another set of techniques consisting of four different weighting approaches. Our approaches perform on par with respect to the best-performing baseline algorithm. Also, the relative prioritization performance of matrix-based approaches increases, as fault repetitions spread over time. We emphasize that matrix-based models are useful beyond the scope of test prioritization and that change-to-failure-matrices enable a number of use cases involving test maintenance.

4. *Exploratory Study* We validate our research in the practical context of *AutoTDD*, a continuous testing system for Squeak/Smalltalk. We corroborate that findings from both, prior work and our approaches, can indeed be adapted from a continuous integration setting to one within a development environment. Our preliminary experiments outline the importance of test prioritization to continuous testing and pose further research questions.

Structure of this Report In Chapter 2, we proceed to categorize and examine relevant prior work from the fields of test prioritization and test selection. Chapter 3 describes how to connect changes and test outcomes in a matrix-like data structure and introduces novel prioritization algorithms, which utilize a probabilistic interpretation of failure events or rely on weighting observations. We continue to characterize the dataset and explain the procedure to create it in Chapter 4. Further, this chapter details how to integrate test prioritization in continuous integration workflows. Chapter 5 comprises the research method (5.1) and gives insights into three differently themed series of experiments: the state of regression testing on open-source projects (5.2), the fault detection efficiency of prior works (5.3), and the efficiency of novel matrix-based approaches (5.4). We conduct an exploratory study concerned with live testing in Chapter 6 and give an outlook to future developments and applications in Chapter 7. Chapter 8 concludes this report.

2. Background and Related Work

In the following, we outline related works and how prior elaborations influence our approach. Section 2.1 details related techniques in the field of Regression Test Prioritization (RTP) and Section 2.2 on influences from the research area of Regression Test Selection (RTS). Section 2.3 emphasizes the need for alternate test orderings in the context of continuous testing and Section 2.4 synthesizes the findings.

RTP techniques reorder test cases to maximize some metric early in the testing process, such as faults detected or code covered. While the former approaches rely on executing the entire test suite, RTS techniques select subsets of test cases to run (Figure 2.1). Both research areas closely relate to each other, because both approaches convert into each other. Trivially, every RTS approach resembles a prioritization algorithm that assigns the highest priority to the selected test cases and the lowest priority to the remaining ones. Conversely, prioritization techniques convert into test selection algorithms by selecting the *n* topmost test cases. The following section introduces related techniques from both fields, compares the needed inputs, and elaborates on the resulting non-functional properties.

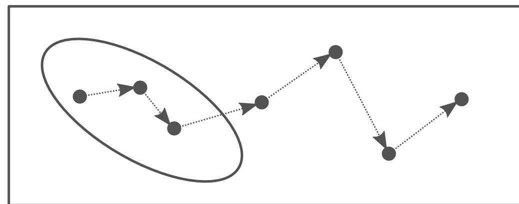


Figure 2.1.: A selection technique (oval) chooses an unordered subset of test cases (dots) to execute. Contrarily, a prioritization algorithm establishes a test ordering (arrows).

2.1. Regression Test Prioritization

The problem of shortening feedback cycles given a set of test cases to execute is known as RTP. RTP aims at reordering test cases such that the time to test failure is minimal (the tests with the maximum likelihood of detecting a failure with respect to the changes execute first). Within the RTP field, we identify three clusters of

related techniques. The first group prioritizes test cases based on a single change. The second set of approaches uses linear histories of test runs. The third adapts techniques of other computer science fields to the RTP problem, most notably from Machine Learning (ML) and Information Retrieval (IR).

Ad-hoc Program Analysis

This family of approaches considers a single change, consisting of two versions of the software, and tries to reason about how the former version transposes into the current software version to prioritize test cases. For example, previously recorded code coverage information can devise if the test case is likely to traverse the modified lines of code. Researchers proposed different strategies that reorder tests according to statement coverage, branch coverage, and Fault Exposing Potential (FEP) [44]. FEP is a mutant-based criterion, that denotes how many faults a test case can detect relative to a known number of seeded faults. The authors conclude that even simple strategies like randomized test ordering can improve fault detection efficiency and that coverage-based strategies considerably increase the rate of fault detection further. The main drawback to coverage and mutant-based prioritization is the cost of obtaining coverage data, which imposes limits on the practical applicability. Second, non-code artifacts, such as configuration files, are not part of the model of a prioritization technique-based coverage statistics and do not influence the resulting test ordering.

History-based Approaches

History-based approaches use a linearized development history to prioritize test cases, which includes multiple regression test sessions and the respective test results. Researchers propose several prioritization heuristics in the context of resource-constrained environments [27]. For example, the least-recently-used test algorithm prioritizes test cases by temporal distance to the last execution, such that all test cases execute regularly. Similarly, the least-recently-used function heuristic uses coverage data to schedule test cases, which exercise program functions that have not been run recently. Another algorithm, recently-failed, gives preference to test cases that revealed faults in recent testing sessions. Building upon these simple heuristics, more elaborate models have been proposed, which try to balance recent execution and fault detection efficiency of a test case when determining its priority over others. Additionally, the researchers address the "cold-start problem", a situation in which no or small fractions of historic data are available, and find it beneficial to seed the initial test priority with the percentage of code covered [13]. Continuing this line of work, researchers propose integrating varying test costs (most commonly modeled as test runtime) and different fault severities into history-based prioritization of test cases. Balancing these two factors shows better fault detection efficiency per cost than functional coverage test case prioritization in a controlled experiment [39].

A study of history-based approaches in continuous integration environments finds that algorithms do not need large amounts of historic data to be effective [22]. While prioritization performance increases with larger history intervals, knowledge about the test verdicts of the immediate previous revision suffices to increase fault detection efficiency over random test ordering. To lower the integration cost of software patches, researchers devise a hybrid approach that consists of two stages: In the 'pre-submit' phase, a cost-effective subset of test cases is selected for execution; in the subsequent 'post-submit' stage, test cases run in a prioritized order to get the complete picture [9]. In both instances, the algorithm uses test-case-specific historic data – the time since the last failure, and the time since execution – to prioritize or select tests. The authors conclude that the hybrid approach is efficient at lowering the integration cost and propose to automatically extract the respective threshold values from the historic data in a future iteration of their approach.

Transfer Approaches and Machine Learning

Novel techniques have been proposed that adapt algorithms and models from other fields of computer science to the RTP problem. Advances in information retrieval power REPIR, a system which treats all test cases as a document repository and equates changes in program text[1] with a document query. As a result, test cases with semantic overlap with respect to the change gain higher priority than those with little or no shared vocabulary [48]. A similar approach leverages vocabulary previously associated with test failures [31]. Researchers built a recommender system that combines telemetry of risky software components and exposure (usage patterns) to prioritize test cases in the particular context of web applications [1]. Using reinforcement learning, the RETECS software agent rewards individual test cases or entire test schedules according to historic test performance and thus is able to propose new schedules with increased fault detection performance [49].

Interestingly, while open-source software projects seldom employ RTP techniques, major software vendors appear to increasingly invest in test prioritization (selection) to appease software quality constraints with increasing code-churn, and to streamline developer workflows. Facebook engineers give insight into the process of building their test failure predictor using boosted decision trees. The classifier contains features such as file change history and cardinality, authorship information, and distance measures with respect to modularity [29]. Researchers at Google outline the design hypotheses for their Test Automation Platform (TAP), regarding its use of dependency information, file type, and file modification frequency [35]. Test engineers at Salesforce use features concerning code coverage, text path and content similarity, failure history, as well as test age, to classify tests [5]. Other works apply ML techniques to industry-grade datasets, for example of a large telecommunication company [12] or of the app development at Spotify [38].

[1] In this case, "program text" refers to the natural language components of the source code. REPIR has been shown to be reasonably effective even when discarding program structure.

2.2. Regression Test Selection

RTS techniques select subsets of test cases for execution. The main difference to prioritization techniques are *safety* properties, that is, the technique guarantees that the subset of selected test cases contains all fault-traversing test cases. To make this type of provable statement, RTS strategies rely on analysis of program structure (compile-time static analysis) or program instrumentation (through on-the-fly recorded coverage at runtime) to "firewall" the program change [10]. Approaches specific to procedural and object-oriented programming languages exist. While some approaches are agnostic to the programming language, many are tailored to specific host environments [23, 42, 43]. For example, researchers propose EKSTAZI, a dependency miner which collects file dependencies of test cases at runtime by instrumenting I/O primitives. This approach allows performing change detection not only on (binary) code artifacts but also incorporates files of any other type, such as configuration files. EKSTAZI shows that working on file granularity, which is typically much coarser than class or method level, also results in efficient test selection [19].

2.3. Continuous Testing

Continuous integration practices describe efforts to make integration of software changes a frequent and frictionless activity. However, despite the fact that developers interact with CI systems heavily, and expect build failures on a regular basis, CI processes typically run decoupled from the development environment. As a consequence, programmers have to pause development to create new changesets, trigger remote CI processes, and wait for their completion. This results in wasted engineering time and prolongs the time from introducing faults to test execution. Continuous testing remedies this fact by continuously executing tests during development time using spare CPU cycles on the programmers' machine, and therefore by providing faster feedback than through remote processes [46]. Early studies show that continuous testing practices used when solving a programming assignment promote program correctness as well as task completion on time [45]. When working with test suites, which may grow in size and duration, test prioritization is inherently required to continuously test. Only if tests of high failure probability run first, developers can still perceive the feedback as immediate. To this end, researchers propose test prioritization within continuous testing using code coverage data [50] or code locations programmers are currently reading and editing in their development environment [34].

2.4. Summary

We continue to outline insights from the related field of regression test prioritization, regression test selection, and continuous testing.

First, program instrumentation, although effective at test prioritization, appears to be too heavy-weight for large-scale deployments, whose project size and code churn place constraints on the computational complexity of the prioritization algorithm. Using sampling for collecting coverage data may address performance concerns to a certain degree but also impedes the safety and accuracy of the resulting test plan. But even in the face of accurate coverage data, we argue that those techniques are prone to overestimate the test set. For instance, coverage-based algorithms would elect integration tests on a regular basis which reach a comparably large share of the system. This happens even though the test may not be effective in fault detection because the ability to do so spreads over a large amount of covered code.

Second, history-based prioritization heuristics address both problems. They factor in the fault detection ability of tests based on prior testing sessions, and not only because the test can reach the modified code. Also, history-based heuristics are lightweight and computationally inexpensive in comparison to coverage-based approaches. Additionally, they perform on par with more complicated ML models with respect to fault detection efficiency.

Third, we recognize the emerging need for test prioritization within continuous testing environments and think that the generally lightweight, computationally inexpensive nature of history-based heuristics caters to this need well.

Fourth, EKSTAZI shows that file granularity is sufficient for purposes of regression test prioritization. This represents a significant difference to prior works, which focus on syntactic elements and program structure. Using file granularity, the prioritization algorithm can remain independent of the host programming language and encompass other non-code artifacts, such as configuration files.

With this background, investigating history-based prioritization algorithms appears worthwhile. One concern remains to be addressed: Existing history-based approaches do not factor in the current code change, which makes their prediction static. We plan to address this issue and explain other desirable traits of an advanced history-based prioritization technique in Chapter 3.

3. Matrix-based RTP

In this chapter, we find previously unmet needs of regression testers, which are beyond the scope of effective prioritization, and introduce a novel approach to meet these. Section 3.1 explores test prioritization requirements and identifies shortcomings of existing approaches. In Section 3.2, we explore a novel approach for modeling the software change along failure histories, and Section 3.3 presents the set of novel prioritization techniques.

3.1. Research Gap

Because of the overhead due to the additional instructions, we observe that RTP based on program instrumentation is too heavyweight to deploy for use with large-scale code bases and is therefore rarely adopted in practice (Section 2). If it is necessary to exercise all tests within an instrumented test run to gather the data required to make an informed decision for the next iteration, the tests could have already been run without instrumentation at a lower cost. Instead, major industry players adopt lightweight, data-driven techniques based on machine learning, which are mainly independent of the host language, and incorporate information about modularity[1] and software version history [12, 29, 35]. On the other hand, simple prioritization heuristics perform on par with ML techniques and are much less data-intensive [38, 49]. The main drawback of existing history-based heuristics is their *static prediction*. Because those heuristics base their prioritization on an immutable history of test execution and test failure events, the ordering of test cases in the next test session is fixed. That way, the software change of the current revision (or the delta to previously graded revisions) is not relevant for test ordering (change independence).

We envision a novel RTP technique located between a simple heuristic and an ML model in Figure 3.1 and which possesses the following traits:

Efficient. Minimizing the feedback time of a test suite helps programmer productivity. Therefore, a prioritization technique is efficient if the reordered test schedule approximates the optimal test plan in which all tests with failures execute first. Utilizing a technique efficient at the task of fault detection, developers can act upon test failures earlier than without prioritization.

[1] In this context, *modularity* refers to build modularity. Independent of the implementation language, a build tool may determine metrics from a dependency graph of software artifacts.

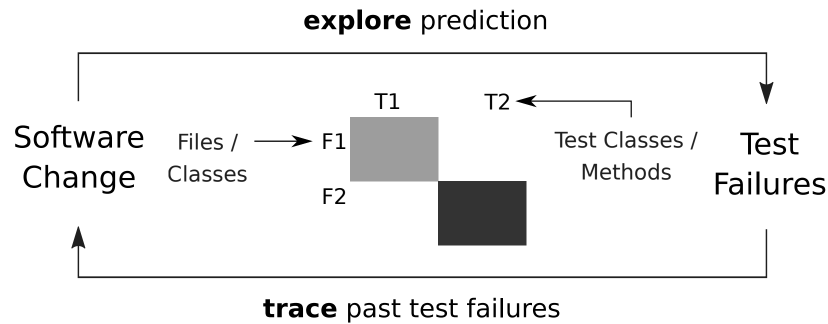


Figure 3.1.: High-level overview of the approach: Contrary to existing algorithms, our model is capable of tracing back test failures to the offending change list

Lightweight. Relying on few inputs is key to a broad adoption of a prioritization technique. Some prior techniques rely on a large volume of input signals, such as detailed authorship information, accurate dependency information, and full change history, which systems may not expose at the required level of detail. A lightweight prioritization technique relies on few, easily accessible key metrics and is portable to a wide range of build systems and programming environments.

Explorable. The data model of the technique allows for exploration, that is, developers can query the model with different inputs and estimate the impact of a software change. For a history-based heuristic, this requires adding another signal to the prediction in addition to the immutable track record of failure histories which describes the software modification. This way, the prioritization technique can be useful beyond the scope of efficient prioritization.

Traceable. An traceable prediction model allows developers to reason about prediction results and to turn (collections of) prediction results into actionable items with respect to software quality (helps developers in performing software maintenance). To our perception, machine-learning models do not cater to this need well. "After the fact" knowledge populates a large fraction of the feature space of existing ML models. While this type of knowledge facilitates a good test failure prediction using correlation, these features give little to no insight into the cause of the test failure. An exemplary feature vector of an RTP classifier [29] may read the following: This test case is more prone to breakage, because...

- the file is of type x.
- the test case is part of an actively developed (unstable) module.
- its distinct author count exceeds a certain threshold.

While the information above may reflect a useful model of reality, developers cannot infer corrective action, for example, if test cases unexpectedly become part of the set of prioritized tests. Those models give explanations only in terms of the metadata of the software modification and do not incorporate which (parts of) the

software change are relevant to the test ordering. A traceable model remedies this fact and pinpoints what made the projected test failure likely, for instance, changes to a certain class or piece of configuration.

3.2. File-to-Failure Matrix

In response to the challenges above, we propose to build a novel model connecting software changes to test failures, which uses a direct description of the change rather than a collection of derived metadata. To this end, Section 3.2.1 introduces the file-to-failure-matrix, the core data structure of the set of novel prioritization approaches, which connects test cases to file-level changes. Section 3.2.2 explains how to create a failure matrix from the results of a single regression test session and Section 3.2.3 details how to combine several observations into a single matrix.

3.2.1. Connecting File Changes to Test Case Failures

To overcome the limitations of simple heuristics and ML approaches, we propose a novel model whose primary objective consists of connecting two entities, namely files (identified by a hierarchical file path) and failure histories (modeled as a linear sequence of failure events of a particular test case). Using these two entities, we obtain the file-to-failure matrix by arranging test case failure counts in a table that assigns each distinct test case name a row and each distinct file path a column.

We consider files as first-class citizens of our model because of the following reasons: First, files are ubiquitous in modern computing and are conceptually independent of their content which may consist of source code, configuration files, or build artifacts. Second, sets of files form *units of change*. Modern version control systems almost exclusively express their commit structure (an atomic change to the software repository) in terms of sets of files that the developer changes. Third, files form *modules* because their organization in a hierarchical file system resembles structural information about software projects. For example, directories in Python projects translate into packages and files into modules [40]. In Java, files export at most one top-level type and the directory layout doubles as package hierarchy [20]. In summary, sets of files can be regarded as a high-level, lightweight description of changes to modules in software repositories.

Failing test cases form the other integral component of the model because they indicate the presence of a program fault or a misaligned test oracle. In either case, the attention of a developer is required who can debug and devise a suitable patch to green the test again. Because many development workflows require passing the entire test suite, for example as part of a series of pre-merge checks or before entering a review phase with another developer, correcting faults is crucial for making progress toward mainline development.

Figure 3.2 shows an exemplary matrix connecting test cases (left axis) and files (lower axis) via accumulated failure counts over 40 revisions. For instance, given that file `HPackDraft05.java` has been changed, `HttpResponseCacheTest`

and `SpdyConnectionTest` are likely test failures, according to the prior observations. If the pattern that connects file changes to failure events was indeed stable across several revisions, a prioritization algorithm could receive a new batch of changed files as input and extrapolate future test failures.

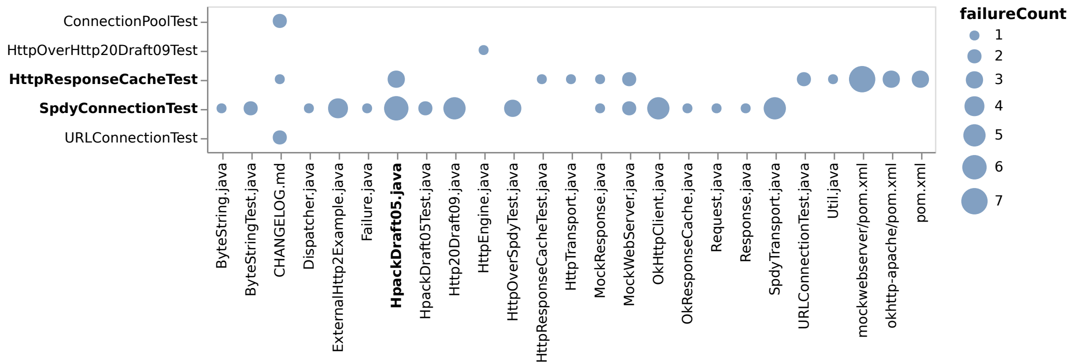


Figure 3.2.: Excerpt of one of the file-to-failure matrices from the `okhttp` project. For brevity, class names refer to test cases instead of a fully-qualified name and file names stand for the otherwise lengthy relative path.

3.2.2. Creating Failure Matrices

First, we describe the fundamental notation for failure matrices. Then, we proceed to define how to derive a failure matrix, given the file changes for a single revision and the test failures of a single regression test session.

Definition 1 *Preliminaries.* $T = \{T_0, T_1, \ldots, T_n\}$ *denotes the ordered set of test cases of a single testing session at the class level. Further, the function* $red(t) \to \mathbb{N}_0, t \in T$ *returns the number of "red" test methods of* t*, that is, the count of test methods which either result in a runtime error or assertion failure.*

At the file level, $F = \{F_0, F_1, \ldots, F_m\}$ *denotes the ordered set of files in the current software revision.* F' *denotes a set of path names of files whose content is about to change. They represent the query, which results in the prediction of future test failures.*

Distributions are expressed using the following convenience notations: $M(F_m)$ *projects a matrix row, that is, a vector containing all the per-test failure counts of* F_m*. Conversely,* $M(T_n)$ *gives us the failure distribution of a test case (matrix column), a vector whose entries consist of per-file failure counts.*

We define matrix entries for a single revision in terms of a relation. Using the schematic of Figure 3.3, this process is equal to populating the matrix columns with the respective failure counts in the matrix rows which resemble the current changeset.

	T_0	T_1	$\cdots$	T_n
F_0	2	1		
F_1	3	2		
$\vdots$				
F_m				

Figure 3.3.: Matrix schema. Files occupy a matrix row and the test case failure distributions are equivalent to columns.

Definition 2 *Matrix Relation. Using T, the current test suite and F', the current change-set, the matrix for a single revision populates as follows:*

$$M(F_m, T_n) = \begin{cases} red(T_n) & \text{if } F_m \in F' \\ 0 & \text{otherwise} \end{cases} ; F_m \in F,\ T_n \in T \tag{3.1}$$

The definition above also allows to express M as a set of triples (F_m, T_n, k_{mn}), where k is the number of failed test methods of test class T_n, given that file F_m has changed. Further, $F(M)$ denotes the distinct set of file identifiers contained in matrix M, and $T(M)$ returns all distinct test names, respectively.

3.2.3. Combining Historic Observations

All matrix approaches are history-based and exploit collections of past failure events from different regression test sessions. Therefore, we describe a way to create a single matrix for an ordered set of revisions that devalues observations of older revisions. This proposal is an extension of prior work to matrices [27].

Definition 3 *Combining Failure Matrices. Given two consecutive regression testing sessions which result in failure matrices M and N and factor $\alpha \in [0,1] \subset \mathbb{R}$, the combined matrix $M \circ N$ computes as follows:*

$$M \circ N = \begin{cases} (f, t, \alpha N(f,t) + (1-\alpha)M(f,t)); & \text{if } f \in F(N) \cap F(M) \\ (f, t, k) & \text{otherwise} \end{cases} ; (f, t, k) \in M \cup N \tag{3.2}$$

In case M and N are of equal dimensions, Equation 3.2 resembles a normal matrix sum in which the factor α determines the weights of the summands. A high α value emphasizes recent observations, while low values of α emphasize older observations. But because files and test cases might be added and removed, the equation also needs to account for summands of different dimensions. In that case, we form the union of file and test case pairs and devalue only observations whose files are present in both summands. As a result, a failure event for a file ages all prior observations connected to it.

Finally, we proceed to find the failure matrix M^*, which combines all past failure events with respect to a software revision into a single failure matrix. Subsequently, this matrix is the basis for all matrix-based prioritization efforts.

Definition 4 *Combining Multiple Matrices. Given a list of matrices $M_0, M_1, \ldots, M_i$ for software revisions 1 through i, we define the reduced matrix M_i^*, which combines all failure events of the individual revisions up to i, to be the result of a left-associative reduction operation:*

$$M_i^* = (((M_0 \circ M_1) \circ M_2) \circ \cdots \circ M_{i-1}) \tag{3.3}$$

In summary, we generalize a history-based approach for failure events for test cases to work in an additional dimension and show how to cope with different underlying sets of files and test cases, respectively.

3.2.4. Exemplary Matrix Combination

$$\begin{array}{c|cc} & T_0 & T_1 \\ F_0 & 3 & 2 \\ F_1 & 1 & 4 \end{array} \quad \circ \quad \begin{array}{c|cc} & T_1 & T_2 \\ F_0 & 6 & 24 \\ F_1 & 5 & 6 \\ F_2 & 10 & 8 \end{array} \quad = \quad \begin{array}{c|ccc} & T_0 & T_1 & T_2 \\ F_0 & 0.6 & 5.2 & 19.2 \\ F_1 & 0.2 & 4.8 & 4.8 \\ F_2 & 0 & 10 & 8 \end{array}$$

Figure 3.4.: Combining two failure matrices using $\alpha = 0.8$

Figure 3.4 shows how to combine two exemplary matrices and aids in discerning three different cases of Equation 3.2.

First, we observe that failure counts for F_2 appear in the final matrix without modification because prior observations for F_2 do not exist. Similarly, if more files did exist in the left summand, their records would appear unchanged in the final matrix. This way, the final matrix accumulates a memory of past failure events.

Second, the algorithm devalues failure events of T_0 because of the arrival of new failure events for both files, F_0 and F_1. The failure events of the right-hand-side matrix are more recent and should therefore dominate over prior entries.

Third, the failure counts of T_1, the only test case common to both summands, consists of a weighted sum of the respective entries of both input matrices. The failure counts of test case T_2 experience the same devaluation, given that the prior failure counts are zero.

For a detailed understanding of the definition, creation, and combination of failure matrices, the implementation provides further guidance (see listings A.2, A.3, and A.4).

3.3. Prioritization Algorithms

The following section explains how our proposed algorithms leverage the file-to-failure-matrix M^* for test prioritization. In principle, we offer two possible interpretations: Viewing the failure matrix as an information source for computing conditional probabilities that test cases will fail and an alternative view in which the algorithm weighs individual failure distributions of files to make an informed decision. Table 3.1 gives an overview of all proposed approaches.

The problem statement for matrix-based test prioritization is as follows: Given a new set of test cases T in untreated order and a set of changed files F', find vector F^* of length $|T|$, which assigns the new priorities F_i^* to test cases T_i. The final output of the prioritization algorithm consists of the list of test cases sorted by their priority in descending order, such that test cases of higher priority do execute first.

Table 3.1.: Weighting Function Overview

Shorthand	Implementation	Idea
conditional-prob	A.5	Decision tree; select roots in F'
path-similarity	–	Favor similar path names according to LCS
recently-changed	A.6	Favor test cases of recently changed files
file-similarity	A.7	Favor similar file failure distributions
tc-similarity	–	Favor similar test case failure distributions

3.3.1. Conditional Probability

Each matrix entry is equivalent to a failed test method count in case of M or a weighted sum of failed test method counts in case of M^*, *given that file F_m has changed*. This allows to transpose the matrix into a decision tree of depth two in which the first level encodes which file has been changed and the second level indicates which tests failed (Figure 3.5).

Therefore, we determine the selection priority of a test case by summing the probabilities that this test case fails in conjunction with any of the files of the current changeset:

$$F_i^* = \begin{cases} \sum_f P(T_i \cap f) = \sum_f P(f) * P(T_i \mid f); & \textit{if } f \in F' \\ 0 & \textit{otherwise} \end{cases} \tag{3.4}$$

In which $P(f)$ is the relative amount of failures that f attracted and $P(T_i \mid f)$ is the relative amount of failures which T_i attracted given that f has been changed.

Figure 3.5.: Exemplary conversion of a failure matrix to a decision tree

Because all test cases which have never failed in conjunction with any of the files of the current changeset possess priority zero, execution of those test cases will keep the chronology of the untreated test order.

3.3.2. Weighted Sum

We introduce a set of approaches that use a weighted sum of failure distributions to prioritize test cases. We propose to find F^* by using a weighted sum of the individual file failure distributions F_m according to different weighting criteria. In the following, we first discuss details of the weighted sum of failure distributions and then proceed to explain the individual weighting functions.

Definition 5 *Weighted Sum Prioritization. Assign test priorities by finding vector F^*, which summarizes failure distributions of all F_m using a weighted sum, determined by weighting function $w : F(M^*) \times T(M^*) \to \mathbb{R}_0^+$. We use the shorthand w_{mn} to refer to the weight of the combination of change F_m and test failure T_n.*

$$F^* = \sum_{m=0}^{|F|} \left[w_{m0} * M(F_m, T_0), w_{m1} * M(F_m, T_1), \ldots, w_{mn} * M(F_m, T_n) \right] \qquad (3.5)$$

For brevity, we elided further input parameters to w, which remain fixed per invocation of the prioritization algorithm. In addition to a combination of a file and a test case, the weighting function has also access to the current reduced matrix M^* as well as the set of changed files F', and possibly other state inferred from earlier invocations of the prioritization algorithm.

In the following, we introduce four weighting functions and describe the rationale behind them. Table 3.1 gives an overview of shorthands and the main ideas of each algorithm.

3.3.2.1. Path Similarity

The *path-similarity* weighting functions assign higher weights to files that possess an absolute path name similar to any of the changed file paths. As Section 3.2.1 outlines, programming languages encourage using directory layouts to express modularity concepts. In consequence, we expect to see cohesion between the path names of files that group related software concepts. A file with a similar path name may treat the same concept or feature, and the related test cases are more likely

to reveal faults in this particular module. Conversely, the algorithm assigns lower priorities to files with dissimilar path names, because their associated tests treat a different problem domain and are therefore less likely to reveal faults.

$$w_{mn} = max\{\, \text{LCS}(f, F_m) \mid f \in F' \} \tag{3.6}$$

Equation 3.6 uses the Longest Common Subsequence (LCS) between a file and any of the changed files to determine the weight of a matrix row.

3.3.2.2. Change Recency

The *recently-changed* weighting function factors in how recently the files were part of a change list. The algorithm assigns higher priorities to files that participated in the current or recent changesets. Given that the change to a module might be spread over several changesets and builds, we expect test cases connected to files changed in a nearby revision to stay relevant for the next testing session. For this to work, the prioritization strategy using *recently-changed* tracks an additional history per file.

$$\begin{aligned}
& f \in F(M_i^*),\, i \in \mathbb{N}_0 : \\
C_i(f) &= \begin{cases} 1 & \textit{if } f \in F' \textit{ at revision } i \\ 0 & \textit{otherwise} \end{cases} \\
\textit{recency}(f, i) &= \begin{cases} \alpha * C_i(f) + (1-\alpha) * \textit{recency}(f,\, i-1); & \textit{if } i \geq 0 \\ 0 & \textit{otherwise} \end{cases} \\
w_{mn} &= \textit{recency}(F_m,\, \textit{currentRevision})
\end{aligned} \tag{3.7}$$

Equation 3.7 shows the definition of recency. It uses the same devaluation algorithm that is also applicable to failure events in one of the simpler heuristics [27] and adopts it to change events to files. $C(f)$ denotes a vector that contains the per-file history. In turn, *recency* uses smoothing constant $\alpha \in [0, 1] \subset \mathbb{R}$ to devalue older observations and emphasize to recent changes of the file. Finally, seeding the recency computation with the current revision number yields the weight of a file in the current test session.

3.3.2.3. File Failure Distribution Similarity

The weighting function *file-similarity* uses the cosine similarity of the failure distribution of files. This way, *file-similarity* assigns higher weights to related files which frequently trigger the same test cases, and lower weights to failure distributions that do not cover related functionality.

Equation 3.8 shows how to compute the cosine of θ, the angle between two failure distributions A and B, using the Euclidean dot product formula. A_i and B_i denote the respective vector component at position i. Because there are no negative failure counts, both vectors remain in the positive space and the respective cosine is bounded in $[0, 1]$. As a result, pairs of identical vectors will possess a similarity score of one and orthogonal vectors will obtain zero, the score of maximum dissimilarity.

$$similarity(A, B) = cos(\theta) = \frac{A \cdot B}{||A||||B||} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}} \tag{3.8}$$

Equation 3.9 shows the actual file weights, which consist of the maximum similarity of a file's failure distribution to the failure distributions of any of the files contained in the current change list.

$$w_{mn} = max\left\{similarity(M^*(F_m), M^*(f)) \mid f \in F'\right\} \tag{3.9}$$

3.3.2.4. Test Case Failure Distribution Similarity

The weighting function *tc-similarity* looks for links between test cases using an indirection. The algorithm first collects the set of test cases that have been failing given any of the files from the current changeset. Then *tc-similarity* assigns higher weights to test cases, which possess a similar failure distribution and lower weights to farther distant test cases, using the previously introduced cosine similarity. Analogously to the file failure distribution, we expect similar test cases to be related, and to test the same or adjacent software functionality as contained in the current changeset.

$$\begin{aligned} relevantTC(f) &= \{t \mid (f, t, k) \in M^*; k > 0, f \in F'\} \\ w_{mn} &= max\{similarity(M^*(T_n), M(t)) \mid t \in relevantTC\} \end{aligned} \tag{3.10}$$

3.3.3. Examples

To show how different sets of changed files impact the prediction and how to put Equation 3.5 to work, we examine two end-to-end examples for matrix-based prioritization: First for the probabilistic interpretation and subsequently for the *file-similarity* weighting function.

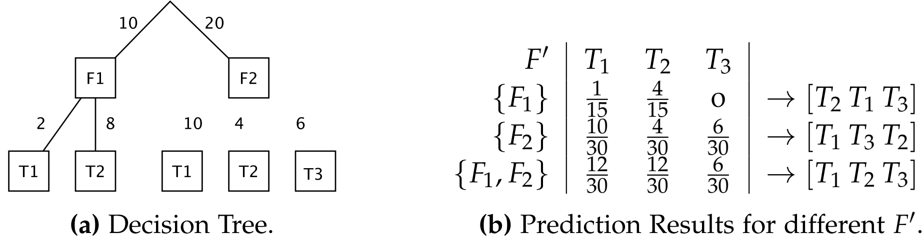
$$M^* = \begin{array}{cccc} & T_1 & T_2 & T_3 \\ F_1 & 2 & 8 & 0 \\ F_2 & 10 & 4 & 6 \\ F_3 & 2 & 4 & 1 \end{array}$$

Figure 3.6.: Running Example

Figure 3.6 shows the file-to-failure-matrix M^*, which serves as running example, and Figure 3.7a shows the result of transposing the matrix into a decision tree of depth two. Assuming both files have changed ($F' = \{F_1, F_2\}$), the probability of T_1 computes as follows:

$$F_1^* = P(F_1 \cap T_1) + P(F_2 \cap T_1) = \frac{10}{30} * \frac{2}{10} + \frac{20}{30} * \frac{10}{20} = 0.4$$

The probabilities for T_2 and $T3$ compute accordingly. Figure 3.7b shows that indeed different definitions of F' result in differing test orderings.



(a) Decision Tree.

F'	T_1	T_2	T_3	
$\{F_1\}$	$\frac{1}{15}$	$\frac{4}{15}$	0	$\rightarrow [T_2\ T_1\ T_3]$
$\{F_2\}$	$\frac{10}{30}$	$\frac{4}{30}$	$\frac{6}{30}$	$\rightarrow [T_1\ T_3\ T_2]$
$\{F_1, F_2\}$	$\frac{12}{30}$	$\frac{12}{30}$	$\frac{6}{30}$	$\rightarrow [T_1\ T_2\ T_3]$

(b) Prediction Results for different F'.

Figure 3.7.: Conditional probability example

In the following, we exemplify the weighting-based approach on the example of the *file-similarity* weighting function. Assuming $F' = \{F_1\}$, we begin by computing the cosine similarities to the remaining files F_2 and F_3, respectively:

$$sim(F_1, F_2) = \frac{2 * 10 + 8 * 4 + 0 * 6}{\sqrt{2^2 + 8^2} * \sqrt{10^2 + 4^2 + 6^2}} \approx 0.51$$

$$sim(F_1, F_3) = \frac{2 * 2 + 8 * 4 + 0 * 1}{\sqrt{2^2 + 8^2} * \sqrt{2^2 + 4^2 + 1^2}} \approx 0.95$$

This gives us the weight vector $w_m = \begin{bmatrix} 1 & 0.51 & 0.95 \end{bmatrix}$, which assigns a weight to every F_m. Using these weights, we proceed to find the column sums, which indicate the final priorities of the T_n:

$$F^* = \begin{bmatrix} 2 + 0.51 * 10 + 0.95 * 2 \\ 8 + 0.51 * 4 \quad + 0.95 * 4 \\ 9 + 0.51 * 6 \quad + 0.95 * 1 \end{bmatrix} = \begin{bmatrix} 9 & 13.84 & 5.01 \end{bmatrix}$$

Since the output consists of the test cases sorted by priority in descending order, this leaves us with a final result of $[T_2\ T_1\ T_3]$.

4. Dataset

Our newly derived dataset contains prolonged build histories of small to medium size open-source projects.[1] In contrast to prior studies, which operate on relatively small test execution histories of less than one hundred revisions [19, 48], the dataset allows studying history-based approaches given sufficiently long build histories. Concerning project size, the elaboration that is based on this dataset is located between prior works that target academic test programs, which seldom exceed 500 lines of code [27, 44], and studies of large-scale enterprise code bases [29, 38].

The dataset enables us to study a diverse set of projects of small to medium size and gives insight into testing efforts resulting from an open contribution model. Because software builds are not reproducible in general,[2] we reconstruct test execution histories from build log files of a popular continuous integration service [53]. Because logfiles may not contain all test events, *fully* reconstructible test executions are not commonplace. Therefore, the following section describes how we obtained the dataset, elaborates on the project selection, and discusses the difficulties of extracting test execution histories from publicly available data.

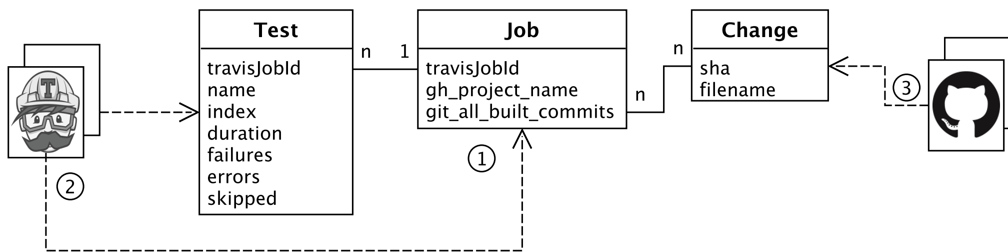


Figure 4.1.: Dataset Overview. Data source and derived entities connect via arrows; solid lines show a foreign-key relationship.

Section 4.1 details the TRAVIS CI service and its peculiarities. The TRAVIS-TORRENT project provides us with a database of past CI jobs that have run on the TRAVIS CI infrastructure. Figure 4.1 shows this database with the central *Job*

[1] The dataset has been published as RTPTorrent [32].

[2] This also applies to projects adhering to continuous integration principles, which prescribe building the software project from scratch in a fixed environment. Non-deterministic test cases, such as induced by using a RNG without seed, querying online third-party services, or referring to dependencies with fuzzy coordinates, hinder reproducibility.

relation (①), which contains important metadata, such as the project name, and a pointer to relevant commits. Further, we access the collection of raw logfiles of TravisTorrent (②), redo the build log analysis, and recover the additional attributes of the *Test* relation, which Section 4.3 explains in detail. Lastly, our dataset consists of *Changes*, which we recover from GHTorrent, a database of commits from GitHub (③). Section 4.4 details the recovered changes on a per-project basis. The dataset consists of one csv file per project per entity of Figure 4.1 and totals 1.7 GB in size.

4.1. Continuous Integration at Travis CI

travis ci is a popular hosted continuous integration service, allowing developers to re-execute procedures for build, test, and deployment upon every check-in to the software repository. Central to this service is a dedicated configuration file, which lays out the build commands to be run, as well as their environment. Furthermore, developers can view the software build process live on the web and there is a tight integration to third-party services, such as GitHub. This integration allows triggering builds as soon as new commits arrive and informs contributors of the current build status at project or merge request level. Because open-source projects can utilize travis ci for their builds free of charge, the service is popular in the open-source community. Also, larger organizations, such as the Apache Software Foundation, are adopting travis ci at an increasing scale [14]. This is true even though travis ci imposes several limitations on its non-paid services: At most five concurrent builds are allowed per project, a job may not take more than 50 minutes to complete, and might be terminated if the build does not create any output every ten minutes [52].

In the following, we refine the terms *build* and *job*, discuss the structure of build histories, and detail how this affects prioritization. Figure 4.2 shows the structure of a project build history consisting of two builds. Each build consists of one or more jobs. Jobs resemble variations in the build process, which might require the software version to successfully compile and pass the test suite, given different operating systems, processor architectures, or language runtime versions. In particular, jobs execute in isolation and possibly in parallel. As a result, there is no particular order between jobs of the same build. A build is completed after all subordinate jobs are completed and it is successful if all associated jobs terminated normally.

4.2. Build Linearization and Learning

We adapt existing as well as newly conceived prioritization heuristics to this setting as follows. We split the algorithms in two stages: The first stage incorporates knowledge about the outcomes of past regression test sessions (learning) and the second stage prioritizes test cases. Figure 4.2 shows that, while prioritization takes

place for every job, learning occurs after every build. This setup implements two important design rationales.

First, jobs cannot see test outcomes from another job of the same build. This way, the predictions for jobs within the same build are independent of each other and the results of a particular job only influence prediction after completely grading the software revision. Learning once per build also abstracts from the scheduling policy of jobs. The prioritization results are the same, regardless of whether jobs execute sequentially, or in parallel.

Second, builds cannot see test results of partially completed builds even though a subset of jobs of the latter builds may have completed. To this end, we keep a list of pending builds whose outcome is yet to be determined, when prioritizing test cases for the current build. As a direct result, the learning stage consists of incorporating knowledge about a set of builds, which make the transition from pending to completed, before the prioritization algorithm grades the next build. By implementing basic build isolation, we set up a linearized, deterministic history of prioritization and learning steps.

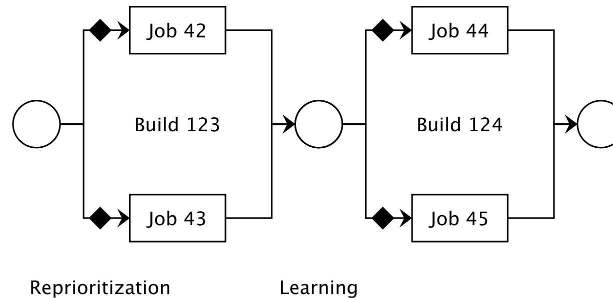


Figure 4.2.: Jobs and builds within history

4.3. Recovering Test Execution Histories from Public Data

Our dataset is based on projects from TravisTorrent [4], a publicly available dataset of projects from GitHub [18], a popular code hosting service, ranked by highest accumulated test failure count. Because computing fault detection efficiency measures, such as APFD, require all executed test cases regardless of their test verdict as input, we redo the build log analysis to recover all test cases and to include additional per-test metadata. Table 4.1 explains attributes of concern in detail. Furthermore, the number of *red* tests refers to the sum of failures and errors for a given test class. For the remainder of this report, we do not assign significance to whether the fault in question manifests as runtime exception or assertion failure.

Table 4.1.: Recovered test attributes from build log analysis

Test Attribute	Description
Name	Name of test class, possibly fully qualified
Index	Position of the test case in this build
Duration	Execution time in seconds
Count	Number of test methods of this test class
Failures	Number of test methods with assertion failure
Errors	Number of abnormally completed test methods (exception)
Skipped	Number of skipped test methods (unmatched precondition)

From a programming language perspective, this study is limited to Java projects despite the fact that the TravisTorrent dataset also contains projects with Ruby or JavaScript as their primary programming language. This is due to the fact that only very few test runners output test results to the console regardless of their verdict. The majority of test runners prefer to reduce verbosity by outputting only red tests along with their cause, which renders reconstruction of all executed tests by means of build log analysis infeasible. To the best of our knowledge, only two build tools possess detailed enough output, including the Apache Maven Surefire integration [51] and Facebook's Buck build tool [11], whose primary application domain is Java projects.

Table 4.2 lists all projects, their primary functionality, and basic project statistics, such as the number of test classes. We observe that the problem of RTP boils down to finding that one test class that fails often. Second, all projects employ the "fail-at-end" strategy during test execution. That means, the build system does not abort the test run after the first failure and executes the remainder of the test suite, even though the build outcome has already been determined (verification failure due to at least one red test). For all projects, we can observe builds with multiple, distinct test failures of up to 63 simultaneously failing tests on average.

Table 4.2.: Study subjects with basic statistics: number of test classes (= TC), test methods (= TM), and failed test cases, averaged per-build

Project	Description	Jobs	Red Jobs		TC	TM	Failed TC
achilles	Object-relational mapper	803	27	3 %	205	1,566	2
buck	Build system	1,547	341	22 %	652	3,909	1
cloudify	Cloud orchestration	5,752	496	9 %	38	125	1
deeplearning4j	Deep learning & linear algebra	2,791	566	20 %	12	28	1
dspace	Digital asset management system	2,142	82	4 %	58	175	35
dynjs	ECMAScript runtime for the JVM	1,009	41	4 %	74	844	1
graylog2-server	Log management	8,859	247	3 %	40	239	1
hikaricp	High-performance JDBC connection pool	1,611	125	8 %	40	171	1
jade4j	Implementation for Jade template language	1,122	96	9 %	41	247	19
jcabi-github	Object Oriented Wrapper of Github API	2,360	337	14 %	158	627	2
jetty.project	HTTP Server	408	327	80 %	111	1,116	1
jooq	Write SQL in Java	3,897	523	14 %	32	195	1
jsprit	Vehicle routing problem solver	1,097	51	5 %	82	963	1
littleproxy	High performance HTTP proxy	671	62	9 %	17	68	2
okhttp	An HTTP+HTTP/2 client for Android	5,781	779	13 %	35	813	1
optiq	Dynamic data management framework	1,554	70	5 %	47	1,355	1
sling	REST framework	4,412	812	18 %	174	949	1
sonarqube	Continuous Inspection	30,142	620	2 %	413	2,334	1
titan	Distributed Graph Database	1,061	254	24 %	38	294	2
wicket-bootstrap	Twitter Bootstrap for Wicket	1,167	342	29 %	41	146	29

4.4. Changesets

We recover *changesets*, that is, sets of file paths that are changed together in one Git commit, by looking up the Git commit IDs of TravisTorrent (commits in the push that triggered the build, or commits since the last build) in the GHTorrent dataset [21]. Table 4.3 gives insight into the number of changes and the average size of changes in open-source software. In the table, we can see that only a handful of files are changed within one commit on average. A tendency exists towards comparatively small sets of changes that pinpoint a particular problem or feature, as opposed to mass-committing large refactorings. The number of unique files indicates the maximum cardinality of files that prioritization algorithms work on. Because the metric denotes the number of files that exist over the entire lifetime of a project, the number may overestimate the projects' true size in terms of the number of files and may rather be an indicator to rename activity.

Table 4.3.: Changeset statistics

Project	Commits	Files per Commit	Unique Files
`achilles`	763	5	2,684
`buck`	7,160	4	12,074
`cloudify`	11,078	2	3,724
`deeplearning4j`	2,607	4	26,470
`dspace`	3,779	2	4,840
`dynjs`	531	2	781
`graylog2-server`	6,146	2	4,123
`hikaricp`	1,703	1	446
`jade4j`	374	3	1,621
`jcabi-github`	753	1	471
`jetty.project`	205	2	602
`jooq`	2,006	2	7,147
`jsprit`	308	1	426
`littleproxy`	611	2	399
`okhttp`	2,315	3	1,257
`optiq`	846	3	1,771
`sling`	13,763	2	13,200
`sonarqube`	5,253	4	14,961
`titan`	679	2	1,813
`wicket-bootstrap`	1,294	2	1,407

5. Evaluation

In the following, we proceed to evaluate the dataset and prioritization techniques in detail. To this end, Section 5.1 explains the relevant measures for fault detection efficiency and test schedule similarity. Section 5.2 introduces the state of affairs in regression testing on open-source software projects and examines the fault detection efficiency of current real-world test schedules. We continue to analyze the performance of prioritization approaches from related work in Section 5.3. Section 5.4 reports on the performance of the proposed set of matrix-based approaches. From these three sets of experiments, we extrapolate further findings in Section 5.5 and detail the threats to validity in Section 5.6.

5.1. Method

In the following, we describe the study's environment, detail on compared approaches, measures, and implementation details. Section 5.1.1 describes prioritization approaches from related work, to which we will compare the novel set of matrix-based approaches. Further, Section 5.1.2 introduces Average Percentage of Faults Detected (APFD), a measure for fault detection efficiency of individual test schedules and Section 5.1.3 outlines Rank-biased Overlap (RBO), which we use to quantify similarities between two test schedules. We give insight into the technical details of the CI simulation in Section 5.1.4.

5.1.1. Baseline Approaches

The following section elaborates on all six baseline approaches in detail and we compare the novel approaches in the following experiments.

5.1.1.1. Untreated

The untreated test schedule consists of tests in order of *test discovery*, which is largely a black box defined by implementation details of the test runner. We generally believe the untreated test order to be arbitrary, but fixed to a reasonable degree. Newly developed test cases may appear at any index, but the relative order of tests to each other stays the same across different regression test sessions. Many factors weigh in on the untreated order: Among others, we identified the order of file directory traversal, declaration order of tests in the target programming language, and build dependencies on multi-module projects, to be influential.

5.1.1.2. Randomized

Because the untreated test order is relatively fixed, we propose to randomize test order. For example, a randomized test order would bring improvement in situations when test discovery has a tendency to append newly developed and therefore unstable test cases at the end of the test schedule. Also, if the position of a test case with high fault detection ability is fixed towards the end of the test schedule, a randomized test order would allow earlier test execution on average. Earlier experiments consistently show an improvement in fault detection efficiency of a randomized test schedule [44].

5.1.1.3. Least Recently Used (LRU)

Originally developed for test case prioritization in resource constrained environments [27], we adapt the notion of the least recently used (executed) test case to non-constrained settings. On every invocation, the LRU algorithm seeds the test order with newly developed (previously unseen) test cases. Subsequently, the last test case of the prior testing session is queued and after that, the remaining test cases are scheduled in order of their last execution. This way, the algorithm cycles through all tests, similar to the functionality of a ring buffer. We expect to see improvements in fault detection efficiency in scenarios in which newly added test cases are the most unstable ones and need several test sessions to stabilize, in which they still claim top positions in the test schedule.

5.1.1.4. Recently Failed

Without modification, we implement history-based test case prioritization based on *demonstrated fault effectiveness* [27]. Because a test case failure may be an indicator to several faults, which are hiding each other, the test case requires execution in multiple, subsequent regression test sessions, until the developer eliminated all failure causes. Similarly, performing development iterations on the same production code, either for different behavior due to changing requirements or due to bug removal, may require several test case failures until that code stabilizes. For these reasons, the *recently-failed* algorithm assigns high selection probabilities to test cases that failed in recent regression testing sessions and low probabilities to those which did not fail, or whose failures are increasingly in the past. Listing A.1 contains the implementation of the *recently-failed* algorithm.

5.1.1.5. Optimal Failure

Optimal test schedules only approximate an Average Percentage of Faults Detected (APFD) value of 1.0, but in practice never reach it. Therefore, we need to compute the optimal ordering, in which all red test classes execute first, to be able to precisely determine the gap between prioritization strategies and theoretical optimal ordering concerning fault detection efficiency. The *optimal failure* test plan orders failed test classes by the ratio between failed test methods and all test methods of this class.

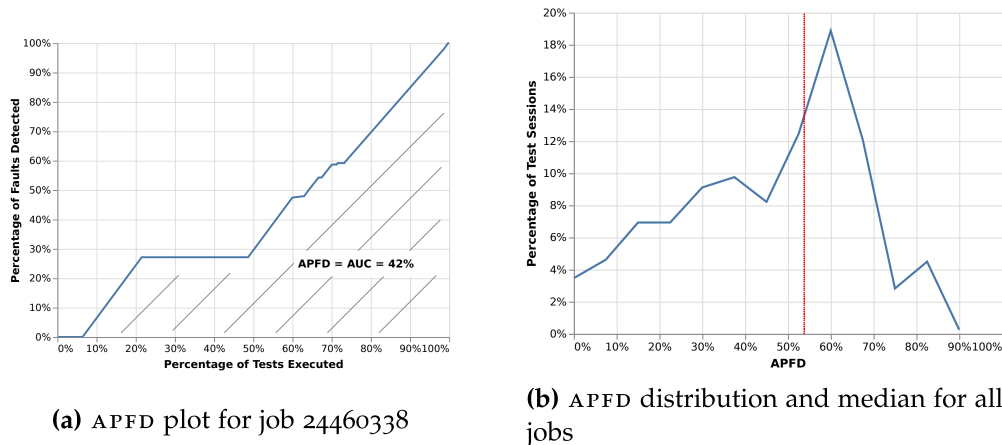
5.1.1.6. Optimal Failure Duration

This prioritization algorithm reorders tests such that classes with a better trade-off with respect to the duration of failed test methods to the duration of the entire test class execute first.

5.1.2. Fault Detection Efficiency Measure

We introduce Average Percentage of Faults Detected (APFD) [8, 44] as a state-of-the-art measure for fault detection efficiency for regression test prioritization. The measure grades individual regression testing sessions with respect to how fast this particular test order detected faults. APFD values range from zero to one and higher APFD values imply that the tests reveal more faults earlier in the process. Figure 5.1a shows an exemplary plot showing the fault detection progress, given the percent of the test suite executed and the percent of faults detected. The APFD value is equal to the Area Under Curve (AUC); if the test plan detected faults earlier, the curve moved to the top left corner, increasing the area accordingly. Assuming that all faults have equal severity and that all test cases have equal cost, the APFD value computes as follows, using ordered sets $F = \{f_0, \ldots, f_m\}$, the present faults, and $T = \{T_0, \ldots, T_n\}$, the test schedule:

$$\text{APFD}(F, T) = 1 - \frac{\sum_{f \in F} \text{argmin}_i t_i(f) = \text{fail}}{n * m} + \frac{1}{2 * n} \tag{5.1}$$



(a) APFD plot for job 24460338

(b) APFD distribution and median for all jobs

Figure 5.1.: APFD plots of the untreated test order of the `okhttp` project

If we report on the performance of a prioritization strategy on a collection of regression test sessions, we use the median APFD value. Figure 5.1b gives an overview of the fault detection efficiency of the untreated test order on all 779 regression test sessions of the `okhttp` project. When comparing different prioritization approaches, we transpose the detailed APFD distribution into a boxplot.

5.1.3. Similarity Measure for Test Schedules

In addition to comparing the efficiency of prioritization techniques, we also require a measure for assessing test schedule similarity. To this end, we introduce Rank-biased Overlap (RBO), a similarity measure for indefinite ranked lists [55], argue why the measure fits our use case using relevant properties of RBO, and briefly outline its calculation.

Despite the fact that prioritization algorithms provide different test schedules, the APFD score may vary insignificantly, if the test classes contain failed test methods at a similar rate. In particular, which test class revealed the faults is not relevant to the APFD metric. For this reason, comparing APFD is a good indicator for prioritization performance, but not for similarities of compared test schedules. Second, because APFD employs a uniform cost model, the measure is unsuited to estimate the impact of cost-cognizant versus non-cost-cognizant prioritization. Therefore, we view prioritized test schedules as ranked lists, in which test cases of higher failure probability occupy higher ranks and turn to RBO, a similarity measure for indefinite rankings, to quantify differences in prioritization.

In particular, the combination of the following properties makes RBO a good fit for the test prioritization scenario. First, the RBO measure handles ties without further modification or preprocessing. This is beneficial, given that ties are a frequent occurrence because all prioritization heuristics will attempt to place likely test failures first. Second, the RBO measure is symmetric, which is particularly helpful in absence of a gold standard to compare other prioritization approaches to. In fact, we propose two different optimality criteria for test schedules and require to mutually compare them. Third, the RBO measure employs configurable top-weightedness. Driven by the objective to minimize feedback time, we naturally assign greater weight to the first couple of tests to execute and are less interested in which test case executes last or second-to-last. Conversely, the similarity measure should emphasize the agreement of the prefix of two different test schedules more and emphasize the remainder less, as it is of diminishing relevance to regression testers.

We outline the basics of RBO. As opposed to correlation-based measures, RBO is based on the size of the set intersection at depth d:

Definition 6 *Given two lists S and T, the agreement-at-depth A_d consists of the size of the intersection of the prefixes of S and T of length d, in relation to the number of distinct elements in S and T:*

$$A_{S,T,d} = \frac{2 * |S_{:d} \cap T_{:d}|}{|S_{:d}| + |T_{:d}|} \tag{5.2}$$

Above formulation of agreement accounts for ties. Using the geometric progression as weights, we express RBO as a weighted sum of agreements:

Definition 7 *Given user persistence parameter $p \in [0, 1]$ and the ranked lists S and T, the Rank-biased Overlap constitutes a weighted sum of agreements:*

$$RBO(S, T, p) = (1 - p) \sum_{d=1}^{\infty} p^{d-1} * A_{S,T,d} \tag{5.3}$$

The parameter p models the user's persistence. If p is zero, the top-weightedness of RBO is at its maximum, and only the topmost element of both lists is compared and denotes the similarity of the two lists. As the persistence increases, the prefix becomes longer, and the user's evaluation becomes arbitrarily deep, resulting in a lower degree of top-weightedness.

Because RBO is conceptually geared towards the list of infinite length, we need to bound prefix evaluation up to a given d. Therefore, it is possible to report on RBO using a min-max-interval, in which the former is a lower bound to the RBO value, assuming that all elements past index d are different, and the latter is an upper bound, assuming that all remaining elements are equal. However, min-max-values are dependent on the length of the evaluated prefix in a significant manner. Therefore, we choose to report on RBO_{EXT}, which assumes that the rate of similitude of the prefix continues indefinitely. The value of the RBO extrapolation stays in between the min-max-range and represents the best guess using the information from the bounded prefix. Detailing the calculation of RBO_{EXT} appears outside the scope of this project; the existing elaboration should be sufficient for evaluation purposes.

5.1.4. Implementation

For our series of experiments, we need to simulate the CI workflow and examine the resulting test schedules. In the following, we describe three components: log extraction, CI simulation, and evaluation. The log extraction component takes a set of TRAVIS CI log files as input and reconstructs a regression test session as a list of *test events* (see Section 4.3). To this end, the log extraction uses standard parsing techniques based on regular expressions, as well as simple heuristics for error tolerance in case of inconsistent logs. For example, the Apache Maven log parser first splits the log files by module and then treats module-specific snippets individually. This way inconsistencies in one module do not propagate into the parsing result of subsequently built modules of the same CI job. The CI simulation component has two responsibilities: first, implement an algorithm, which runs through the builds in their original order and ensures visibility and isolation properties (see Section 4.1). Second, house the prioritization strategies, which reorder the test cases given the currently available past test results and changesets. Listing 5.1 shows the basic prioritization strategy trait and parameter object. In particular, prioritization strategies need to cope with an incomplete picture of the past: Inside the `reorder` method, some prior builds might not have completed yet. Once pending builds have finished, the job dispatcher calls `acceptFailedRun`, to update the memory of the prioritization strategy. For further implementation details of a curated subset of strategies, we refer the reader to Appendix A. We implemented both components – log extraction and CI simulation – in Kotlin [26].

Listing 5.1: Prioritization Strategy Trait and Parameter Object.

```
class Params(
  val job: Job,
  val priorJobs: List<Job>,
  val changedFiles: List<String>,
  val testResults: List<TestResult>
)

interface PrioritizationStrategy {
  fun reorder(p: Params): List<TestResult>

  fun acceptFailedRun(p: Params) {}
}
```

For inspection of the resulting test ordering, we resort to a standard data science stack in Python [41], using Pandas [33] for data analysis and Altair [54] for most visualization tasks.

5.2. Regression Testing on Open-source Software Projects

Because the dataset is novel and contains relevant real-world software projects, we examine their current performance without any prioritization efforts by heuristics. Also, we investigate the relevance of test cost (duration) for prioritization.

RQ 1.1 To what extent do open-source projects require RTP? We determine the performance of the current test schedule and quantify the gap to the fault detection efficiency of the hypothetical optimal test ordering.

RQ 1.2 How does fault detection efficiency develop over time? Analyzing the current test schedules, is there a trend in fault detection efficiency over the projects' observation span?

RQ 1.3 Do open-source projects require cost-cognizant RTP? Prior work finds that incorporating test cost into prioritization can help find test schedules that are better suited. Therefore, we estimate the impact and check if newly developed approaches should account for test cost.

To this end, Figure 5.2 shows the distribution of the APFD values of all projects, of the currently used *untreated* and theoretical *optimal-failure* test schedule (see Section 5.1.1). Further, we examine the fault detection performance of the untreated test schedule over all observations of a project using Figure 5.3. Lastly, we compute *optimal-failure-duration*, a test schedule with a different optimality criterion – detected faults per cost – and compare it with the optimal schedule according to detected faults per test class. For this purpose. we report on RBO, a similarity measure outlined in Section 5.1.3, in Table 5.2.

5.2.1. Performance of Untreated Test Schedules

Figure 5.3 shows that the untreated test schedule for most projects is not efficient at the task of fault detection. The projects' default test runner orders tests, such that the resulting schedule fails to meet at least 50 % APFD on a regular basis. For half of the projects, we can identify at least one stretch of jobs with inferior fault detection efficiency below 10 % APFD, for example in the graphs of `jooq` or `deeplearning4j`. Other projects, such as `cloudify` or `sling`, appear to be adding faults almost exclusively at the end of their test schedules, thus resulting in very low APFD scores. Figure 5.2, which shows the distribution of APFD values for all projects, confirms this observation: on average, untreated test schedules attain an APFD value of 30 %, which is far from the median optimal of 97.5 % APFD. Factoring in that prediction models cannot constantly predict the optimal test order, finding enhanced test schedules with up to a threefold increase in fault detection efficiency appears possible. In conclusion, we find that the gap toward optimal test ordering is sizable, and prioritizing tests in those projects can lead to significant benefits.

5.2.2. Trends in Fault Detection Efficiency

We cannot extrapolate clear trends with respect to fault detection efficiency in Figure 5.3. This is contrary to our expectation: Because almost all projects accumulate more test cases over time, we estimate that, on average, more tests must execute before a test detects a randomly seeded fault. Instead, we attribute visible temporary declines in fault detection efficiency, as for projects `deeplearning4j`, `okhttp`, or `titan`, to developmental trends. These "dents" in APFD then resemble periods of time, in which developers introduce novel features, whose test cases initially lack maturity and stability, or modify code, which is covered by test cases that appear towards the end of the untreated test schedule.

5.2.3. Cost-cognizant Prioritization

Because the overarching objective of test case prioritization is faster feedback, we investigate if prioritization techniques, which also incorporate test duration, are justified for use with the dataset at hand. This is of particular importance because prior work finds that cost-cognizant prioritization techniques can improve fault detection efficiency [39] or can be helpful in at least some scenarios [30]. Studying test cost will also show, if APFD is an appropriate measure for assessing the resulting test schedules,or if its cost-cognizant counterpart APFD_c is better suited [8]. To estimate the impact of cost-cognizant RTP, we compare the test schedule, which is the result of reordering test classes by faults detected per test methods executed, to another test schedule, which is the result of prioritizing test classes by faults detected per time spent. Table 5.2 shows the results of the test schedule comparison using RBO similarity measure with top-weightedness parameter p set to 0.98. With a median test schedule similarity of 0.94, we conclude that the durations of the failing test cases per build vary marginally. This is not

surprising, given that the dataset characteristics in Table 4.2 show that for the majority of projects, one test class fails per build on average, leaving no room for variation between the two different schedules. However, if the test schedules are indeed different, their median similarity is still at 0.93 RBO. We conclude that the two different optimality criteria, namely detected faults per test class and detected faults per duration, result in very similar, if not identical, test schedules. Therefore the original formulation of APFD, which is based on identical test costs and fault severities, is well applicable in this scenario. Due to the similarities of the compared test schedules, we cannot find an imminent need for cost-cognizant test prioritization on this dataset.

Table 5.2.: RBO values for comparing the optimal test schedule accounting for test cases ran versus time spent. The overall median is 0.94.

Project	RBO	Project	RBO
`achilles`	0.980	`jetty.project`	0.922
`buck`	0.683	`jooq`	1.0
`cloudify`	1.0	`jsprit`	1.0
`deeplearning4j`	0.970	`littleproxy`	0.948
`dspace`	0.647	`okhttp`	0.944
`dynjs`	1.0	`optiq`	0.927
`graylog2-server`	1.0	`sling`	0.920
`hikaricp`	0.980	`sonarqube`	1.0
`jade4j`	0.832	`titan`	0.928
`jcabi-github`	1.0	`wicket-bootstrap`	0.772

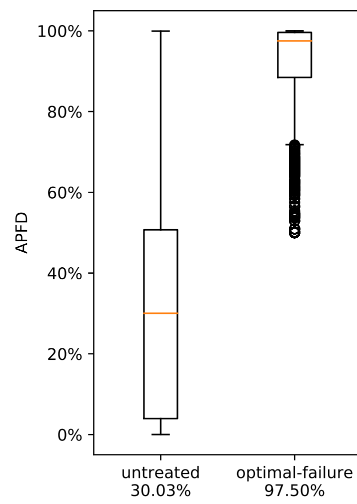


Figure 5.2.: APFD distribution for the unchanged and optimal test schedule of all projects

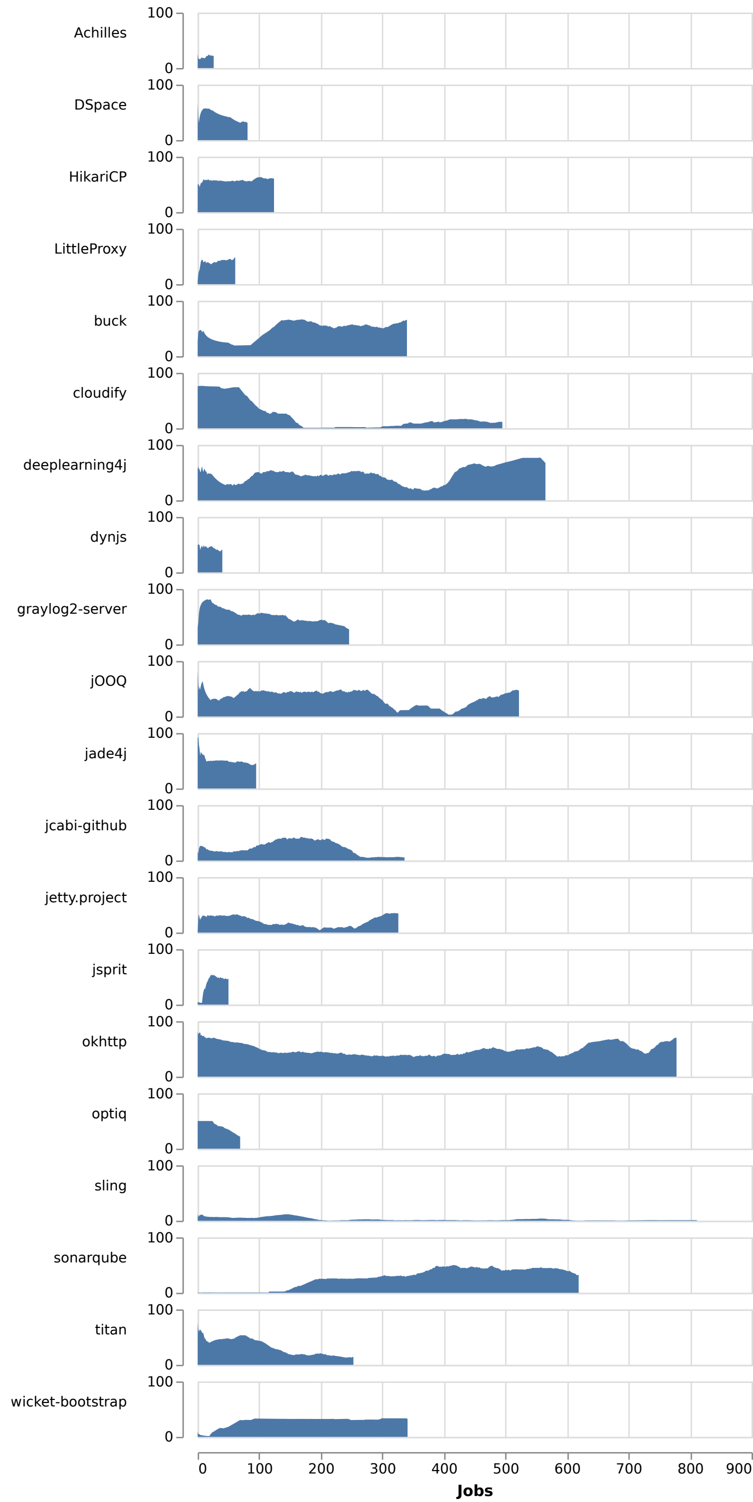


Figure 5.3.: Fault detection effectiveness (APFD) of the untreated test schedules over time

5.3. Baseline Performance

Section 5.1.1 detailed on the four baseline approaches. In the following, we investigate their performance on the newly derived dataset of test execution histories of open-source projects.

RQ 2 How do state-of-the-art baseline approaches and history-based heuristics perform with respect to fault detection efficiency?

To this end, Figure 5.4 compares the median APFD of the baseline approaches *untreated*, *random*, *lru*, *recently-failed* (see Section 5.1.1).

The history-based heuristic *recently-failed* outperforms all other heuristics on each of the projects at a median APFD of 95 %, which is close to the fault detection efficiency of 98 % APFD of the optimal plan. The good performance is in accordance with prior reports, which show that simple heuristics can perform on par with state-of-the-art machine-learning algorithms [38, 49] or generally present considerable improvements over the untreated test order [9]. Figure 5.5 provides further insight into the failure distribution on an example project and aids in explaining the efficiency of the *recently-failed* heuristic. The visualization shows that a small subset of very unstable test cases exists, which subsequently possess a high failure probability in any of the builds. Also, if a test case fails, it will continue to be red for a couple of iterations in the majority instances. Both aspects might explain the efficiency of prioritizing recently failed test cases.

Even though the *lru* algorithm assigns higher priorities to novel test cases, which are subject to active development and may require to stabilize, we find no evidence that *lru* performs better than random prioritization. The projects' opinion towards novel test cases is split: Out of 20 subjects in total, random prioritization supersedes *lru* on eleven projects. Contrary to related work [9, 13, 27], which emphasize the incorporation of last executions in their prioritization heuristics, we infer that the *lru* algorithm did not adapt well to the execution histories in the dataset.

While the *untreated* strategy appears to have the worst fault detection performance, this is true only for 60 % of projects. On the remaining 40 % of instances, either the *lru* or *random* prioritization heuristics yield test plans of inferior fault detection efficiency. We conclude that disregarding the untreated test order in favor of the simplest of heuristics can lead to unsatisfactory results and is possibly detrimental to the ability to detect faults early. On a subset of projects, the untreated test order favors faulty test cases during the time frame of the observation.

5.4. Matrix Prioritization Performance

This section examines the performance of matrix-based prioritization approaches and discusses the applicability of the weighting functions presented in Section 3.3.2. Finally, we reason about the impact and frequency of re-emerging test failures.

Figures 5.8 and 5.9 show the prediction performance of re-emerging test failures and their distribution.

RQ 3.1 How do prioritization efforts employing weighting functions compare to a probabilistic interpretation of file changes and test outcomes?

RQ 3.2 Which type of weighting function is best applicable when using a matrix-based prioritization heuristic?

RQ 3.3 Do matrix-based prioritization techniques improve fault detection efficiency in case repeating faults spread over longer periods of time?

To this end, Figure 5.6 shows the distribution of APFD values of all matrix-based approaches. For better comparison, *recently-failed*, the best performing history-based heuristic, is also included.

5.4.1. Comparison of Weighted Sum and Probabilistic Approaches

Previously, we introduced a prioritization approach, which assigns failure probabilities to test cases using a decision tree whose roots consist of the changed files and whose leaves leaves are test cases. Figure 5.6 shows that the Interquartile Range (IQR) of the probabilistic approach is almost twice as large as the IQR of prioritization efforts, which utilize the weighted-sum approach. In comparison to weighting approaches, as well as the best baseline heuristic, the probabilistic interpretation has the lowest median APFD score of only 54 %.

We conclude that the lower fault detection efficiency is due to the fact that the probabilistic approach only selects only a couple of roots of the decision tree, which represent the changed files of each graded revision. As a consequence, only a small fraction of test cases, which previously failed given any of the changed files, possess non-zero failure probabilities and executes first. The remainder of the test cases, which has no prior relation to the current changeset, has failure probability zero and therefore follow prioritized test cases in the order of the untreated test suite. This behavior is supported by the fact that, on average, there are twice as many files as test cases per build. As a result, the cardinality of the set of prioritized test cases using the probabilistic approach can even be lower than the number of changed files.

5.4.2. Applicability of Weighting Functions

5.4.2.1. *tc-similarity*

The matrix-based prioritization, which prioritizes test cases by the similarity of their failure distribution to the distributions of a set of test cases relevant for the changeset, has comparable performance to the previously discussed probabilistic approach (see 5.4.1) and has a bottleneck for a similar reason. Given that the input to the prioritization algorithm consists of a set of changed files, the algorithm has to determine the set of relevant test cases first, whose failure distributions form the basis for assigning priorities for the remaining test cases. To do that, the *tc-similarity*

weighting function relies on test cases, which have failed because some of the files of the current changeset have been modified. Therefore, none of the previous failure events may connect any of the test cases to any of the files of the current changeset. In this case, this prioritization heuristic prioritizes no test cases and outputs the untreated test order. The fact that this heuristic assigns priorities to all test cases, for which failure events have been recorded, explains the improvement in performance to 66 % APFD with regard to the probabilistic approach.

5.4.2.2. *file-similarity, path-similarity*

These two weighting functions determine the weight of the individual matrix rows by comparing their failure distribution or path name, to corresponding traits of the files which are part of the current changeset. At 84 % APFD, *file-similarity* and *path-similarity* are the second-best performing matrix-based approaches and possess identical IQRs. We attribute the remarkable similarity in the effectiveness of these two approaches to the fact, that the path name is a good predictor for the failure distribution of a file, and vice versa. This appears rational, because both, failure distribution and path names, are strongly linked to the production code they cover. Given that the failure distributions of two files are sufficiently similar and the files are covered by the same test cases, they are therefore likely to implement related or mutually dependent software concepts. As a result of the tendency of software developers to co-locate those files in similar positions in the source tree, the files possess a similar path. Alternatively, if two given files share vocabulary in their path names, which might consist of module and package names, or indicate the respective feature of the software, they likely treat the same core concept and thus "fail together". The good performance of these two weighting functions is the result of the ability to cope with (partly) unseen changesets. In the case of *file-similarity*, changesets containing only one file with a failure history are already sufficient to assign priorities to all test cases. For *path-similarity*, none of the files of the input changesets are required to be known to the heuristic a priori, because the path name comparison naturally allows for partial matches.

5.4.2.3. *recently-changed*

The matrix-based prioritization technique using the *recently-changed* weighting functions performs best at a median APFD of 93 %. As a result, this prioritization method is on-par with the *recently-failed* heuristic with respect to IQR, as well as median fault detection efficiency (95 % APFD). The good performance matches prior findings, which emphasize the fault affinity of recently churned code [35].

5.4.3. Sensitivity to Failure Distance

Because the matrix-based strategies spread test failure events across files, their potential to correctly identify repeated test failures increases. If a change contains the same or similar change list, matrix-based strategies assign higher priorities to test cases, which have been previously connected to that change list. This stands in contrast to *recently-failed*, which only assigns priorities by the proximity of the last

failure of the test case to the current build. In principle, the efficiency of *recently-failed* declines, as the distance between failures of the same test case increases.

Because of these characteristics, we investigate the extent to which different failure patterns influence test prioritization on this dataset and start by defining the term *failure distance*. The failure distance is a property of a build, which denotes the shortest distance (in the number of builds) of any of the test failures of this build to a failure of the same test case of a prior build.[1] Therefore, the distance property allows us to make statements such as 'a test case of the current build has last failed N builds ago' and 'the closest related build (= at least one common test failure) lies N builds in the past'. As a result of this definition, the size of the underlying set of builds of the following elaboration decreases by 7 %, because the set excludes builds, which lack a preceding related build. In other words, 7 % of builds possess a set of failed test cases, none of which has been seen before, which means all of them constitute first occurrences of a particular test failure.

Figure 5.8 compares the fault detection efficiency of the best-performing matrix-based prioritization technique to the performance of the best baseline heuristic with respect to different classes of failure distances. We recognize that the prioritization efficiency for builds with failure distances of up to 51 builds varies only slightly around 95 % APFD. This means, that the change lists, also being a potentially noisy signal, do not impede the performance of the matrix-based technique. Further, the matrix-based prioritization shows superior performance by a margin of 10 % APFD over the baseline heuristic in three failure distance categories ($51 - 101$, $151 - 201$, $251 - 301$). This shows that spreading the memory of faults not only across time, but also across files, can help prioritization performance on repeating, distant failures. Within the failure distance buckets $101 - 151$ and $201 - 251$, the *recently-failed* heuristic shows better performance characteristics. These data points stem from `sonarqube` and `sling` projects and show that in some instances, the low test failure cardinality per build helps the performance of *recently-failed*.

In case there are few test cases with recent failures, the algorithm will assign historic relevance similar to those shown in the sketch in Figure 5.7. Because the rank constitutes the resulting priorities, they are independent of the differences in historic relevance. Test cases three and four will occupy a top rank in the prioritized test schedule, even though the last failure may be located hundreds of builds ago. In combination with a low cardinality set of preceding test failures, this behavior makes *recently-failed* effective in predicting repetitions of distant test failures, even though the failures did not occur recently.

Also, Figure 5.8 shows that the matrix-based technique is in principle capable of correctly predicting failures with distances exceeding 450 builds. However, Figure 5.9 shows that the evidence is theoretical because the vast majority of test failures permeate within 50 builds. As such, improvements in the prediction performance of test failures of higher fault distance do not contribute significantly to the overall fault detection efficiency of this dataset.

[1] In agreement to Section 4.1, *prior* builds are builds, that did not only start before the current build but also have completed by the time the current build began.

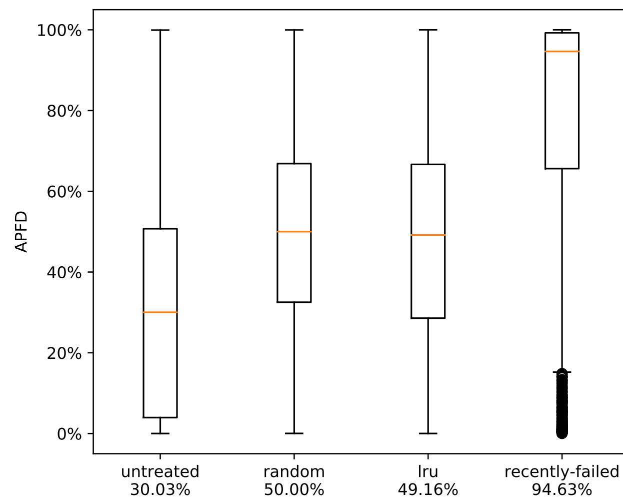


Figure 5.4.: Performance of baseline strategies

Figure 5.5.: Failure histories for the `okhttp` project. Each red bar indicates a test failure for a given build number.

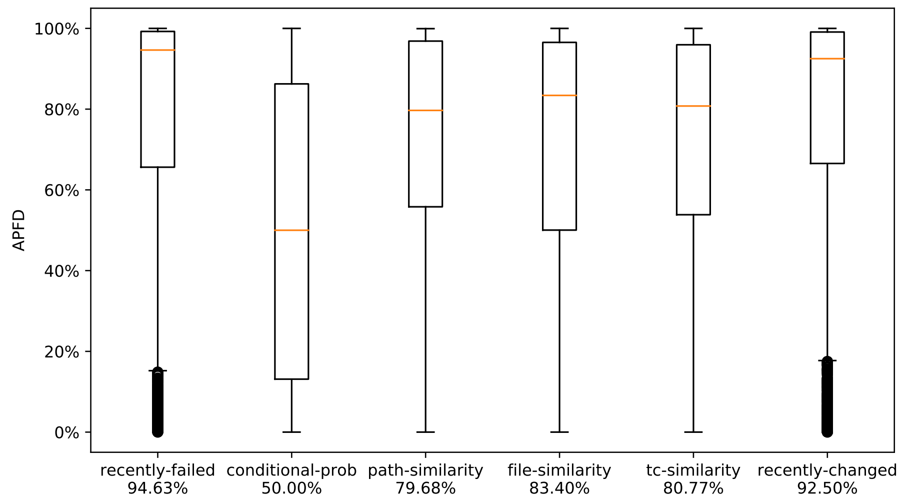


Figure 5.6.: Matrix Prioritization Performance

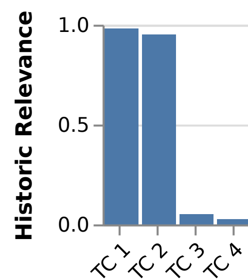


Figure 5.7.: *Recently-failed* assigns the ranks regardless of actual historic relevance. In case the number of distinct failing test classes per build is low, this represents an effective prioritization even though the failures did not occur recently.

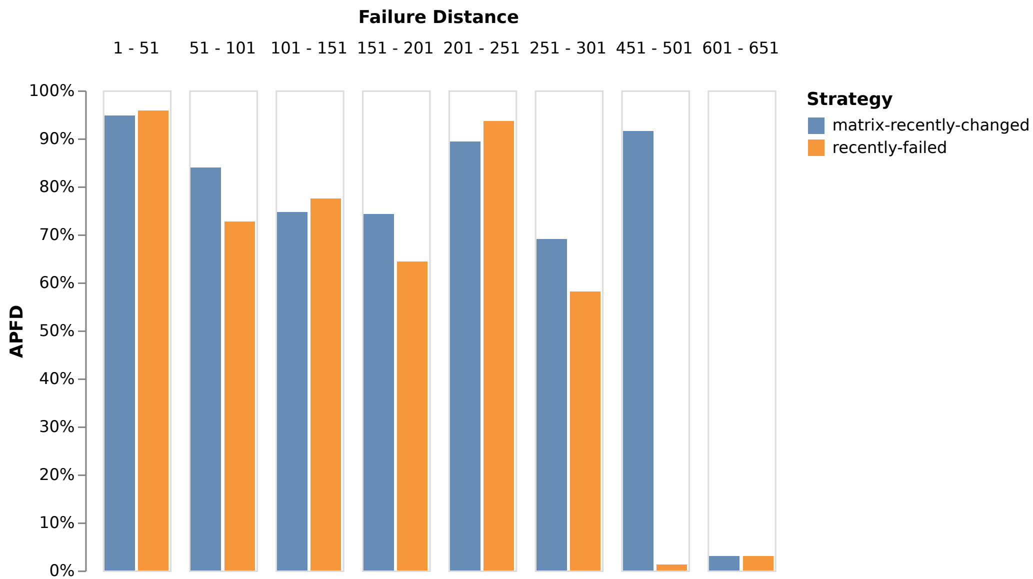


Figure 5.8.: Fault detection efficiency (median APFD) of strategies *recently-changed* (left) and *recently-failed* (right) given different fault distance ranges

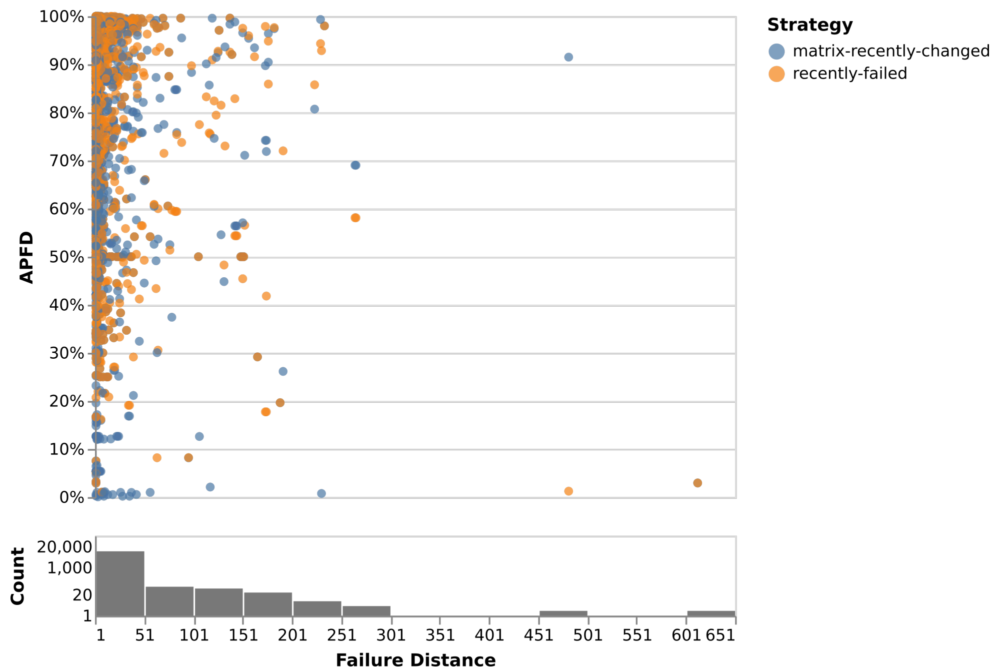


Figure 5.9.: Distribution of failure distances and APFD values (top) and number of data points in different fault distance ranges (bottom)

5.5. Discussion

In the following, we discuss the findings of the individual experiments that revolved around the state of regression testing in open-source projects, the performance of existing approaches, and matrix-based prioritization. We also extrapolate further observations.

Using only failure events, simple heuristics approximate optimal test plans.

We observe that failure events sufficiently approximate the optimal test ordering. The prioritization heuristics achieve good performance, in the dismissal of age and duration information, precise authorship information, code coverage data, call trees, dependency graphs, or change-level textual similarity scores. As a result, the remaining gap towards the optimal test ordering of 3 % APFD leaves only room for minor improvements. We suspect that advanced models, for example, machine-learning approaches, are seldom worth the added complexity, considering their potential contribution to this dataset.

Also, related works suggest combining history-based with diversity-based approaches [22]. In the latter, the algorithm does not prioritize similar test cases, but rather tries to find test cases of maximum dissimilarity. This way, the diversity-based algorithm reaches different parts of a software system quickly, rather than exercising sets of test cases which target the same fault. While diversity-based approaches alleviate potential problems of history-based approaches, such as the cold-start problem, or prediction of unprecedented test failures, the upper bound to their potential contribution to fault detection efficiency on this dataset is also 3 % APFD.

Aspects of time and duration are of secondary importance to RTP.

We observe that the existing heuristics, as well as the set of newly developed approaches, have good performance characteristics and do not employ information about test duration. Because the over-arching objective of RTP consists of faster feedback for developers and testers, this fact surprises us. However, this study revealed that failing test cases took equally long and that a majority of jobs resulted in one failing test case. For the presented reasons, duration-aware test execution plans differ only by a small margin from test plans, which do not incorporate timing information. Therefore, we do not assign relevance to duration-aware prioritization in this particular context of open-source projects.

Also, we observe that prioritization algorithms of reasonable performance do not employ information about points in time, for example, how much time has passed between regression test sessions or the timely distance between failures of the same test case. This way, all presented algorithms are independent of development frequency and their fault detection efficiency does not deteriorate in case of stalled development activity. This is in contrast to machine-learning approaches [5, 29], which discretize periods of up to two months for determining test failure rates, for example. To approximate the optimal test plan sufficiently, the algorithms relied only on ordered sets of test events without timing information.

Matrix-based techniques can predict distant faults.

The experiments in Section 5.4.3 show how the failure patterns in our dataset slightly work to the advantage of the *recently-failed* heuristic. In the following, we take the opposite route and describe a setting in which matrix-based prioritization methods provide better fault detection performance. In particular, those methods benefit from a combination of the following parameters to failure patterns:

1. *Intermittent Test Failures.* As the failure distance experiment shows, we find evidence that matrix-based strategies can predict re-emerging test case failures based on the correlations between changes and the induced test failures. Because matrix-based strategies do not necessarily allocate top ranks to test failures of the last test session, the ability to predict faults improves over *recently-failed*, when the test cases fail intermittently (as opposed to consecutive streaks of test failures in Figure 5.5). In case of alternating test failures, the prediction of the *recently-failed* algorithm is one step behind, whereas a matrix-based technique can recognize the inducing change and prioritize accordingly.

2. *Increased Failure Distances.* Because test failures go out-of-scope of the *recently-failed* heuristic, as the distance between failures of the same test case enlarges and the overall cardinality of the distinct set of failing test cases increases, matrix-based algorithms can likely provide better prioritization in instances of distant repeating test failures.

3. *Edge Changelists.* After a prolonged series of regression test sessions without test failures, all test cases appear to be of low historic relevance to *recently-failed*. In those instances, the prioritization performance of *recently-changed* is likely better, because the strategy can update its model without relying solely on failure events. The *recently-changed* algorithm can learn about recently churned code from the change lists of green builds and prioritize tests for the next test session accordingly, which may contain an edge change list (discontinue the streak of green builds).

In summary, the prioritization of matrix-based techniques can benefit from prolonged failure distances and higher cardinality of the distinct set of failing test cases, over the simpler *recently-failed* heuristic.x

Matrix-based models are useful beyond effective prioritization.

While the family of matrix-based approaches achieves better prioritization performance on only 25 % of projects, we argue that there are qualitative arguments. For instance, we make the case that files indeed connect to failures. To surpass the best baseline heuristic, it would only require distant test failures to re-emerge. Because matrix-based approaches can connect the new change to priorly seen failures, they would still place relevant test cases first. In contrast, the performance of the *recently-failed* heuristic would deteriorate, because failures of the re-emergent pattern are too far in the past and therefore have a very small influence on the

current prediction. Second, the matrix-based approaches can deliver predictions for parts of the software system. Due to the added flexibility, the matrix models enable several use cases. For example, testers may limit testing to a subset of modules and need a prediction concerning only the selected modules. Also, build tools may choose to test modules in parallel and can make use of individual predictions for each of the modules. Due to the inexpensiveness of the approach, developers may observe changes in test prioritization, as they grow their change list during the editing process and engage in pre-submit testing. Lastly, we envision that the matrix-based models are useful beyond regression test prioritization. After this data structure becomes available, it can be used for several test maintenance activities. For instance, a file-to-failure matrix can be used for deflaking purposes. Because flaky tests would show an abnormal failure distribution, they would become easily identifiable. This is just one example of a "test smell" [36] which can become detectable using the said data structure. We further explore the solution space of regression test maintenance using file-to-failure-matrices in Section 7.

5.6. Threats to Validity

In the following, we discuss threats to the validity of this study.

Threats to Internal Validity

The first threat to validity are faults within our implementation. We mitigate this threat by reusing standard components and by unit-testing the prioritization strategies extensively. To support reproducibility, our implementation is available online.[2] Further threats to internal validity consist of trade-offs within the parsing of build log files. We develop two parsers, for the Apache Maven and Buck build systems, to reconstruct regression test sessions from build logs. This works on a best-effort basis; in particular, there are instances of interleaved test log lines, which may break the association between test name and test outcome. Furthermore, the parsers may not catch all possible permutations of the textual output of the build systems with regard to test states, duration formats, or naming conventions. We mitigate this threat by checking for the plausibility of parsing results, by probing test log files and by basing our implementation on the existing parsers of TravisTorrent [3]. Also, this study relies on the completeness of build log files, which we cannot assume for a minority of jobs. In case the build process exits prematurely, for example, due to a fatal exception, resource exhaustion, or users aborting the build, the testing process can only be reconstructed partially.

[2] The code used in this report has been published and archived [7].

Threats to External Validity

While we study open-source projects, the selected subset of projects might not be representative. First, we rely on the TravisTorrent dataset as the basis, which links Travis projects to GitHub projects and contains important metadata, such as built Git commits. From there on, our project selection is subject to technical limitations. While the dataset basis contains projects, whose main implementation languages are Ruby, JavaScript, and Java, we found that only two build tools of the latter type of project produce sufficient textual output. In detail, to reconstruct regression test sessions in their entirety, we require both, passed and failed test cases, to be printed. Second, to answer any of the research questions, we rely on the presence of test failures in build logs. This assumption rules out projects, which mute the build tool output, do not possess test automation, or conduct extensive pre-submit testing.

Threats to Construct Validity

Our *test model* does not account for flaky tests, that is, tests that seemingly fail at random, in ignorance of the actual change to the software. While prior works assign value to deflaking tests in training their machine-learning classifier [29], this is not possible due to the abstraction level of our method. We rely on the analysis of build logs. This way, we can inspect actual development histories of prolonged duration independent of build reproducibility concerns, but lose the ability to execute tests for deflaking purposes.

Further, our test model is based on the following assumptions: First, we think of test classes to be independent. We do not assign significance to the untreated test order and do not think of it as deliberate. As a result, we disregard test class dependencies, as we cannot reconstruct them from build logs due to missing evidence. This includes explicit dependencies, in which developers design tests to run in a specific order, and implicit dependencies, in which test classes inadvertently affect each other through side effects.

Second, we disregard concurrent test execution. Neither do we have conclusive evidence if the untreated test order was the result of concurrent test execution, nor do we account for concurrent execution of the resulting prioritized test execution plan. Assuming test linearity prevents us from making further assumptions, such as the degree of parallelism. We think that research conducted on a linear test ordering extends well to a concurrent setting; after all, likely test failures still have to execute first.

Lastly, we disregard module boundaries and assume that, once all modules of a multi-module project are compiled, we are free to pick any of the test classes for execution. Assuming that test prioritization may freely reorder tests stands in contrast to build tools, which may decide to interleave testing and compilation phases of different modules. Because other build steps, such as compilation, are not part of our cost model, we can make this simplifying assumption. Also, if test prioritization saves cost in a single monolithic testing session, distributing the

savings proportionally across multiple, module-specific test runs will not diminish the return on investment.

Our *fault model* treats each test case failure as equivalent to a real-world fault. In reality, a single fault may manifest in none, one, or many test case failures. Due to working on build log files, we cannot locate or enumerate real-world faults; and if we were to work on actual source trees, doing so for the presented number of revisions and projects would be prohibitively expensive. Also, due to the granularity of the textual output of test runners, this elaboration can only reason about faults on the level of a test class. Because log files only contain method counters, such as the number of passed or failed instances, we cannot determine if production code changes correlate to subsets of methods of a test class.

6. Exploratory Study on Test Prioritization for Continuous Testing

In the following, Section 6.1 introduces *AutoTDD*, a tool for continuous test execution in Squeak/Smalltalk. We continue to detail how we extended the tool for test prioritization purposes and report preliminary findings of using the enhanced *AutoTDD* version in Section 6.2. Because the early results are promising, we deem it worthwhile to deepen our understanding of test prioritization within development environments, and lay out potential user study designs in Section 6.3.

6.1. Introduction to *AutoTDD*

Figure 6.1 shows *AutoTDD*, which allows continuously executing tests inside Squeak/Smalltalk. Its user interface follows the metaphor of a traffic light for tests: a green check mark indicates that the unit tests are currently passing, a yellow exclamation mark signifies that some tests are failing due to an unmet assertion, and a red one tells the developer about runtime errors.

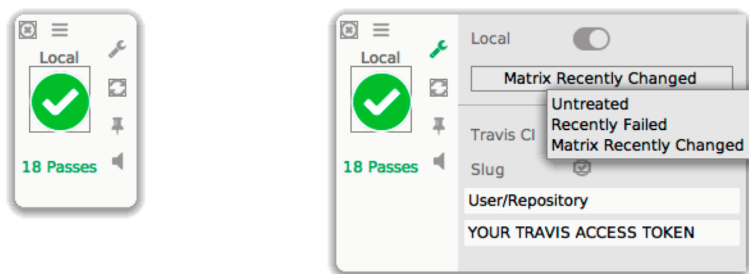


Figure 6.1.: *AutoTDD* traffic light in collapsed mode (left) and with settings pane and open menu for strategy selection (right)

In addition to visual feedback, *AutoTDD* can also play a distinctive sound, in event that a test session completed and the status of the test suite has changed. The application triggers new test runs, as soon as the developer has modified the implementation of a method in the browser and confirms (saves) the change. This way, the size of the change is quite small and the developer can perceive the effect on the unit test suite almost immediately. Because developers can connect changes and test outcomes during this interleaved edit and testing process, they spent less

time on fault localization and are therefore more productive [46]. The existing implementation of *AutoTDD* does not practice test selection or prioritization and executes all tests of a configured subset of packages in the untreated test order.

6.2. Extensions of *AutoTDD*

We extend *AutoTDD* to include an extensible set of prioritization strategies, which initially consists of the untreated test order, *recently-failed*, the best-performing baseline heuristic, and *recently-changed*, the best performing matrix-based approach.

Because the concept of source files is missing the image-based environment of Squeak/Smalltalk, they are not available as unit of change to the matrix-based strategies. As a result, the *recently-changed* algorithm uses the name of the class as the identifier for the element the developer modified. Additionally, we include a way to visualize the current fault detection efficiency trend. Figure 6.2 shows a live-updating bar chart, which compares the efficiency of the untreated and the matrix-based strategy, using the median APFD values of an exemplary development session (multiple test runs).

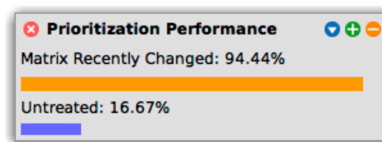


Figure 6.2.: Comparison of APFD values of an exemplary test session

We use *AutoTDD* with test prioritization to fuel further testing efforts of *AutoTDD* and the prioritization strategies themselves. Despite the fact that the observed number of instances is too low to be representative, we believe that this early type of experiment incentivizes further research and tool development for next-level regression testing within development environments. The insights from the continuous integration setting appear to translate well to the new circumstances. Table 6.1 details test prioritization using *AutoTDD* and clearly shows that test prioritization efforts substantially increase fault detection efficiency in comparison to the untreated test plan.

6.3. Proposed Research Directions

Due to the promising perspective resulting from our work on and with *AutoTDD*, we recognize the need for further user studies. Those study proposals address needs, which arise from the combination of continuous testing techniques with test prioritization. In the following, we propose questions regarding the agree-

Table 6.1.: Key statistics for test prioritization using *AutoTDD*

Description	Value
Test Sessions	271
Red Test Sessions	19.92 %
Median *untreated* APFD	61 %
Median *recently-changed* APFD	97 %

ment between human and automated test discovery, the impact of prioritized and continuous testing processes on test-driven development, and realized cost savings.

Test Discovery and Execution Policies

To prioritize tests more effectively within a development environment, we propose to collect rationales for human test discovery, like naming conventions, directory layouts, and prior experience. Such a study has to validate if existing techniques, which have been primarily developed with CI systems in mind, model those motives exhaustively. Quantitatively, this study can ask developers to assess the appropriateness of test plans, which are the result of prioritization heuristics. Also, we assign relevance to investigating the perceived immediacy of test results. Because the changeset considered by *AutoTDD* is at the method level, the application spawns new test sessions frequently. In case a new edit has completed, *AutoTDD* aborts the pending test run and schedules a new one. This behavior results in generally higher processor load and stands in contrast to the fact that a meaningful software change oftentimes consists of changing more than one method. A study should investigate if debouncing test execution can save on resources without impacting the immediacy of test results as perceived by the developer.[1] This represents only one example of an alternative to the 'run as many tests as fast as possible' policy and questions how to determine the appropriate pace of prioritized, continuous test execution within *AutoTDD*.

Test-Driven Development

Given that real-world studies concerning the implementation the Test-Driven Development (TDD) [17] process exist and that these studies provide us with an inference model for detecting TDD usage from instrumented development environments [2], how do both continuous and prioritized test execution impact adoption of TDD? Also, prior works find that tests and production code do not co-evolve gracefully,

[1] The only earlier research we know employs tactics such as "execute tests after 5 seconds of idle time if at least 15 seconds passed or 30 keystrokes occurred since the last run", which appears arbitrary [45].

a problem that a tool like *AutoTDD* can potentially address using the immediate feedback mechanism.

We can only hypothesize that relieving the developer from the burden of test discovery, test selection, and test execution, is likely to make a test-driven or test-first approach to software development more attractive. Getting an immediate hold of disagreeing test and production code would also remove a lot of friction in the development process. The proposed study has to extend the TDD inference model to account for the asynchronous test execution of *AutoTDD* and observe developer behavior in presence of prioritized, continuous test execution.

Hours Saved Metric

The computation of the *hours saved* metric is trivial (the difference between the time to detect all faults in the untreated versus prioritized test order) but it is usually not meaningful on its own. When comparing the time savings of individual test sessions, the return on investment may appear minuscule and not worthwhile of the prioritization effort. However, this conception does not account for environmental circumstances. The yield of test prioritization increases considerably when accounting for multiple developers working on the same software product, the overhead of context switching when working on different features simultaneously and the development time needed to locate faults once tests have been skipped to ship faster.

While a prior work finds that participants of a controlled study are three times more likely to complete an assignment correctly using continuous testing, the participants did not complete the task faster [45]. However, the applicability is limited, because respondents had been working towards a fixed deadline and the test suite was too small for test prioritization. A new study has to address these two concerns and has to investigate how much time savings are realistic for single developers and teams.

7. Future Work

In the following, Section 7.1 lays out extensions to matrix-based test prioritization, which require further research to quantify their impact and usefulness. Also, we envision improvements to the software development process and tooling using history-based regression test prioritization techniques. While previous chapters covered this aspect tangentially, Section 7.2 extends the scope beyond what was detailed in the reference implementation chapter (Section 6). Lastly, even if the computational overhead of the proposed test prioritization method appears negligible, creating the data structure cannot be assumed to be free of expenses. Therefore, we investigate how file-to-failure-matrices may be useful beyond the scope of regression test prioritization in Section 7.3 by identifying abnormal test behavior.

7.1. Extensions to Matrix-based Test Prioritization

File Type Oracles The proposed set of matrix-based approaches correlates changes in files to test failures. The algorithm assumes a uniform set of files and does not make further distinctions and assumptions. This way, the approach is broadly applicable and remains portable across different programming languages and environments. However, it remains to be investigated, if additional gains in prioritization performance are possible for a subset of file types. In particular, we think it might be fruitful to give up some of the abstraction and develop language-specific oracles, which categorize files into "production code", "test code", or "configuration/resource file". Because of their different nature, files of different categories may expose different test failure behavior and therefore may require the use of different weighting functions.

Heterogenous Change Lists Prior elaboration focused on files, which are tangible artifacts of the software development process and are well understood. We think that files are worthy units of abstraction, but can be easily replaced with other concepts. Because image-based programming environments, such as Squeak/Smalltalk, do not possess files, exchanging them for classes has already been a precondition to the successful completion of the reference implementation presented in Chapter 6. Also, we envision heterogeneous change lists, in which not only the code changes in the source tree of the current project are present, but also selected parts of the environment. In addition to files, changes to third-party dependencies, such as programming libraries, could be part of the change list. This would allow regression testers to formulate requirements such as "rerun all tests, which previously failed due to changes in the API of the third-party computer

graphics library". Furthermore, we might also extend the reach of change lists to platform changes, such as patches to the operating system or upgrades to the compiler or interpreter infrastructure. This could help reveal tests sensitive to these types of changes and prioritize those tests accordingly.

Model Bootstrapping In principle, history-based techniques suffer from a cold-start problem. After the deployment of a new prioritization technique, no or little historic data is present and prioritization performance can be weak. Because the quality of the prediction is mainly dependent on the distance of the failures of the same test case, the decline is project dependent. Prior works find that prioritization performance increases noticeably, as soon as more observations become available [22]. Therefore, we assume that bootstrapping matrix-based models is beneficial in the early stages of adoption and enabled by using means of mutation testing [6]. In this scenario, a mutation agent syntactically modifies a source file and observes if a test case signals a failure. This way, the test case "kills" the mutant and detects a software modification, which might be detrimental to functioning software. In turn, the modified source file forms a new change list and the observed test failures constitute new entries in the file-to-failure-matrix. In summary, more research is required to determine if matrix-based approaches are amenable to bootstrapping through mutation testing and how this affects prioritization right after deployment.

7.2. Software Development Process Improvements

While the benefits of regression test prioritization have already been discussed at length within a continuous integration setting, we propose to extend the benefits to development environments. Utilizing the set of inexpensive, matrix-based prioritization approaches, it is possible to close the feedback loop on unit tests. In principle, there are two directions within a development environment: automation and manual intervention.

Viable test automation knows which files have been changed and automatically re-executes high-risk test cases after a source file has been saved to disk. This way, fast unit tests can become a valuable source of feedback to software developers, which becomes available quickly, similar to how syntax errors are highlighted during typing program source code. We investigated the potential of this type of *continuous testing* [46, 50] within our reference implementation inside *AutoTDD* (see Section 6) and are looking forward to future developments.

But also when the test suite is too slow for near-realtime feedback, we argue that tools within a development environment, which require manual intervention, can help foster programmers' productivity. Integrating a widget inside the development environment, which displays currently relevant test cases with a live update, would directly address the problem of human test discovery. We hypothesize that developers rarely practice test case selection beyond simple naming conventions and are seldom aware of all modification-traversing test cases. Therefore, such

integration into the development environment would be worthwhile, because it encourages test execution before going through the hurdles of bundling changes into commits, publishing it to a central repository, and accessing a CI server. We also envision that seeing a new test case suddenly being prioritized may encourage developers to run it to check if it still passes. However, such a user study is outside the scope of this project and therefore remains for future work.

7.3. Test Maintenance Support

We envision that once the file-to-failure-matrices are available, several use cases become viable, which involve test maintenance. In particular, we lay out three areas of research directions: First, how connecting changes to test failures can help identify and fix test smells; second, we frame the concept of sensitivity, which describes how interdependence between code change and test failure may appear irregular; and third, we outline implications of very similar failure distributions.

Test Smells Similar to a code smell, a *test smell* is a symptom of a problem with the unit test suite that regression test practitioners may easily spot [15]. While smells neither directly point to a root cause nor always be indicative of a problem, they appear irregularly and deserve deeper investigation by developers. One example is a *flaky* test: A test case that fails regardless of the change to software [36]. Reasons for occasional test failures are manifold and include bad assumptions about the environment (ordering of records in a database or ignorance of daylight saving time), randomizing test data, tainting the program's global state, relying on non-deterministic concurrency, and many more [47]. Because the matrix-based prioritization models connect a stimulus (file changes) to test outcomes, we suggest it is possible to develop a new, history-based approach for determining test flakiness, which does not rely on manifold re-execution of test cases [28]. To this end, an algorithm would have to identify test cases, whose failures almost evenly spread over all changed files. Using this information, an automatied process could easily flag suspected test cases and a suitably augmented test runner could quarantine them until developers investigate the root cause. As a result, every build containing a potentially flaking test would directly contribute to the test statistics and the need for aggressive deflaking using repeated test executions would decline.

Sensitivity Also using file-to-failure-matrices, we describe the concept of sensitivity. For instance, a test case is overly sensitive, when the failure of the test case consistently depends on the same large set of files. Because this means the test case covers a comparatively large share of the code base, test maintenance cost might increase due to difficulties in fault localization. Also, the asserted test outcome might be overly broad and might be insufficient to check the correct function of the substantial amount of covered code. The concept of sensitivity also translates to files, which inevitably make a large fraction of the test suite fail when changed. We reason that the underlying file might be overly covered, either because the file

constitutes a legitimate hotspot of verification efforts or because the file contains overly critical or brittle, code. Because of the need for corrective action, we argue that identifying overly sensitive files and test cases is interesting to maintenance programmers and that such phenotypes should be considered code smells or test smells.

Similarity Another possible research direction concerns the investigation of similar failure distributions, either of files or test cases, which can be used to cluster the respective sets. Because test cases, which possess very similar failure distributions, likely localize the same faults, test engineers might consider them duplicates. As a result, those test cases are subject to elimination during test set reduction without impeding fault detection effectiveness. In turn, a development team may save on test code maintenance cost and experience shorter test suite runtime. A similar notion extends to sets of files with very similar failure distributions. They may resemble overly coupled or brittle units of code, which developers have to modify together to green the test suite. While some instances may be the result of the underlying software architecture, developers may deduce the need for refactoring.

8. Conclusion

Regression testing continues to be one of the most common verification and validation activities and is one of the core building blocks to successful continuous integration and continuous deployment practices. However, as software grows in size and complexity, we evidence scalability problems concerning time and resources dedicated to testing activities. This report shows that those problems extend to small to medium-sized open-source projects and finds evidence in a newly derived dataset. The reconstructed test session resembles reality much closer and therefore supersedes academic subject programs with respect to fault quality. Also, this dataset enables studying a more diverse set of projects, than existing test collections, which are the result of verification efforts on large-scale, homogeneous software suites. Using the new dataset, we extensively study prior approaches in the field of history-based prioritization techniques and find that those heuristics cope well with a dataset originating in continuous integration. The newly proposed set of matrix-based approaches addresses a major shortcoming of existing heuristics, namely the static prediction, while maintaining or improving, the fault detection efficiency of the resulting test execution plans. At the same time, the novel techniques retain the advantage of the negligible computational overhead of existing heuristics. Without compromising prediction performance, their barriers to successful adoption and operation are considerably lower than state-of-the-art machine learning approaches. We lay out how the proposed models can be useful beyond the purpose of regression test prioritization and how file-to-failure-matrices can be useful to test maintenance activities.

While the RTP efforts in this report target shorter feedback cycles within continuous integration, we believe CI systems are only step stones to greater adoption of test prioritization as a whole. Figure 8.1 shows the perceived immediacy of feedback on tests. Obviously, continuous testing within development environments provides the tightest feedback loop. Occasional testing already relies on the developers' dedication to testing and because builds are started afresh on CI systems they are the most expensive way of obtaining feedback.

Figure 8.1.: Perceived immediacy of regression test results

While the immediacy of test results declines from development to CI environments, we anticipate that the rate of adoption of test prioritization will behave reciprocally. At first, heavily automated test systems will benefit from test prioritization and subsequently, prioritization will approach development environments, and influence developer behavior. This way, we project that regression test prioritization in continuous, as well as development environments, can significantly contribute to optimizing developers' workflow by making automated tests first-class citizens.

A. Source Code

The programming language of below listings is Kotlin [26].

Listing A.1: Recently Failed

```
class RecentlyFailedStrategy(val alpha: Double) : PrioritizationStrategy {

 private val histories = mutableMapOf<String, BitSet>()

 override fun reorder(p: Params): List<TestResult> {
  val priorities = mutableMapOf<String, Double>()

  for (tc in p.testResults) {
   val history = historyFor(tc)
   priorities[tc.name] = getValue(history, p)
  }

  return p.testResults.sortedByDescending { priorities[it.name] }
 }

 override fun acceptFailedRun(p: Params) {
  for (tc in p.testResults) {
   if (tc.red > 0) {
    historyFor(tc).set(p.job.jobNumber)
   }
  }
 }

 private fun historyFor(tc: TestResult) =
  histories.computeIfAbsent(tc.name) { BitSet() }

 private fun getValue(history: BitSet, p: Params): Double {
  var prob = 0.0

  for (job in p.priorJobs) {
   prob = alpha * historyAt(history, job) + (1 - alpha) * prob
  }

  return prob
 }

 private fun historyAt(history: BitSet, job: Job) =
  if (history[job.jobNumber]) 1.0
  else 0.0
}
```

Listing A.2: Matrix Data Structure

```
@Serializable
data class Matrix(private val matrix: Map<Key, Double>) :
  Map<Key, Double> by matrix {
    companion object
}

@Serializable
data class Key(val fileName: String, val testName: String)
```

Listing A.3: Unit Matrix Construction

```
class UnitMatrix(
  val repository: Repository,
  val cache: Cache
) {

  fun get(job: Job): Matrix {
    return cache.get(job, ::computeMatrix)
  }

  private fun computeMatrix(job: Job): Matrix {
    val changedFiles = repository.changedFiles(job)
    val testResults = repository.testResults(job)
    return createUnitMatrix(changedFiles, testResults)
  }

  private fun createUnitMatrix(changedFiles: List<String>,
                               testResults: List<TestResult>): Matrix {
    val matrix = mutableMapOf<Key, Double>()
    for (test in testResults.filter { it.red > 0 }) {
      for (file in changedFiles) {
        matrix.merge(Key(file, test.name),
          test.red.toDouble(), Double::plus)
      }
    }
    return if (matrix.isEmpty()) Matrix.empty() else Matrix(matrix)
  }
}
```

Listing A.4: Reducer: Combining Matrices

```
internal typealias Reducer = (Matrix, Matrix) -> Matrix

class DevaluationReducer(val alpha: Double) : Reducer {

  override fun invoke(left: Matrix, right: Matrix): Matrix {
    val m = mutableMapOf<Key, Double>()
    val commonFiles = left.fileNames.intersect(right.fileNames)

    for (entry in left) {
      val fileIsCommon = entry.key.fileName in commonFiles
      val factor = if fileIsCommon (1 - alpha) else 1.0
      m[entry.key] = factor * entry.value
    }

    for (entry in right) {
      val fileIsCommon = entry.key.fileName in commonFiles
      val factor = if fileIsCommon alpha else 1.0
      m.merge(entry.key, factor * entry.value, Double::plus)
    }

    return Matrix(m)
  }
}
```

Listing A.5: Strategy: *conditional-prob*

```
class ConditionalProbability(
  repository: Repository,
  cache: Cache,
  val reducer: Reducer
) : PrioritizationStrategy {
  private val unitMatrix = UnitMatrix(repository, cache)

  override fun reorder(p: Params): List<TestResult> {
    val unitMatrices = p.priorJobs.map(unitMatrix::get)
    val sumMatrix = unitMatrices.fold(Matrix.empty(), reducer)
    val probabilities = probabilities(p.changedFiles,
                      p.testResults.map { it.name },
                      sumMatrix)
    return p.testResults.sortedByDescending {
          probabilities[it.name] }
  }

  internal fun probabilities(changedFiles: List<String>,
    tests: List<String>,
    sumMatrix: Matrix
  ): Map<String, Double> {

    val fileCounts = fileCounts(sumMatrix)
    return tests.associateWith { tc ->
      prob(changedFiles, sumMatrix, fileCounts, tc)
      }
  }

  private fun fileCounts(m: Matrix):
   Map<String, Pair<Double, Double>> {
    val counts = mutableMapOf<String, Double>()
    for (entry in m) {
      counts.merge(entry.key.fileName, entry.value, Double::plus)
    }

    val sum = counts.values.sum()
    return counts.mapValues { entry ->
       entry.value to (entry.value / sum)
    }
  }

  private fun prob(
    changedFiles: List<String>,
    m: Matrix,
    fileCounts: Map<String, Pair<Double, Double>>,
    tc: String
  ): Double {

    return changedFiles.map { file ->
      val (fileCount, fileProbability) =
        fileCounts[file] ?: Pair(0.0, 0.0)
      val count = m[Key(file, tc)] ?: 0.0
      fileProbability * (count / fileCount)
    }.sum()
  }
}
```

Listing A.6: Strategy: *recently-changed*

```
class RecentlyChanged(
  repository: Repository,
  cache: Cache,
  val reducer: Reducer,
  val alpha: Double
) : PrioritizationStrategy {

  private val unitMatrix = UnitMatrix(repository, cache)
  private val histories = mutableMapOf<String, BitSet>()

  override fun reorder(p: Params): List<TestResult> {
    p.changedFiles.forEach {
      histories.computeIfAbsent(it) { BitSet() }.set(p.job.jobNumber)
    }
    val unitMatrices = p.priorJobs.map(unitMatrix::get)
    val sumMatrix = unitMatrices.fold(Matrix.empty(), reducer)
    val filePriorities = priorities(p.priorJobs + p.job, sumMatrix)

    val testPriorities = p.testResults.associateWith { tc ->
      sumMatrix.filterKeys { it.testName == tc.name }
        .map { filePriorities.getOrDefault(it.key.fileName, 0.0)
               * it.value }
        .sum()
    }

    return p.testResults.sortedByDescending { testPriorities[it] }
  }

  internal fun priorities(priorJobs: List<Job>, m: Matrix):
   Map<String, Double> {
    return m.fileNames().associateWith { similarity(priorJobs, it) }
  }

  private fun similarity(priorJobs: List<Job>, fileName: String):
   Double {
    return getValue(histories.computeIfAbsent(fileName) { BitSet() },
      priorJobs)
  }

  private fun getValue(history: BitSet, priorJobs: List<Job>):
   Double {
    var prob = 0.0

    for (job in priorJobs) {
      prob = alpha * historyAt(history, job) + (1 - alpha) * prob
    }

    return prob
  }

  private fun historyAt(history: BitSet, job: Job) =
    if (history[job.jobNumber]) 1.0
    else 0.0
}
```

Listing A.7: Strategy: *file-similarity*

```
class FileFailureDistributionSimilarity(
  repository: Repository,
  cache: Cache,
  val reducer: Reducer
) : PrioritizationStrategy {

  private val unitMatrix = UnitMatrix(repository, cache)

  override fun reorder(p: Params): List<TestResult> {
    val unitMatrices = p.priorJobs.map(unitMatrix::get)
    val sumMatrix = unitMatrices.fold(Matrix.empty(), reducer)
    val priorities = priorities(p.changedFiles, sumMatrix)
    return p.testResults.sortedByDescending { priorities[it.name] }
  }

  internal fun priorities(
    changedFiles: List<String>,
    matrix: Matrix
  ): Map<String, Double> {
    val fileToSimilarity = similarity(changedFiles, matrix)

    return matrix.testNames().associateWith { tc ->
      matrix.filterKeys { it.testName == tc }
          .map { entry -> (fileToSimilarity[entry.key.fileName] ?: 0.0)
                       * entry.value }
          .sum()
    }
  }

  private fun similarity(
    changedFiles: List<String>,
    m: Matrix
  ): Map<String, Double> = m.fileNames().associateWith { file ->
    changedFiles.parallelStream().mapToDouble { changedFile ->

      val a = m.testDistribution(changedFile)
      val b = m.testDistribution(file)
      var dot = 0.0
      var normA = 0.0
      var normB = 0.0

      for ((va, vb) in a.zip(b)) {
        dot += va * vb
        normA += va.pow(2.0)
        normB += vb.pow(2.0)
      }

      dot / (sqrt(normA) * sqrt(normB))
    }.max().orElse(0.0)
  }
}
```

Bibliography

[1] M. Azizi and H. Do. "A Collaborative Filtering Recommender System for Test Case Prioritization in Web Applications". In: *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. ACM. 2018, pages 1560–1567.

[2] M. Beller, G. Gousios, A. Panichella, and A. Zaidman. "When, How, and Why Developers (Do Not) Test in Their IDEs". In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM. 2015, pages 179–190. DOI: 10.1145/2786805.2786843.

[3] M. Beller, G. Gousios, and A. Zaidman. *GitHub: TravisTorrent Tool Suite*. 2019. URL: https://github.com/TestRoots/travistorrent-tools (visited on 2019-05-27).

[4] M. Beller, G. Gousios, and A. Zaidman. "TravisTorrent: Synthesizing Travis CI and GitHub for Full-Stack Research on Continuous Integration". In: *Proceedings of the 14th Working Conference on Mining Software Repositories*. 2017, pages 447–450. DOI: 10.1109/MSR.2017.24.

[5] B. Busjaeger and 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*. ACM. 2016, pages 975–980. DOI: 10.1145/2950290.2983954.

[6] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. "Hints on Test Data Selection: Help for the Practicing Programmer". In: *Computer* 11.4 (1978), pages 34–41.

[7] F. Dürsch, T. Mattis, P. Rein, and R. Hirschfeld. *Implementation of History-based, Lightweight Test Prioritization Techniques*. 2022. DOI: 10.5281/zenodo.6518365.

[8] S. Elbaum, A. Malishevsky, and G. Rothermel. "Incorporating Varying Test Costs and Fault Severities into Test Case Prioritization". In: *Proceedings of the 23rd International Conference on Software Engineering*. IEEE. 2001, pages 329–338. DOI: 10.1007/978-3-642-02949-3_5.

[9] S. Elbaum, G. Rothermel, and J. Penix. "Techniques for Improving Regression Testing in Continuous Integration Development Environments". In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM. 2014, pages 235–245. DOI: 10.1145/2635868.2635910.

[10] E. Engström, P. Runeson, and M. Skoglund. "A Systematic Review on Regression Test Selection Techniques". In: *Information and Software Technology* 52.1 (2010), pages 14–30. DOI: 10.1016/j.infsof.2009.07.001.

[11] Facebook. *Buck: a fast build tool*. 2019. URL: https://buckbuild.com/ (visited on 2019-03-18).

[12] N. Fazeli. "Machine Learning to Uncover Correlations Between Software Code Changes and Test Results". Master's thesis. Chalmers University of Technology, University of Gothenburg, 2017.

[13] Y. Fazlalizadeh, A. Khalilian, M. A. Azgomi, and S. Parsa. "Incorporating Historical Test Case Performance Data and Resource Constraints into Test Case Prioritization". In: *International Conference on Tests and Proofs*. Springer. 2009, pages 43–57.

[14] A. S. Foundation. *Apache Gains Additional Travis-CI Capacity*. 2015. URL: https://blogs.apache.org/infra/entry/apache_gains_additional_travis_ci (visited on 2019-05-08).

[15] M. Fowler. *CodeSmell*. 2006. URL: https://martinfowler.com/bliki/CodeSmell.html (visited on 2019-05-29).

[16] M. Fowler. *Continuous Integration*. 2006. URL: https://martinfowler.com/articles/continuousIntegration.html (visited on 2019-05-08).

[17] M. Fowler. *Test-Driven Development*. 2005. URL: https://martinfowler.com/bliki/TestDrivenDevelopment.html (visited on 2019-07-03).

[18] GitHub, Inc. *GitHub*. 2019. URL: https://github.com (visited on 2019-03-18).

[19] M. Gligoric, L. Eloussi, and D. Marinov. "Practical Regression Test Selection with Dynamic File Dependencies". In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM. 2015, pages 211–222. DOI: 10.1145/2771783.2771784..

[20] J. Gosling, B. Joy, G. Steel, G. Bracha, A. Buckley, and D. Smith. *The Java Language Specification*. 2019. URL: https://docs.oracle.com/javase/specs/jls/se12/html/jls-7.html#jls-7.6 (visited on 2019-04-01).

[21] G. Gousios. "The GHTorrent Dataset and Tool Suite". In: *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR) 2013*. Piscataway, NJ, USA: IEEE, 2013, pages 233–236. ISBN: 978-1-4673-2936-1.

[22] A. Haghighatkhah, M. Mäntylä, M. Oivo, and P. Kuvaja. "Test Prioritization in Continuous Integration Environments". In: *Journal of Systems and Software* 146 (2018), pages 80–98. DOI: 10.1016/j.jss.2018.08.061.

[23] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi. "Regression Test Selection for Java Software". In: 36.11 (2001), pages 312–326. DOI: 10.1145/504311.504305.

[24] M. Hilton, N. Nelson, T. Tunnell, D. Marinov, and D. Dig. "Trade-offs in Continuous Integration: Assurance, Security, and Flexibility". In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM. 2017, pages 197–207. DOI: 10.1145/3106237.3106270.

[25] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig. "Usage, Costs, and Benefits of Continuous Integration in Open-source Projects". In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM. 2016, pages 426–437. DOI: 10.1145/2970276.2970358.

[26] JetBrains s.r.o. *Kotlin Programming Language*. 2019. URL: https://kotlinlang.org (visited on 2019-07-29).

[27] J.-M. Kim and A. Porter. "A History-based Test Prioritization Technique for Regression Testing in Resource Constrained Environments". In: *Proceedings of the 24th International Conference on Software Engineering*. ACM. 2002, pages 119–129. DOI: 10.1145/581339.581357.

[28] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie. "iDFlakies: A Framework for Detecting and Partially Classifying Flaky Tests". In: *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE. 2019, pages 312–322. DOI: 10.1109/ICST.2019.00038.

[29] M. Machalica, A. Samylkin, M. Porth, and S. Chandra. "Predictive Test Selection". In: *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*. IEEE. 2019, pages 91–100. DOI: 10.1109/ICSE-SEIP.2019.00018.

[30] A. G. Malishevsky, J. R. Ruthruff, G. Rothermel, and S. Elbaum. *Cost-cognizant Test Case Prioritization*. Technical report. TR-UNL-CSE-2006-0004, University of Nebraska-Lincoln, 2006.

[31] T. Mattis and R. Hirschfeld. "Lightweight Lexical Test Prioritization for Immediate Feedback". In: *The Art, Science, and Engineering of Programming* 4.3 (2020), 12:1–12:32. DOI: 10.22152/programming-journal.org/2020/4/12.

[32] T. Mattis, P. Rein, F. Dürsch, and R. Hirschfeld. "RTPTorrent: An Open-source Dataset for Evaluating Regression Test Prioritization". In: *Proceedings of the 17th International Conference on Mining Software Repositories (MSR) 2020*. ACM, 2020, pages 385–396. DOI: 10.1145/3379597.3387458.

[33] W. McKinney. "Data Structures for Statistical Computing in Python". In: *Proceedings of the 9th Python in Science Conference*. Edited by S. van der Walt and J. Millman. SciPy, 2010, pages 51–56. DOI: 10.25080/Majora-92bf1922-00a.

[34] D. Meier, T. Mattis, and R. Hirschfeld. "Toward Exploratory Understanding of Software Using Test Suites". In: *Companion Proceedings of the 5th International Conference on the Art, Science, and Engineering of Programming*. ACM, Mar. 2021, pages 60–67. DOI: 10.1145/3464432.3464438.

[35] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco. "Taming Google-scale Continuous Testing". In: *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. Piscataway, NJ, USA: IEEE, 2017, pages 233–242. DOI: 10.1109/ICSE-SEIP.2017.16.

[36] G. Meszaros. *xUnit Patterns: Test Smells*. 2008. URL: http://xunitpatterns.com/Test%20Smells.html (visited on 2019-05-27).

[37] V. Nakoryakov. *Why XOD Switched from Travis to Circle CI*. 2017. URL: https://hackernoon.com/why-xod-switched-from-travis-to-circle-ci-98b6832f2d96 (visited on 2019-05-08).

[38] P. Öhlin. "Prioritizing Tests with Spotify's Test & Build Data using History-based, Modification-based & Machine Learning Approaches". Master's thesis. Linköping University, Software and Systems, 2017.

[39] H. Park, H. Ryu, and J. Baik. "Historical Value-based Approach for Cost-cognizant Test Case Prioritization to Improve the Effectiveness of Regression Testing". In: *Second International Conference on Secure System Integration and Reliability Improvement*. IEEE. 2008, pages 39–46. DOI: 10.1109/SSIRI.2008.52.

[40] Python Software Foundation. *Modules — Python 3.7.3 Documentation*. 2019. URL: https://docs.python.org/3/tutorial/modules.html (visited on 2019-04-01).

[41] Python Software Foundation. *Python 3 Language Reference*. 2019. URL: https://www.python.org/ (visited on 2019-07-30).

[42] G. Rothermel and M. J. Harrold. "A Safe, Efficient Regression Test Selection Technique". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 6.2 (1997), pages 173–210. DOI: 10.1145/248233.248262.

[43] G. Rothermel, M. J. Harrold, and J. Dedhia. "Regression Test Selection for C++ Software". In: *Software Testing, Verification and Reliability* 10.2 (2000), pages 77–109. DOI: 10.1002/1099-1689(200006)10:2%3C77::AID-STVR197%3E3.0.CO;2-E.

[44] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. "Prioritizing Test Cases for Regression Testing". In: *IEEE Transactions on Software Engineering* 27.10 (2001), pages 929–948. DOI: 10.1109/32.962562.

[45] D. Saff and M. D. Ernst. "An Experimental Evaluation of Continuous Testing During Development". In: 29.4 (2004), pages 76–85. DOI: 10.1145/1013886.1007523.

[46] D. Saff and M. D. Ernst. "Reducing Wasted Development Time via Continuous Testing". In: *14th International Symposium on Software Reliability Engineering (ISSRE) 2003*. IEEE. 2003, pages 281–292. DOI: 10.1109/ISSRE.2003.1251050.

[47] S. Saffron. *Tests That Sometimes Fail*. 2019. URL: https://samsaffron.com/archive/2019/05/15/tests-that-sometimes-fail (visited on 2019-05-29).

[48] R. K. Saha, L. Zhang, S. Khurshid, and D. E. Perry. "An Information Retrieval Approach for Regression Test Prioritization based on Program Changes". In: *Proceedings of the 37th International Conference on Software Engineering (ICSE)*. IEEE. 2015, pages 268–279. DOI: 10.1109/ICSE.2015.47.

[49] H. Spieker, A. Gotlieb, D. Marijan, and 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*. ACM. 2017, pages 12–22. DOI: 10.1145/3092703.3092709.

[50] B. Steinert, M. Haupt, R. Krahn, and R. Hirschfeld. "Continuous Selective Testing". In: *International Conference on Agile Software Development*. Springer. 2010, pages 132–146. DOI: 10.1007/978-3-642-13054-0_10.

[51] The Apache Maven Project. *Apache Maven Surefire*. 2019. URL: https://maven.apache.org/surefire/index.html (visited on 2019-03-18).

[52] Travis CI GmbH. *Customizing the Build*. 2019. URL: https://docs.travis-ci.com/user/customizing-the-build (visited on 2019-05-07).

[53] Travis CI GmbH. *Travis CI - Test and Deploy Your Code with Confidence*. 2019. URL: https://travis-ci.org (visited on 2019-03-14).

[54] J. VanderPlas, B. Granger, J. Heer, D. Moritz, K. Wongsuphasawat, A. Satyanarayan, E. Lees, I. Timofeev, B. Welsh, and S. Sievert. "Altair: Interactive Statistical Visualizations for Python". In: *Journal of Open Source Software* 3 (32 2018). DOI: 10.21105/joss.01057.

[55] W. Webber, A. Moffat, and J. Zobel. "A Similarity Measure for Indefinite Rankings". In: *ACM Transactions on Information Systems (TOIS)* 28.4 (2010), page 20. DOI: 10.1145/1852102.1852106.

[56] S. Yoo, R. Nilsson, and M. Harman. "Faster Fault Finding at Google using Multi Objective Regression Test Optimisation". In: *8th European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'11)*. 2011.

Current Technical Reports of the Hasso-Plattner-Institute

Vol.	ISBN	Title	Authors/Editors
144	978-3-86956-526-2	**Die HPI Schul-Cloud – Von der Vision zur digitalen Infrastruktur für deutsche Schulen**	Christoph Meinel, Catrina John, Tobias Wollowski, HPI Schul-Cloud Team
143	978-3-86956-531-6	**Invariant Analysis for Multi-Agent Graph Transformation Systems using k-Induction**	Sven Schneider, Maria Maximova, Holger Giese
142	978-3-86956-524-8	**Quantum computing from a software developers perspective**	Marcel Garus, Rohan Sawahn, Jonas Wanke, Clemens Tiedt, Clara Granzow, Tim Kuffner, Jannis Rosenbaum, Linus Hagemann, Tom Wollnik, Lorenz Woth, Felix Auringer, Tobias Kantusch, Felix Roth, Konrad Hanff, Niklas Schilli, Leonard Seibold, Marc Fabian Lindner, Selina Raschack
141	978-3-86956-521-7	**Tool support for collaborative creation of interactive storytelling media**	Paula Klinke, Silvan Verhoeven, Felix Roth, Linus Hagemann, Tarik Alnawa, Jens Lincke, Patrick Rein, Robert Hirschfeld
140	978-3-86956-517-0	**Probabilistic metric temporal graph logic**	Sven Schneider, Maria Maximova, Holger Giese
139	978-3-86956-514-9	**Deep learning for computer vision in the art domain proceedings of the master seminar on practical introduction to deep learning for computer vision, HPI WS 2021**	Christian Bartz, Ralf Krestel
138	978-3-86956-513-2	**Proceedings of the HPI research school on service-oriented systems engineering 2020 Fall Retreat**	Christoph Meinel, Jürgen Döllner, Mathias Weske, Andreas Polze, Robert Hirschfeld, Felix Naumann, Holger Giese, Patrick Baudisch, Tobias Friedrich, Erwin Böttinger, Christoph Lippert, Christian Dörr, Anja Lehmann, Bernhard Renard, Tilmann Rabl, Falk Uebernickel, Bert Arnrich, Katharina Hölzle

ISBN 978-3-86956-528-6
ISSN 1613-5652