Quality-Aware Learning to Prioritize Test Cases

to obtain the academic degree of a

Doctorate in Natural Sciences

KIT Department of Informatics
Karlsruhe Institute of Technology (KIT)

approved

Dissertation

of

Safa Omri

from Tunis

Date of Oral Examination: 28.01.2022

First Reviewer: Prof. Dr. Carsten Sinz
Karlsruhe Institute of Technology (KIT)

Second Reviewer: Prof. Dr. Ina Schaefer
TU Braunschweig

Quality-Aware Learning to Prioritize Test Cases

Zur Erlangung des akademischen Grades einer

Doktorin der Naturwissenschaften

von der KIT-Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Safa Omri

aus Tunis

Tag der mündlichen Prüfung:	28.01.2022
Erster Gutachter:	Prof. Dr. Carsten Sinz Karlsruhe Institute of Technology (KIT)
Zweiter Gutachter:	Prof. Dr. Ina Schaefer TU Braunschweig

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Karlsruhe, December 3, 2021

. . . .

(**Safa Omri**)

Acknowledgements

First and foremost, I would like to express my deep gratitude to my supervisor Prof. Dr. Carsten Sinz for his support, understanding and encouragement during my Ph.D. Prof. Dr. Carsten Sinz has been supportive and helped me develop myself as a researcher. I am grateful for his scientific advice and knowledge and the many insightful discussions and suggestions that made a huge contribution towards the progress of my Ph.D. project.

I also thank Prof. Dr. Ina Schäfer, who was willing to act as a second referee for my thesis.

Furthermore, my thanks go to Dr. Steffen Görzig from Daimler AG (Research and Development) for providing me with the opportunity to pursue a thesis at an excellent workplace during the first three years of my Ph.D.

Finally, a special thank to my family for their unconditional love!

Zusammenfassung

Softwareanwendungen entwickeln sich aufgrund von kontinuierlichen Funktionserweiterungen, sich ändernden Anforderungen, Code-Optimierungen und Fehlerkorrekturen schnell weiter. Außerdem besteht moderne Software oft aus Komponenten, die von verschiedenen internen oder externen Entwickler-Teams in unterschiedlichen Programmiersprachen entwickelt werden. Während dieser Entwicklung ist es entscheidend, eingeschlichene Fehler kontinuierlich zu erkennen und ständig neue Funktionen zu liefern. Das Testen von Software zielt darauf ab, dieses Risiko zu verringern, indem eine bestimmte Menge von Testfällen regelmäßig - oder wenn der Quellcode geändert wird - ausgeführt wird. Aufgrund der großen Anzahl von Testfällen ist es jedoch nicht möglich, alle Testfälle auszuführen. Automatisierte Techniken zur Priorisierung und Auswahl von Testfällen wurden untersucht, um die Kosten zu senken und die Effizienz der Testaufgaben zu verbessern. Die heutigen Techniken sind jedoch in mehrfacher Aspekten eingeschränkt. Erstens gehen die vorhandenen Techniken zur Priorisierung und Auswahl von Testfällen oft davon aus, dass die Fehler gleichmässig über die Softwarekomponenten verteilt sind, was dazu führen kann, dass der größte Teil des Testbudgets für Komponenten ausgegeben wird, bei denen die Wahrscheinlichkeit eines Fehlers geringer ist als bei denen, die mit hoher Wahrscheinlichkeit Fehler enthalten. Zweitens haben die vorhandenen Techniken ein Skalierbarkeitsproblem, nicht nur in Bezug auf die Größe der ausgewählten Testsuite, sondern auch in Bezug auf die Umlaufzeit zwischen den Code-Commits und der Rückmeldung des Entwicklers über fehlgeschlagene Testfälle im Kontext von Continuous Integration (CI). Schliesslich ist es schwierig, das Wissen der menschlichen Tester algorithmisch zu erfassen, das für die Test- und Release-Zyklen entscheidend ist.

Diese Arbeit ist eine neue Herangehensweise an das alte Problem, das die Kosten für das Software-Testing in dieser Hinsicht reduziert, indem sie einen datengesteuerten, leichtgewichtigen Ansatz für die Auswahl und Priorisierung von Testfällen vorstellt, der (i) während der CI-Zyklen für schnelles und Ressourcen optimalen Rückmeldung an die Entwicklern und (ii) während der Release-Planung durch Erfassung des Fachwissens der Tester und der Release-Anforderungen verwendet wird. Unser Ansatz kombiniert Software-Qualitätsmetriken mit Code-Churn-Metriken, um ein regressives Modell zu erstellen, das die Fehlerdichte jeder Komponente vorhersagt, sowie ein Klassifizierungsmodell zur Unterscheidung zwischen fehlerhaften und nicht fehlerhaften Komponenten. Beide Modelle werden verwendet, um den Testaufwand auf die Komponenten zu richten, die wahrscheinlich die größte Anzahl von Fehlern enthalten. Die Vorhersagemodellen wurden an acht industriellen Automobilsoftwareanwendungen bei Daimler validiert und zeigten eine Klassifizierungsgenauigkeit von 89% und eine Genauigkeit von $85,7\%$ für das Regressionsmodell. In dieser Arbeit wird ein Modell zur Priorisierung von Testfällen entwickelt, das auf den Merkmalen der Codeänderung, der Ausführungshistorie der Tests und der Entwicklungshistorie der Komponenten basiert. Das Modell reduziert die Kosten der CI, indem es vorhersagt, ob eine bestimmte Codeänderung die einzelnen Testsuiten und ihre entsprechenden Testfälle auslösen sollte. Um das Fachwissen und die Präferenzen der Tester algorithmisch zu erfassen, hat unser Ansatz ein Testfallauswahlmodell entwickelt, das die Präferenzen der Tester in der Form eines probabilistischen Graphen aufnimmt und das Problem der optimalen Testbudgetzuweisung sowohl online im Kontext von CI-Zyklen als auch offline bei der Planung eines

Releases löst. Schließlich wird ein theoretisches Kostenmodell vorgestellt, welches beschreibt, wann unser Priorisierungs- und Auswahlansatz sinnvoll ist. Der Gesamtansatz wird an zwei industriellen analytischen Anwendungen im Bereich des Energiemanagements und der prädiktiven Instandhaltung validiert. Dabei zeigt sich, dass über 95% der Testfehler an die Entwickler zurückgemeldet werden, während nur 43% der insgesamt verfügbaren Testfälle ausgeführt werden.

Abstract

Software applications evolve at a rapid rate because of continuous functionality extensions, changes in requirements, optimization of code, and fixes of faults. Moreover, modern software is often composed of components engineered with different programming languages by different internal or external teams. During this evolution, it is crucial to continuously detect unintentionally injected faults and continuously release new features. Software testing aims at reducing this risk by running a certain suite of test cases regularly or at each change of the source code. However, the large number of test cases makes it infeasible to run all test cases. Automated test case prioritization and selection techniques have been studied in order to reduce the cost and improve the efficiency of testing tasks. However, the current state-of-art techniques remain limited in some aspects. First, the existing test prioritization and selection techniques often assume that faults are equally distributed across the software components, which can lead to spending most of the testing budget on components less likely to fail rather than the ones highly to contain faults. Second, the existing techniques share a scalability problem not only in terms of the size of the selected test suite but also in terms of the round-trip time between code commits and engineer feedback on test cases failures in the context of Continuous Integration (CI) development environments. Finally, it is hard to algorithmically capture the domain knowledge of the human testers which is crucial in testing and release cycles.

This thesis is a new take on the old problem of reducing the cost of software testing in these regards by presenting a data-driven lightweight approach for test case prioritization and execution scheduling that is being used (i) during CI cycles for quick and resource-optimal feedback to engineers, and (ii) during release planning by capturing the testers domain knowledge and release requirements. Our approach combines software quality metrics with code churn metrics to build a regressive model that predicts the fault density of each component and a classification model to discriminate faulty from non-faulty components. Both models are used to guide the testing effort to the components likely to contain the largest number of faults. The predictive models have been validated on eight industrial automotive software applications at Daimler, showing a classification accuracy of 89% and an accuracy of 85.7% for the regression model. The thesis develops a test cases prioritization model based on features of the code change, the tests execution history and the component development history. The model reduces the cost of CI by predicting whether a particular code change should trigger the individual test suites and their corresponding test cases. In order to algorithmically capture the domain knowledge and the preferences of the tester, our approach developed a test case execution scheduling model that consumes the testers preferences in the form of a probabilistic graph and solves the optimal test budget allocation problem both online in the context of CI cycles and offline when planning a release. Finally, the thesis presents a theoretical cost model that describes when our prioritization and scheduling approach is worthwhile. The overall approach is validated on two industrial analytical applications in the area of energy management and predictive maintenance, showing that over 95% of the test failures are still reported back to the engineers while only 43% of the total available test cases are being executed.

Contents

1. Introduction

This is an introductory chapter. Section 1.1 presents the motivation for the problem handled in this thesis, followed by an illustrative scenario in Section 1.2 and the description of the problem statement in Section 1.3. The contributions of this thesis are discussed in Section 1.4. Finally, Section 1.5 presents the outline of this thesis.

1.1 Motivation

Software applications evolve rapidly because of continuous functionality extensions, changes in requirements, code optimization, and fixes of faults. Moreover, modern software is often composed of components engineered with different programming languages by different internal or external teams. During this evolution, it is crucial to detect unintentionally injected faults while continuously releasing new features.

Therefore many industries use a Continuous Integration (CI) strategy, a common agile technique in which engineers frequently commit their latest code changes to the mainline code base, enabling them to quickly and cost-effectively verify that their code could indeed pass tests in multiple system environments. Developers commonly need to know that unmodified code has not been affected after such code changes, and regression testing is a crucial technique to ensure that these changes do not introduce new bugs at each CI cycle.

The most straightforward strategy for regression testing is to re-execute all existing test cases. This method is easy to implement, but it could be unnecessarily expensive because first, changes occur frequently and tests should be run on every code change, and second, changes often affect only a small part of the software project. Moreover, regression testing could consume 80% of the overall testing budget and require weeks to run all test suites [Kan97]. For example, Google reported high testing costs, running on the order of 150 million tests per day [BX16]. Facebook reported that they make around 60K code changes per day, and they run on the order of 10K tests per change [MSPC19]. These facts introduce new challenges for the testing activities due to (i) the dynamic environment resulting from frequent code changes and (ii) time constraints since testing should be fast enough to enable frequent builds and test the source code.

In order to reduce the cost of testing, several techniques, such as the test case minimization, selection and prioritization techniques, have been developed [EMR01, LHH07, MHZ$^+$12, MB16, MB17, RUCH99, SZKP15, TAS06, YH12]. Test case minimization aims to eliminate the redundancy in the test suites or remove obsolete test cases to reduce the size and thus the cost. Test case selection aims at identifying the test cases that execute the changed areas of the source code. However, selecting the test cases most likely to detect faults for early execution is hard. Test case prioritization is popularly used as an optimization mechanism for ranking tests by their likelihood of revealing failures without eliminating any test case from the test suites.

However, existing prioritization techniques are usually time and resource-intensive to be applied within CI cycles. Moreover, existing techniques do not allow to capture algorithmically the testers' domain knowledge and requirements which are essential to speed up and control the expensive testing and release process.

This thesis is a new take on the old problem of reducing the cost of software testing in these regards by presenting a data-driven lightweight approach for test case prioritization that is being used (i) during CI cycles for quick and resource-optimal feedback to engineers, and (ii) during release planning by capturing the testers domain knowledge and release requirements.

1.2 An Illustrative Scenario

Assume you are a tester responsible for a software application. This software application is composed of a set of components. The software components are developed in different programming languages by different internal or external teams. It is a mature software product with multiple previous versions that are already launched on the market. Consider that you have a large pool of test cases used to test the previous versions of the software. The test cases for each component are executed on specific testing machines which are pre-configured based on the components' programming language. A new version of the software is due to be released soon with some new features. The software application faces new changes in the source code by adding these new features and test cases might be added or modified. As a tester, one of your tasks is to ensure that the newly developed features do not affect the functionality of the previous versions of the software application. Unfortunately, you have a limited testing budget, and the existing set of test cases is too large to be processed thoroughly. As software components evolve, the number of test cases grows, making it practically impossible to execute all test cases. So, you decide to execute a test case prioritization technique to reduce, as possible, the round-trip time between code commits and the feedback about failing test cases and to enable quick fix and integration of the newly added code within the mainline source code base. Meanwhile, you recognize that the order generated by the test case prioritization technique is not adequate, as you know that certain test cases belonging to components that are highly to be executed or highly to contain faults should be considered first. However, the test case prioritization technique prioritizes these test cases at the end of the order. Consequently, given a time limit, it can lead to a sub-optimal allocation of test cases; meaning that test cases could be executed in some of the components, neglecting other components that are more critical (e.g., highly to contain faults, highly to be executed). Furthermore, you have a very crucial business agreement, and a particular feature or functionality is expected to be more used than other features; thus, it must be tested first. Unfortunately, it is not possible to incorporate the new requirements and your domain knowledge into the existing test case prioritization techniques. Such information is essential to speedup and control the expensive testing and release process. Therefore, you are facing the following challenges:

- How to find the most effective and efficient subset of test cases to execute on the available test execution machines while satisfying your preferences within a given budget constraint? (the third contribution is described in Chapter 6, where we solve the test case scheduling challenge across all software components)
- Which test cases should you run first in each component? (the second contribution is described in Chapter 5, where a rank is given for each test case in each software component) and,

- In which code-part should you focus the testing and run the test cases? (the first contribution, as described in Chapter 4, is to direct the testing in areas in the software that are most likely to be failed)

1.3 Problem Statement

Continuous Integration (CI) is a cost-effective and common software development practice to allow developers frequently integrate their work. Regression testing is a crucial technique to ensure that the changes do not introduce new bugs at each CI. Testing in CI involves test case prioritization and execution at each cycle with the goal to accelerate detecting faults and the release of new features. Automated test case prioritization and selection techniques have been studied in order to reduce the cost and improve the efficiency of testing tasks. However, the current state-of-art techniques remain limited in some aspects which defines the different problem statements of this thesis:

1. the existing test prioritization techniques often assume that faults are equally distributed across the software components, which can lead to spending most of the testing budget on components less likely to fail rather than the ones highly to contain faults,
2. in CI environments, traditional test prioritization techniques can be difficult to apply. Most traditional techniques require gathering code coverage data or performing static analysis which makes such techniques (i) time intensive and cannot be executed within CI cycles and (ii) limited from a practicality point of view as modern software is often written in different programming languages.
3. existing techniques do not allow to capture algorithmically the testers' domain knowledge and requirements which are essential to speedup and control the expensive testing and release process.

In summary, the existing test prioritization techniques are time and resource-intensive to be executed within CI cycles in an online and dynamic fashion, and do not allow to capture algorithmically the testers' domain knowledge and preferences. In addition, they ignore the modular architecture of software applications and the relationships between their components. In fact, a seasoned tester might have knowledge about which components are more error-prone. Moreover, the testing activities might sometimes require prioritizing some components over others because of release requirements (e.g., a particular feature or functionality is expected to be more used than other features).

In order to tame these problems, we propose a new lightweight test case prioritization in CI environments. Our approach combines software quality metrics with code churn metrics to build a regressive model that predicts the fault density of each component and a classification model to discriminate faulty from non-faulty components. Both models are used to guide the testing effort to the components likely to contain the largest number of faults. Moreover, our approach minimizes the test prioritization overhead. It continuously adapts to the changing environment as new code and new test cases are added in each CI cycle through formulating the test prioritization problem as a computational efficient online learn-to-rank model using reinforcement learning techniques. In addition, our approach developed a test case execution scheduling model that consumes the testers' preferences in the form of a probabilistic graph, allowing to capture the tester's domain knowledge and the preferences algorithmically and solves the optimal test budget allocation problem both online in the context of CI cycles and offline when planning a release.

In more details, Figure 1.1 illustrates the whole approach where a software application is composed of a set of components. The software components are developed in different languages (e.g Python for the data engineering components, JavaScript for the User Interfaces, Java for backend

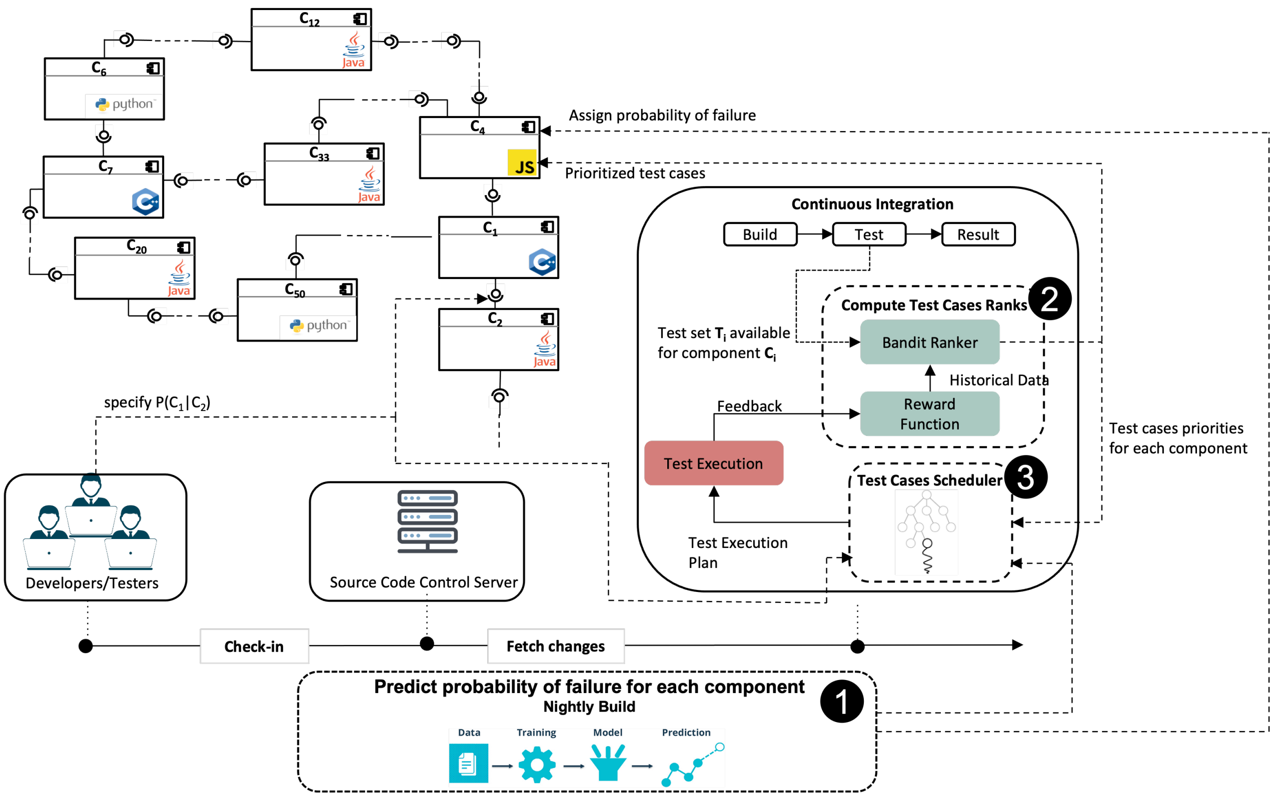


Figure 1.1: Test Case Scheduler across Software Components based on Ranked Test Cases and with Human in the Loop

workflows, $C++$ for some compressing algorithms, etc). For each software component, a set of test cases has been created, and test cases might be added or modified after each change of the application's source code. For each component a prioritized test cases is computed (2), as described in Chapter 5. The test cases for each component are executed on specific testing machines which are pre-configured based on the components programming language and setup requirements. A probability of failure is assigned for each software component based on a nightly build where quality metrics are computed (1), as described in Chapter 4. Moreover, the software tester can add probabilities of execution on the edges connecting the components. Such probabilities can be extracted (i) after profiling how the application's clients are using the application, or (ii) based on preferences and domain expertise of the testers (3), as described in Chapter 6. The overall approach describes more thorough the three following main steps:

(1) **Predict probability of failure for each component:** mining software quality and code churn metrics to predict the fault density of each component during a nightly build to guide testing to the software component likely to fail. In an exploratory study, described in Chapter 4 in Section 4.4, we show that the combination of code complexity metrics together with static analysis and with churn metrics results allows accurate prediction of fault density and build classifiers discriminating faulty from non-faulty software components. The experiments were carried out using eight software projects of an automotive head unit control system (Audio, Navigation, Phone, etc.). Each project, in turn, is composed of a set of components. The total number of components is 54. These components have a collective size of $\sim 24MLOC$ (million LOCs without comments and spaces). All components use the object oriented language $C++$. In experiments to separate fault-prone from non-fault-prone components, the developed approach achieved a classification accuracy of 89%, and the regressor predicted the fault density of software components with an accuracy of 85.7%

(2) **Compute test cases ranks:** a lightweight test case prioritization technique in CI based on foundational results of learning-to-rank from the field of information retrieval and reinforcement learning is developed. The prioritization of the test cases is executed in each software component. In particular, the developed approach used historical test results to learn a ranking model used to predict rankings for all test cases in each software component. The model is based on reinforcement learning principles allowing us to design an adaptive method capable of learning from the execution environment. Adaptiveness means in the context of this thesis, that the developed approach can progressively improve its efficiency after each test case's execution cycle. Unlike other test prioritization approaches, the developed technique is able to adapt to situations where test cases are added or deleted, or when testing priorities change because of changing failure indications in different code regions as the code matures or as the requirements change. Moreover, our technique does not require computationally intensive operations during the ranking process. It uses knowledge about the execution history of the test cases at each CI cycle and updates this knowledge from feedback provided by a reward function. The developed approach is validated, as described in Chapter 5, Section 5.3, on eight open source projects and on four industrial case studies shows that over 95% of the test failures are still reported back to the software engineers while only 40% of the total available test cases are being executed,

(3) **Test cases scheduler:** formulation of the test case prioritization problem across all software components as a sequential decision-making process using reinforcement learning to capture the testers' requirements. The developed technique uses graph neural networks to represent the states of the reinforcement learning problem and capture the structure of the software application as well the domain knowledge and requirements of the testers. The goal of the developed technique is to schedule the test cases of the software components for execution on the available test execution machines while satisfying the tester's preferences and the time constraint for each CI cycle. The developed approach is validated on two industrial case studies, showing that over 95% of the test failures are still reported back to the software engineers while only 43% of the total available test cases are being executed, as in Chapter 6, Section 6.6.

Finally, the thesis presents a theoretical cost model that describes when our prioritization and scheduling approach is worthwhile. The overall approach is validated on two industrial analytical applications in the areas of energy and asset management, showing over 95% of the test failures are still reported back to the engineers, while only 43% of the total available test cases are being executed.

1.4 Contributions

The contributions of this thesis are as follows:

1. this thesis guides the testing to the software components likely to fail by proposing a regressive model that predicts the fault density of each component based on quality metrics and code churn metrics,

2. a lightweight test cases prioritization technique based on foundational results of learning-to-rank from the field of information retrieval and reinforcement learning is developed. The prioritization of the test cases is executed in each software component,

3. in order to capture the testers requirements, this thesis formulates the test prioritization problem across all software components as a sequential decision-making process using reinforcement learning. The developed technique uses graph neural networks to represent the states of the reinforcement learning problem and capture the testers' domain knowledge and requirements.

4. this thesis also presents a theoretical cost model that describes when the developed prioritization and scheduling approach is worthwhile.

1.5 Outline

According to the stated main goals, the remainder of this thesis is structured as outlined in the following:

- **Chapter 2** introduces the foundations which are considered as necessary and important to understand this thesis.
- **Chapter 3** presents the related work and their relationship to this work.
- **Chapter 4** describes how mining software quality metrics and code churn metrics helps to predict components failures, as mentioned in the first contribution in Section 1.4.
- **Chapter 5** formulates the test case prioritization as an online learning-to-rank problem within each software component based on reinforcement learning principles, as described in the second contribution in Section 1.4.
- **Chapter 6** formulates the test prioritization problem as a sequential decision-making process using reinforcement learning, as mentioned in the third contribution in Section 1.4.
- **Chapter 7** presents the validation of the overall approach combining the contributions (described in Section 1.4) on two industrial applications.
- **Chapter 8** concludes the thesis by summarizing the contributions, and outlines planned future work.

2. Background

This section presents basic definition and principles related to our approach. This background section is organized as follows: Section 2.1 provides definitions of both software quality metrics, test metrics and their different techniques. The code quality metrics presented in Section 2.1.1 are used in our first contribution in Chapter 4 to define the inputs and outputs metrics of the regressive model and the classification models. Section 2.1.2 provides notations and definitions of the different regression testing techniques. It introduces necessary notations and presents the addressed problem in a formal way. It also includes a formalization of the test prioritization problem addressed in our work. Section 2.2 presents the statistical and mathematical techniques used in all three contributions which are discussed in Chapter 4, 5 and 6. It also describe the main elements of reinforcement learning and introduces basic concepts such as artificial neural network, agent, policy and reward functions. Finally, in order to compare the performance of the different methods addressed in our approach, Section 2.3 presents the evaluation metrics used to validate our approach for both software fault prediction in Chapter 4 and test case prioritization in Chapter 5 and in Chapter 7.

2.1 Software Test Cases Management Techniques

The test cases management process depends on both software quality metrics, test metrics and their different techniques:

2.1.1 Code Quality Metrics

2.1.1.1 Failures and Faults

In this work, we use the term fault to refer to an error in the source code. We refer to an observable error at program run-time as failure. We assume that, every failure can be traced back to a fault, but a fault does not necessarily result in a failure.

Faults which have been identified before a software release, typically during software testing, are referred to as pre-release faults. If faults are identified after a software release as a result of failures in the field (by the customer), then such faults are referred to as post-release faults. The focus of this work is on pre-release faults to obtain an early estimate of software component's fault-proneness in order to guide software quality assurance towards inspecting and testing the components most likely to contain faults.

Fault-proneness is defined as the probability of the presence of faults in the software [DMP02]. Such probability is estimated based on previously detected faults using techniques such as software testing. The research on fault-proneness has focused on (i) the definition of code complexity and testing thoroughness metrics, and (ii) the definition and experimentation of models relating metrics with fault-proneness.

2.1.1.2 Code Complexity Metrics

Software complexity metrics were initially suggested by Chidamber and Kemerer [CK94]. Basili et al. [BBM96], and Briand et al. [BWIL99] were among the first to use such metrics to validate and evaluate fault-proneness. Subramanyam and Krishnan [SK03], and Tang et al. [TKC99] showed that these metrics can be used as early indicators of external software quality. In this work, the studied code metrics (e.g., relevant Lines of Code(LOC), complexity, nesting, statements, paths, parameters) are presented in Table 4.2 in Chapter 4.

Nagappan et al. [NBZ06] empirically confirmed that code complexity metrics can predict post-release faults. Based on a study on five large software systems, they showed that (i) for each software, there exists a set of complexity metrics that correlates with post-release faults, (ii) there is no single set of metrics that fits all software projects, (iii) predictors obtained from complexity metrics are good estimates of post-release defects, and (iv) such predictors are accurate only when obtained from the same or similar software projects. Our work builds on the study of Nagappan et al. [NBZ06], and focuses on pre-release faults while taking into consideration not only the code complexity metrics but also the faults detected by static analysis tools to build accurate pre-release fault predictors.

2.1.1.3 Static Analysis

In this work, we used the faults detected by static analysis tools to predict the pre-release fault density. Our basic hypothesis is that while static analysis tools only find a subset of the actual faults in the program's code, it is highly likely that these detected faults, combined with code complexity metrics would be a good indicator of the overall code quality. This is explained by the fact that static analysis tools can find faults that occur on paths uncovered by testing. On the other hand, testing has the ability to discover deep functional and design faults, which can be hardly discovered by static analysis tools. In other words, code complexity metrics would complement the static analysis fault detection capabilities to account for the type of faults that cannot be detected by static analysis tools, and hence such a combination can form accurate predictors of pre-release faults.

Nagappan et al. [NWH$^+$04] showed at Nortel Networks on an 800 KLOC commercial software system, that automatic inspection faults detected by static analysis tools were a statistically significant indicator of field failures and is effective to classify fault-prone components. Nagappan et al. [NB05a] applied static analysis at Microsoft on a 22 MLOC commercial system and showed that the faults found by static analysis tools were a statistically significant predictor of pre-release faults and can be used to discriminate between fault-prone and non fault-prone components. Again, our approach does not only make use of the faults detected by static analysis, but also uses code complexity metrics; it goes beyond the works of Nagappan et al. by not only using the faults detected by static analysis tools as an indicator of pre-release faults, but also combines these faults with code complexity metrics in a mathematical model which delivers a more accurate predictor and classifier of pre-release faults. It is important to note that the focus of our work as well as the related works [NB05a, NWH$^+$04] is on the application of non-verifying static analysis tools. The study of the impact of using verifying static analysis tools on the prediction accuracy of pre-release fault densities goes beyond the scope of this work and is planned as a future work.

2.1.1.4 Code Churn Metrics

The amount of code change that occurs within a software unit over time is measured as code churn. It can be easily extracted from a system's change history, which is automatically recorded by a version control system. Most version control systems employ a file comparison utility (such as diff) to estimate how many lines were added, deleted, or changed by a programmer in order to create a new version of a file from an old version. These distinctions serve as the foundation for churn measures. Different recent works have used past changes as indicators for faults because the more changes are done to a part of the source code, the more likely it will contain faults [KZWJZ07, KWZ08]. Thus, we mine the git repositories databases to extract several code churn metrics (e.g., added LOC, removed LOC, etc., see Table 4.2) used in our first contribution to predict software fault density and described in Chapter 4. For example, Total Lines of Code (LOC) is the number of lines of non-commented executable lines in the files, Churned LOC is the sum of the added and changed lines of code between a baseline version and a new version of the files, Deleted/Added LOC is the number of lines of code deleted/added between the baseline version and the new version, Files churned is the number of files that churned, etc.

2.1.2 Regression Testing Techniques

This section introduces the basic concepts and definitions of regression testing and minimization, selection and prioritization techniques.

2.1.2.1 Test Suite Minimization

Test suite minimization techniques aim to reduce the size of a test suite by removing redundant test cases. Rothermel et al. [GR02] define the minimization problem as a "test suite reduction". More formally, following Rothermel et al. [GR02], the test suite minimization is defined as follows:

Definition 1. *Test Suite Minimization Problem.*
Let T be a test suite. A test suite is a collection of test cases organized in a logical order (i.e., es case for registration will precede the test case for login) to test a software application or its specific functionality. Given a test suite T , a set of test requirements $\{r_1, ..., r_n\}$ that must be satisfied to provide the "adequate" testing of the program, and subsets of T, $\{T, T_1, ..., T_n\}$, one associated with each of the r_is such that any one of the test cases $t_j \in T_i$ can be used to test r_i . The goal is to find a representative set T' of test cases from T that satisfies all r_is.

The testing criterion is fulfilled when every test requirement in $\{r_1, ..., r_n\}$ is satisfied. The r_is can represent various test case requirements, such as source code statements, decisions or specification items [GR02].

2.1.2.2 Test Case Selection

While most test case selection techniques aim to reduce the size of a test suite, the majority of them are modification-aware. The test cases are chosen because they are relevant to the changed parts of the system, which usually requires a white-box static analysis of the program code. More formally, following Rothermel and Harrold [RH98], the selection problem is defined as follows:

Definition 2. *Test Case Selection Problem.*
Given a program P, a modified version P' and a test suite T . The goal is to reduce the size of the test suite by finding T' a subset of T, with which to test the modified version P'.

2.1.2.3 Test Case Prioritization

Test case prioritization refers to the ordering of test cases in order to maximize some specific properties, such as the rate of fault detection, as soon as possible. It attempts to find the best permutation of the test case sequence. It does not involve test case selection and assumes that all test cases may be executed in the order of the permutation it generates, but that testing may be terminated at any time during the testing process. The prioritization problem is more formally defined as follows:

Definition 3. *Test Case Prioritization Problem.*
Given a test suite T, PT the set of permutations of T, and a function from PT to real numbers, $f : PT \longrightarrow \mathbb{R}$. The goal is to find $T' \in PT$ such that $(\forall T'' \in PT)$ $(T'' \neq T')[f(T') \geq f(T'')]$.

2.2 Statistical Modeling and Mathematical Optimization Techniques

2.2.1 Statistical Techniques

A number of statistical techniques have been used to analyze software quality. Khoshgoftaar et al. [KML93], and Larus et al. [MK90] used multiple linear regression analyses to model the software quality as a function of software metrics. The coefficient of determination, R^2, is usually used to quantify how much variability in the software quality can be explained by a regression model. A major difficulty when using regression models to combine several metrics is the issue of multicollinearity among the metrics, which is explained by the existence of inter-correlations between the metrics. Multicollinearity can lead to an overestimation of the regression estimate. For example, high cyclomatic complexity usually correlates with a high amount of code lines [NBZ06].

One approach to overcome multicollinearity is applying Principal Component Analysis (PCA) [Jac03, JC16b] on the metrics before applying regression modeling. Using PCA, a subset of uncorrelated linear combinations of metrics, which account for the maximum possible variance, is selected for use in regression models. Denaro et al. [DMP02] used PCA on a study that considered 38 software metrics for the open source projects Apache 1.3 and 2.0 to select a subset of nine principal components which explained 95% of the total data variance. Nagappan et al. [NBZ06] used PCA to select a subset of five principal components out of 18 complexity metrics that account for 96% of the total variance in one of the studied commercial projects.

2.2.2 Reinforcement Learning

Reinforcement learning is defined as learning through interaction with the environment and receiving feedback based on the taken actions. It is the nature of learning when we think about how the learning process works [SB18]. Reinforcement learning, in other words, is the process of making decisions based on prior experience [SB18]. Interacting with the environment aids in the accumulation of experience. A decision is made, an action is taken, and feedback is received at each state. The received feedback can be either a reward or a punishment. When the same or a very similar state is observed later, the previous experience can be used to make a more informed decision, maximizing reward and minimizing punishment. The learner must investigate and determine what actions can be taken in light of the current situation. A fundamental part of reinforcement learning is mapping the current situation into action in order to maximize the reward [SB18].

We define in the following the fundamental elements of reinforcement learning. The agent interacts with the environment, as shown in Figure 2.1. The interaction between the agent and the environment occurs in discrete steps, one at a time. The agent makes an action in exchange for a set of new states and a reward. The agent-environment interaction naturally divides into sub-sequences, which we refer to it as episodes (such as plays of a game, trips through a maze, or any sort of repeated interaction) [SB18]. Each episode concludes in a special state known as the terminal state, which is then reset to a starting state or a sample from a distribution of starting states.

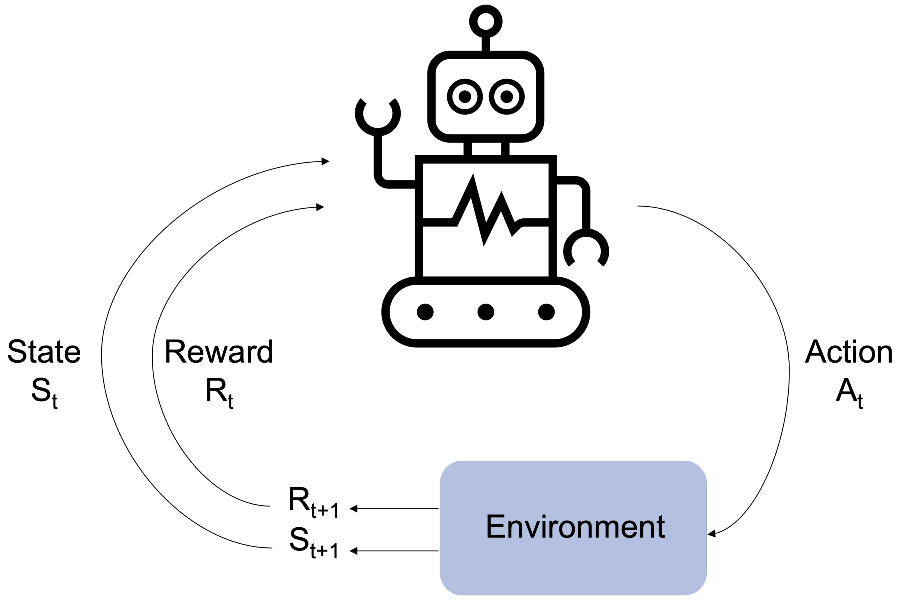


Figure 2.1: The agent-environment interaction in reinforcement learning; adapted from [SB18]

The objective of reinforcement learning is through learning to obtain a behavior policy: $\pi : S \rightarrow A$ where π is the policy. The agent selects one action to get the largest rewards under this policy. The long-term effects of agent behavior must be considered when studying practical problems [SB18]. As a result, the state value function or state-action pair value function is used to define an objective function:

$$V^{\pi}(s_t) = \sum_{i=0}^{\infty} \gamma^i r_{t+i} \tag{2.1}$$

where r_{t+i} is the expected reward for the action a_t in each step from state s_t to s_{t+1} and γ is the discount factor. Based on Eq. 2.1, we can derive the best policy of a behavior:

$$\pi^* = arg \max_{\pi} V^{\pi}(s) \tag{2.2}$$

A single agent can be learned using Q-learning, a simple and well-understood algorithm [WD92], described as follows.

For each state s and action a, Q-value $Q(s, a)$ is defined as follows:

$$Q(s,a) = r(s,a) + \gamma \sum_{s'} p(s,a,s') V^*(s') \tag{2.3}$$

where p is the transition probability: when action a is executed, a transition is made from its current state s to a succeeding state s'. Moreover, $V^*(s') = \max_{a'} Q^*(s', a')$. We can get the following equation:

$$Q(s,a) = (r(s,a) + \gamma \sum_{s'} p(s,a,s') \times \max_{a' \in A(s')} Q^*(s', a')) \tag{2.4}$$

where $A(s')$ is a set of possible actions in the state of s'. From the Q-value above, we get a greedy strategy according to:

$$\pi^*(s) = arg \max_{a \in A(s)} Q(s,a) \tag{2.5}$$

However, as the transition function is unknown in reinforcement learning, Eq. 2.4 cannot be used to calculate the value function Q. Thus, an approximation method is adopted to estimate the value function. Specially, the Monte Carlo method and dynamic programming technology can be integrated to solve the above problem.

According to the predicted deviations [SB18] and a Q-value $Q(s,a)$ that provides an estimation of the value of performing an individual action or a joint action, the learner updates its estimation $Q(s,a)$ as follows:

$$Q(s,a) = (1-\alpha) \times Q(s,a) + \alpha \times (r + \gamma \times \max Q(s',a')) \tag{2.6}$$

where r is consistent with $r(s,a)$, a is the action chosen by the learner at state s and r is the reward resulted from a and γ is the discount factor. $\alpha (0 < \alpha < 1)$ is the learning rate, which decides to what extent the reward and future estimation replacing the current estimation. s' is the resulting state of action a, and a' is the action performed at state s'. If α decreases slowly during learning and all actions are sampled infinitely, which are labeled as GLIE (greedy in the limit, infinite exploration), Q-learning will converge to true Q-values for all actions in the single agent scenario [SJLS00].

2.2.2.1 Exploration versus Exploitation

The Exploration/Exploitation dilemma is often brought up in the Multi-armed bandit problem in the existing literature [Yog12]. The training process of reinforcement learning requires exploration of the action space to collect data and to update the reward. The iteration and collection of data help to learn about the environment, and gain experience about actions and their expected rewards, which leads to choose the next actions based on the developer experience from the past which means exploiting his knowledge. However, when it comes to a highly dynamic environment (e.g., stock trading), the knowledge of the environment is not complete. Therefore, it is crucial to keep exploring during the training process for as long as possible [SB18]. To combat the Exploration/Exploitation dilemma several methods can be utilized. A few of the most common methods may include [SB18]:

- ϵ-greedy and its variations,
- Upper Confidence Bound,
- Gradient Bandit Algorithms,
- Thompson Sampling.

Only ϵ-greedy and its variations are utilized in the learning process. ϵ-greedy is a simple approach to choose between exploration and exploitation. In each iteration of the learning process at time t, the algorithm takes a random action with the probability of ϵ with $0 \leq \epsilon \leq 1$. This means, taking a random action a from the action space, the probability of exploration is $P(a) = \epsilon$ and the probability of exploitation is $1 - \epsilon$. Two extensions of ϵ-greedy are Decaying Epsilon, and Optimistic Initial Values [SB18].

Decaying Epsilon is a variation of ϵ-greedy where ϵ is not constant. Instead, the value decreases as the developer proceed with the learning. The only difference here is that in each iteration at time t, the developer take a random action a with probability of $f(t)$ which is a decaying function. For instance, a popular choice is $f(t) = \frac{1}{t}$ [SB18].

Optimistic Initial Values only contains the greedy part but not the epsilon part of the regular ϵ-greedy. The mean rewards of each action in this method are optimistically initialized to encourage exploration. In each step the action with the highest mean reward is selected, i.e., $A_t = \arg\max Q_t(a)$. The reward is then calculated and updates the mean reward. The means will eventually converge to the true values of the rewards [SB18].

2.2.2.2 Reward, Returns and Episodes

A ***reward*** is the signal that the environment returns to the agent after taking an action. The goal of the agent is to maximize the expected cumulative value of the rewards in the long run [Cum15]. The cumulative sum of rewards is called ***Return***, defined as G_t, at time t [SB18]:

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + ... + R_T \tag{2.7}$$

where R_t is reward at time t and T is the final time step, also referred to as the terminal *state.*

An ***episode*** is an agent's interaction [AMS09] or a sequence of actions [Cum15] with the environment that always starts from the initial state and ends in the terminal state.

2.2.2.3 Markov Decision Process

Markov Decision Problems (MDPs) can be traced back all the way to the fifties to the work of Richard Bellman in stochastic control theory [Bel57]. They are sequential decision-making processes where an action has to be chosen in each state by the system [SB18].

Definition 4. *Markov Decision Process (MDP).*
A MDP is a 4-tuple $M =< S, A(.), P, R >$, where

- S: a finite set of states. When an agent arrives at a state, the agent can observe the complete state.
- $A(s)$: a finite set of actions. The set of available actions depends on the current state $s \in S$.
- P: when an action $a \in A$ is performed, the environment makes a probabilistic transition from its current state s to a resulting state s' according to a probability distribution $P(s'|s, a)$.
- R: similarly, when action a is performed and the environment makes its transition from s to s' , the agent receives a real-valued (possibly stochastic) reward r, whose expected value is $r = R(s'|s, a)$.

In an MDP, the state and the reward at step $t+1$ only depend on the state at step t and the action taken then and the previously observed states, taken actions and received rewards have no effect on what the state and the resulting reward would be in the following step. An MDP on a finite action space and a finite state space should satisfy the following two definitions [Cum15]:

Transition Probabilities: the transition probabilities ($\mathcal{P}^a_{ss'}$) of a finite MDP give the probability of transitioning from the initial state s_t with action a_t to the subsequent state s_{t+1} [Cum15]:

$$\mathcal{P}^a_{ss'} = \mathbb{P}(s_{t+1} = s_t \mid s_t = s, a_t = a) \tag{2.8}$$

Expected Reward. Expected reward for state s_t, action a_t, subsequent state s_{t+1} is the expectation of the reward function given the triplet of (a_t, s_t, s_{t+1}) [Cum15]:

$$\mathcal{R}^a_{ss'} = E(r_{t+1} \mid s_{t+1} = s', s_t = s, a_t = a) \tag{2.9}$$

2.2.2.4 Dynamic Programming

Dynamic Programming (DP) refers to a series of algorithms and methods to derive optimal policy for an environment using MDP [SB18]. DP assumes it can gain knowledge of the entire environment and can compute an exact, deterministic solution. This assumption implies that the environment, its state, action, and reward sets are finite [SB18].

The Bellman equation [Bel57] is used to recursively update the value functions of each state by starting at the terminal state. Values of states are calculated based on the expectation of the reward and the estimation of the subsequent state's value. When the value function is close to the optimal one, the policy can be improved by making it greedy with respect to the value function [Cum15]. Then the state (or state-action) value functions are re-evaluated until the policy converges closer to the optimal policy.

A major drawback of a DP is that the entire environment must be known and be finite, so it is not performant when it comes to tackling environments with large or continuous action space.

2.2.2.5 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is an efficient search algorithm for sequential decision making problems with a win or loss outcome; it has been used in many Chess AIs [DS17] and Go AIs [SD17]. Therefore in case, MCTS keeps a state tree in which each node represents a distinct set of actions and the edges represent individual actions. In addition, each node contains a value that summarizes how beneficial that state is in relation to the objective. Once exploring the search space, MCTS uses this value in conjunction with prior search path information to achieve a good balance of exploiting known information and exploring unknown states. The search tree is expanded by MCTS in four steps: *Selection*, *Expansion*, *Simulation* and *Backpropagation*, shown in Figure 2.2. When making a decision, MCTS will empirically build a tree of states and values by repeating these four steps based on the amount of budget available. The decision will then be made by selecting the action that will result in the child with the highest value.

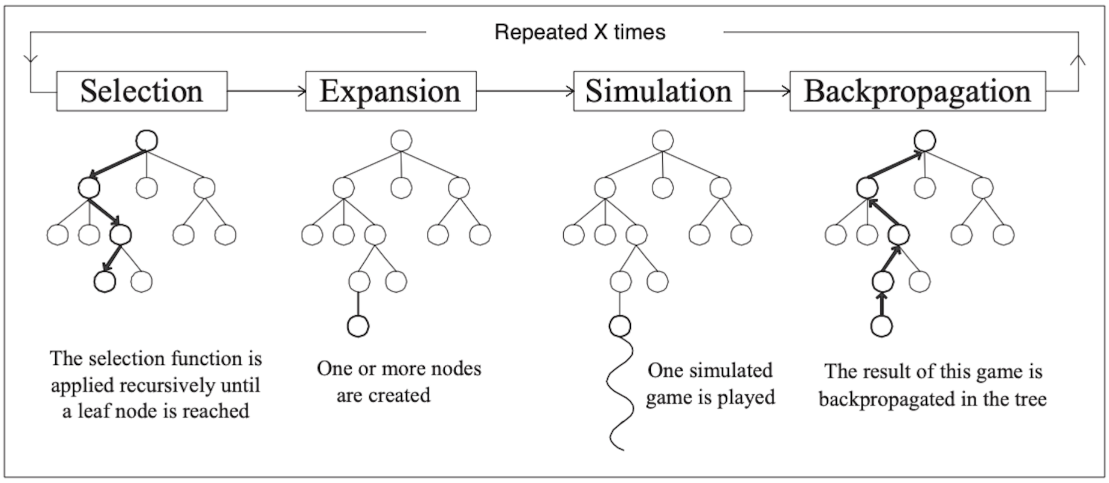

Repeated X times
Selection
Expansion
Simulation
Backpropagation
The selection function is applied recursively until a leaf node is reached
One or more nodes are created
One simulated game is played
The result of this game is backpropagated in the tree

Figure 2.2: The Outline of Monte Carlo Tree Search (adapted from [SB18])

Selection:

MCTS will select a path of interest (sequence of moves) that will be searched further in this step, as shown in Figure 2.2. This step is critical for MCTS because it aims to select the most promising subtree for further investigation. The selection process tries to strike a good balance between exploration and exploitation as it moves down the tree. *Exploration* entails selecting nodes that have not been extensively explored with the goal of discovering new search paths that may be more effective. *Exploitation* entails selecting high-value nodes in order to obtain more information about promising moves. In general, the *Upper Confidence Bound(UCB)* equation is used to select nodes when traversing down the tree [HTL19].

$$\frac{w_i}{n_i} + c\sqrt{\frac{\ln n}{n_i}} \tag{2.10}$$

where n_i is the number of visits the i-th node has received and w_i is the number of wins throughout all of simulations during the visits. Furthermore, n represents the total number of parent node visits, and c represents an exploration factor that balances exploration and exploitation [HTL19]. The first term in the equation is the exploitation score, which describes the win rate from computed simulations. The second term is the exploration score, which decreases as the number of visits increases; this encourages exploring nodes that have not been explored as much.

Expansion:

After selecting the desired path and stopping at a node, the next step is to create a new child node by performing an action from the current state [HTL19]. After that, the new node is appended to its parent and added to the search tree. The action to take is randomly selected in a traditional MCTS approach.

Simulation:

After expansion, a new node is added to the search tree with no information about its value, making it impossible to determine how desirable the new state is. As a result, in this step, MCTS will perform a quick rollout, simulating the game to the end and obtaining the outcome. During the rollout process, actions are chosen at random using a traditional MCTS approach [HTL19].

Backpropagation:

The simulation step's results are then backpropagated up the search tree. The values of all ancestor nodes will be updated based on the results of the simulation. To summarize, when choosing an action in a game, MCTS will iterate through these four steps based on the budget available, building a search tree with information on how favorable each state is[HTL19]. MCTS will then select the action with the highest $\frac{w_i}{n_i}$ value as the next move.

2.2.2.6 On-Policy and Off-Policy Methods

On-policy and off-policy methods are used to ensure that an agent visits all of the states in an environment on a regular basis and continues to select them. An on-policy method attempts to update the policy based on the data that resulted in the most recent action, whereas an off-policy method updates the policy based on data obtained at any point [Cum15].

On-policy methods are in general soft with respect to policy improvement. For example, when they follow an ϵ-greedy policy, the probability of an action being chosen at random is ϵ. However, an on-policy method continually moves towards a more deterministic optimal policy [SB18].

Off-policy methods are typically more difficult to implement than on-policy counterparts because they require additional concepts, models, and data. Furthermore, they are slower to converge and may have more variance. Off-policy methods have a target policy (the one being learned) and a behavior policy (the policy that generates behavior). The optimal policy is determined by the target policy, and the behavior policy investigates the environment to generate the behavior [SB18]. As a result, they are more general and, in most cases, more effective. When both the behavior and the target policies are the same, on-policy methods become a subset of off-policy methods.

2.3 Evaluation Metrics

This section presents the evaluation metrics used to validate our approach for both software fault prediction and test case prioritization.

2.3.1 Evaluation Metrics for Prediction and Classification Analysis

The process of software fault prediction is shown in Figure 2.3, the first step is to collect source code repositories from software archives. The second step is to extract features from the source code repositories and the commits contained therein. There are many traditional features defined in past studies, which can be categorized into two kinds: code metrics (e.g., McCabe features and CK features) and process metrics (e.g., change histories). The extracted features represent the train and test dataset. To select the best-fit defect prediction model, the most commonly used method is called k-fold cross-validation that splits the training data into k groups to validate the model on one group while training the model on the $k-1$ other groups, all of this k times. The error is then averaged over the k runs and is named cross-validation error.

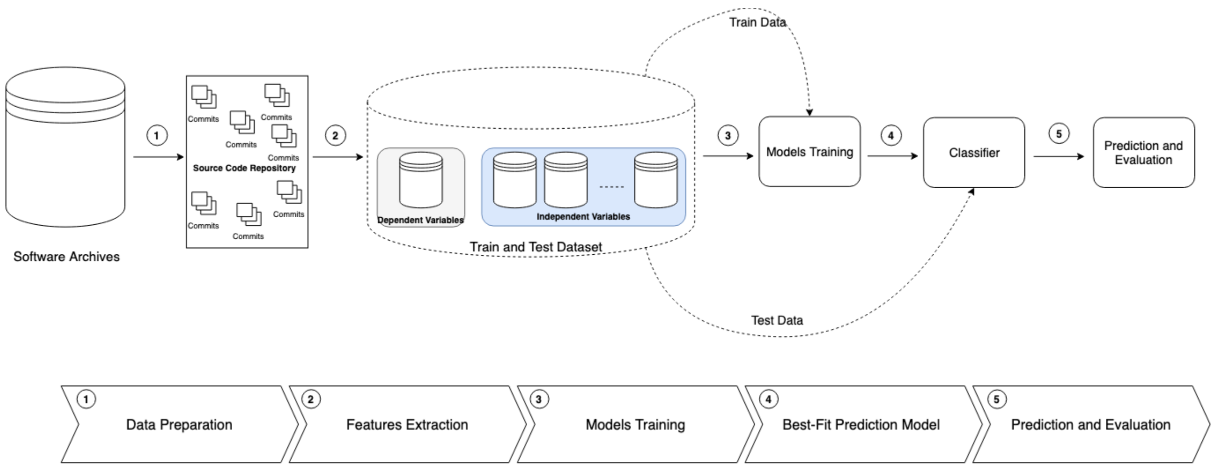
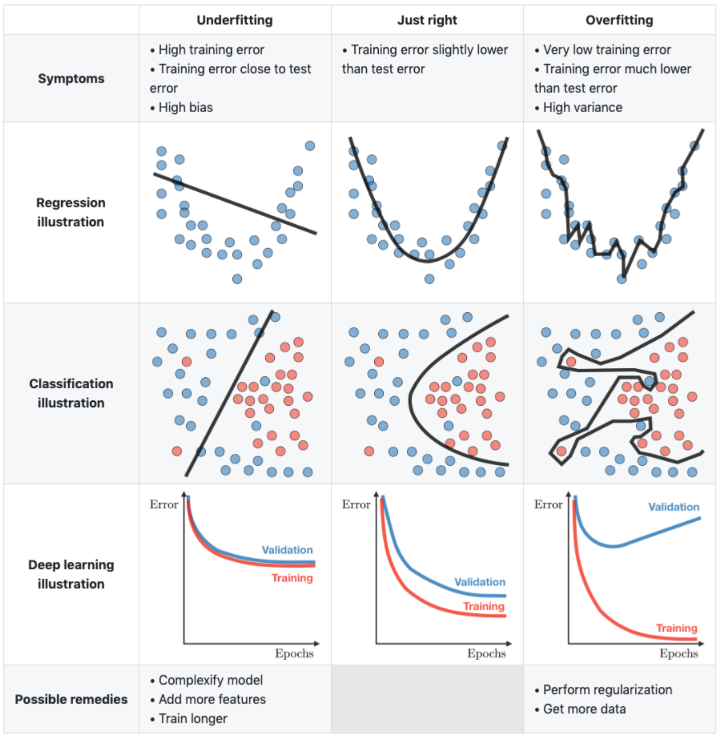


Figure 2.3: Software Defect Prediction Process

The diagnostics of the model is based on these features: *(1) Bias*: A model's bias is the difference between the expected prediction and the correct model that we attempt to predict for given data points. *(2) Variance*: A model's variance is the variability of the model's prediction for given data points. *(3) Bias/variance tradeoff*: the simpler the model, the higher the bias, and the more complex the model, the higher the variance [Ami18]. Figure 2.4 shows a brief summary of how underfitting, overfitting and a suitable fit looks like for the three commonly used techniques regression, classification and deep learning. After selecting a model, it is trained on the entire dataset and tested on the test dataset. Most approaches to defect prediction treat defect prediction as a binary classification problem. After the models have been fitted, the test data is fed into the trained classifier (the best-fit prediction model), which can predict whether the files are buggy or clean. Afterwards, in order to assess the performance of the selected model, quality metrics are computed. To have a more complete picture when assessing the performance of a model, a confusion matrix is used. It is defined as shown in Figure 2.5. We summarize the metrics for the performance of classification models in Table 2.1.

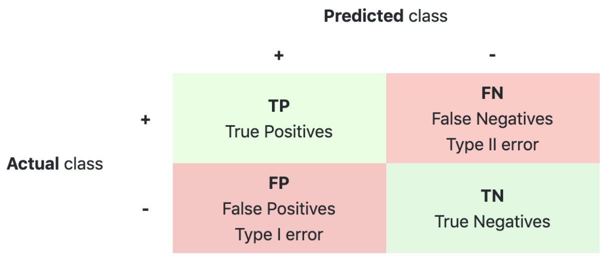
Regression Model Evaluation Methods

After developing a number of different regression models, we have a lot of criteria to evaluate and compare them against each other:



	Underfitting	Just right	Overfitting
Symptoms	• High training error • Training error close to test error • High bias	• Training error slightly lower than test error	• Very low training error • Training error much lower than test error • High variance
Regression illustration			
Classification illustration			
Deep learning illustration			
Possible remedies	• Complexify model • Add more features • Train longer		• Perform regularization • Get more data

Figure 2.4: Fitting Model Diagnostics [Ami18]



		Predicted class	
		+	-
Actual class	+	**TP** True Positives	**FN** False Negatives Type II error
	-	**FP** False Positives Type I error	**TN** True Negatives

Figure 2.5: Confusion Matrix [Ami18]

Table 2.1: Common metrics used to assess the performance of classification models

Metric	Formula	Interpretation
Accuracy	$\frac{TP+TN}{TP+TN+FP+FN}$	Overall performance of model
Precision	$\frac{TP}{TP+FP}$	How accurate the positive predictions are
Recall	$\frac{TP}{TP+FN}$	Coverage of actual positive sample
F1 score	$\frac{2TP}{2TP+FP+FN}$	Hybrid metric useful for unbalanced classes

Definition 5. ***Root Mean Square Error:*** The root mean square error (RMSE) is a popular formula for calculating the error rate of a regression model. However, we can only compare models whose errors are measurable in the same units [Ami18]

$$RMSE = \sqrt{\frac{\sum_{i=1}^{n} (\hat{y}_i - y_i)^2}{n}} \tag{2.11}$$

where $\hat{y}_i$ is the prediction and y_i is the true value.

Definition 6. ***Mean Absolute Error:*** The Mean Absolute Error is the average of the difference between the Actual and Predicted Values. It tells us how far the predictions were off from the actual output. They do not, however, give us any indication of the direction of the error, i.e. whether we are underestimating or overestimating the data [Ami18]. Mathematically, it is represented as:

$$MAE = \sqrt{\frac{\sum_{i=1}^{n} |\hat{y}_i - y_i|}{n}} \tag{2.12}$$

where $\hat{y}_i$ is the prediction and y_i is the true value.

Definition 7. ***Coefficient of Determination:*** The coefficient of determination (R^2) summarizes the explanatory power of the regression model and is computed from the sums-of-squares terms [Ami18].

A data set has n values marked $y_1, ..., y_n$ (collectively known as y_i), each associated with a fitted (or modeled, or predicted) value $f_1, ..., f_n$ (known as f_i, or sometimes $\hat{y}_i$, as a vector f).

Define the residuals as $e_i = y_i - f_i$ (forming a vector e).

If $\bar{y}$ is the mean of the observed data:

$$\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i \tag{2.13}$$

then the variability of the data set can be measured with two sums of squares formulas [Ami18]:

The total sum of squares (proportional to the variance of the data):

$$SS_{\text{tot}} = \sum_i (y_i - \bar{y})^2 \tag{2.14}$$

The sum of squares of residuals, also called the residual sum of squares:

$$SS_{\text{res}} = \sum_i (y_i - f_i)^2 = \sum_i e_i^2 \tag{2.15}$$

The most general definition of the coefficient of determination is

$$R^2 = 1 - \frac{SS_{\text{res}}}{SS_{\text{tot}}} \tag{2.16}$$

In the best case, the modeled values exactly match the observed values, which results in $SS_{\text{res}} = 0$ and $R^2 = 1$. A baseline model, which always predicts $\bar{y}$, will have $R^2 = 0$. Models that have worse predictions than this baseline will have a negative R^2.

Classification Model Evaluation Methods

Accuracy is most likely the most well-known metric for evaluating machine learning models. It multiplies the number of correctly predicted samples by the sample size. In terms of classification models, model accuracy can be defined as the ratio of correctly classified samples to total number of samples [Ami18]. For binary classification models, the accuracy can be defined as described in Table 2.1, where

- True Positive (TP): A true positive is an outcome where the model correctly predicts the positive class.
- True Negative (TN): A true negative is an outcome where the model correctly predicts the negative class.
- False Positive (FP): A false positive is an outcome where the model incorrectly predicts the positive class.
- False Negative (FN): A false negative is an outcome where the model incorrectly predicts the negative class.

2.3.2 Specific Evaluation Metrics for Test Case Prioritization

Our approach learns priority ranks for all test cases. Test case selection techniques, however, might omit test cases that might have exposed faults if they were executed. Our approach, when applied in the CI context, might terminate before all test cases are executed. This behavior is similar to the behavior of test case selection techniques. We used the following standard commonly used metrics in software test selection to assess the effectiveness of our approach when not all test cases are executed [PP21, BX16, MSPC19]:

Test recall:

Intuitively, the test recall approximates the empirical probability of a particular test selection strategy catching an individual failure.

$$testRecall = \frac{\sum\limits_{c \in \mathcal{C}} \mid \mathcal{T}_c \cap TF_c \mid}{\sum\limits_{c \in \mathcal{C}} \mid TF_c \mid} \tag{2.17}$$

where $\mathcal{C} = \{c_1, c_2, ..., c_M\}$ a set of CI cycles, and such that for the failed tests TF_c, $\exists_{c \in \mathcal{C}} TF_c \neq 0$, and T_c are the selected test cases from a test suite $\mathcal{TS}$.

For the evaluation of the effectiveness of our test case prioritization approach, we use the standard evaluation metric:

Average Percentage of Faults Detected (APFD):

APFD was introduced in [RUCH99] to measure the effectiveness of test case prioritization techniques. It measures the quality via the ranks of failure-detecting test cases in the test execution order. It ranges from 0 to 1, with higher numbers implying faster fault detection. Let $\mathcal{TS}$ be a test suite of n test cases, the $APFD$ of a prioritized test suite $\mathcal{TS}'$ is calculated using the following formula:

$$APFD = 1 - \frac{\sum\limits_{t \in TF_i} rank(t)}{nm} + \frac{1}{2n} \tag{2.18}$$

where m refers to the total number of faults, n refers to the total number of test cases, $rank(t)$ returns the rank of the first test case that reveals the ith fault in test suite $\mathcal{TS}'$.

Furthermore, we used an extension of the APFD metric:

Normalized Average Percentage of Faults Detected (NAPFD):

NAPFD is the ratio between detected and detectable failures within the test suite $\mathcal{TS}$. The $NAPFD$ of a prioritized test suite $\mathcal{TS}'$ is calculated using the following formula:

$$NAPFD = p - \frac{\sum\limits_{t \in TF_i} rank(t)}{nm} + \frac{p}{2n} \tag{2.19}$$

where m refers to the total number of faults, n refers to the total number of test cases, TF_i is the set of failed test cases, $rank(t)$ returns the rank of the first test case that reveals the ith fault in test suite $\mathcal{TS}'$, and p is the number of faults detected by t divided by m.

3. Related Work

This chapter highlights existing approaches and addresses the relationship to our work.

Test Case Prioritization in Continuous Integration:

In this section, we provide an overview of the existing research in relation to our whole approach, which is test case prioritization in the context of continuous integration. A detailed description and analysis of the related works of each contribution is explored in Chapter 4 Section 4.6 for the first contribution, in Chapter 5 Section 5.4 for the second contribution, and in Chapter 6 Section 6.7 for the third contribution.

We summarize in Table 3.1 the studies related to our work and categorize them related to the considered information to prioritize test cases and to the evaluation metric considered to validate their works.

Research on test case prioritization in continuous integration environment is very active today. Various techniques have been proposed to identify test cases that exercise change code during CI cycles [GEM15, KSM$^+$15] using machine learning techniques. Pan et al. [PXN13] showed that techniques such as k-means based on coverage information can be used with good results. Mirarab et al. [MAT12] proposed a prioritization approach based on Integer Linear Programming (ILP) and greedy method and using coverage metrics. Zhang et al. [Zha19] proposed time-aware prioritization using ILP. Similarly, Walcott et al. [WSKR06] presented a genetic algorithm for time-aware regression test suite prioritization for frequent code rebuilding. Strandberg et al. [SSA$^+$16] apply a novel prioritization method with multiple factors in real-world embedded software and show the improvement over industry practice. Busjaeger and Xie [BX16] used Support Vector Machine (SVM) map [YFRJ07] that ranks by training a model based on training data labeled as 'relevant' or 'non-relevant'. They used coverage data, test file path similarity and test content similarity, failure history, and test age. These ranking-based approaches used machine learning techniques, assuming that before training, the full data is available, and thus incremental learning is not supported (i.e., integrating new data into already constructed models); however, new models are constructed from scratch [PBGB21]. Especially in CI environments, this can lead to potentially outdated models, which are very inefficient and time-consuming. There are various approaches for test case prioritization; however, most of them are challenging to apply in practice and especially in a CI context due to the complexity and computational overhead typically required to collect and analyze different parameters such as test coverage, mapping between code changes and test executions, etc. In fact, it is frequent to remove test cases from some CI cycles as they test obsolete features, or to add new test cases to test new features. Moreover, some test cases might be important at some point of time because they test features important for the customers and later loose their prevalence because of a testing focus shift. To summarize, non-adaptive techniques might miss changes in the importance of test cases over others because they are based on systematic prioritization approaches (i.e., they do not consider the test case volatility as described in Table 3.3).

In comparison, our approach is a lightweight method that uses only historical execution results from previous CI cycles. In addition, our approach adaptively learns new ranks for the test cases in each software component as code changes and test suites evolve. The prioritization of the test cases is executed in each software component by applying reinforcement learning to make the ranking approach adaptive and suitable to our CI context. Our approach minimizes the testing overhead and continuously adapts to the changing environment as new code and new test cases are added in each CI cycle. Adaptiveness means in our context that our approach can progressively improve its effectiveness after each test case's execution cycle. Unlike other test prioritization approaches, our technique is able to adapt to situations where test cases are added or deleted, or when testing priorities change because of changing failure indications in different code regions as the code matures or as the requirements change. Moreover, our approach developed a test case scheduling model that consumes the testers' preferences in form of a probabilistic graph and solves the optimal test budget allocation problem both online in the context of CI cycles and offline when planning a release across all software components as a sequential decision-making process using reinforcement learning.

The two works closest to ours are [SGMM17] and [PLV20], as described in Table 3.3. Spieker et al. [SGMM17] were among the first that used RL in the context of test case prioritization and selection in continuous integration. However, the approach presented by Spieker et al. [SGMM17] is implicitly building a classifier which would put all likely to fail test cases in the same priority and the same for the non-faulty ones. So if, for example, out of $20,000$ test cases, 5000 are classified as likely to fail, then all 5000 test cases would have similar priority leaving the system to select randomly out of them. Nevertheless, our approach continuously learns a rank for each test case, where a rank corresponds to the failure likelihood of a test for a given CI cycle. This allows executing test cases by descending priority until a CI time-limit is reached or all test cases are executed. Lima et al. [PLV20] used a Multi-Armed Bandit (MAB) approach for test case prioritization and showed that they outperform Spieker et al.'s approach [SGMM17]. Lima et al. [PLV20] used the test case rewards in a sliding window based on the approach's idea of Elbaum et al. [ERP14] where they use sliding time window to select test suites to be applied during presubmit testing by tracking their history, then prioritizes test suites based on such window to be performed subsequent post-submit testing.

However, all these recent works consider the software application as one single component when executing test cases, as described in Table 3.3 (i.e. they do not consider the diversification challenge, which is test case scheduling across all software components). Unfortunately, though, modern applications are composed of different components, which lead to sub-optimal allocation of test cases (i.e. test cases could be executed only in some of the components, neglecting other components that could be more important for testing). Furthermore, the software components of one application have different probability of failure and different probability of execution that are added from the testers either after profiling how the application's client are using the application or based on their preferences and their domain knowledge. Thus, recent works do not consider the testers' domain expertise, as described in Table 3.3.

Our approach rank the test cases in each software component from the highest likely to fail to the lowest ones and then schedule the test cases across all software components for execution on the available test execution machines while satisfying the tester's preferences and the time constraint for each CI cycle.

Table 3.1: Summary of Studies in the Context of Test Case Prioritization in Continuous Integration (CI)

	Considered Information		Evaluation Metric					
	Coverage-based	**History-based**	APFD	APFDc	NAPFD	FD	Time	Stmt
Jiang et al. [JZTC09]	✓$^{(3),(4)}$	–	✓	–	–	–	–	✓
Jiang et al. [JC10]	✓$^{(3),(4)}$	–	–	–	–	–	–	✓
Jiang et al. [JCT11]	✓$^{(3),(4)}$	–	–	–	–	–	–	✓
Yoo et al. [YNH11]	✓$^{(5)}$	✓$^{(1),(2)}$	–	–	–	–	✓	–
Jiang et al. [JZC$^+$12]	✓$^{(3),(4)}$	–	–	–	–	–	–	✓
Marijan et al. [MGS13]	–	✓$^{(1),(2)}$	–	✓	–	✓	✓	–
Elbaum et al. [ERP14]	–	✓$^{(1),(2)}$	–	–	–	–	✓	–
Marijan et al. [Mar15]	✓$^{(5)}$	✓$^{(1),(2)}$	–	✓	–	–	–	–
Knauss et al. [KSM$^+$15]	–	✓$^{(1)}$	–	–	–	–	–	–
Marijan [ML16]	–	✓$^{(1),(2)}$	✓	–	–	✓	–	–
Busjaeger et al. [BX16]	✓	✓$^{(1)}$	✓	–	–	–	–	–
Cho et al. [CKL16]	–	✓$^{(1)}$	✓	–	–	–	–	–
Strandberg et al. [SSA$^+$16]	–	✓$^{(1),(2)}$	–	–	–	✓	✓	–
Jiang et al. [JC16a]	✓$^{(3),(4)}$	–	–	–	–	–	–	✓
Spieker et al. [SGMM17]	–	✓$^{(1),(2)}$	–	–	✓	–	–	–
Kim et al. [KJL17]	–	✓$^{(1)}$	✓	–	–	–	–	–
Marijan et al. [MLG$^+$17]	✓$^{(5)}$	✓$^{(1),(2)}$	–	–	–	✓	–	–
Marijan et al. [ML17]	✓$^{(5)}$	✓$^{(1),(2)}$	–	–	–	✓	✓	–
Strandberg et al. [ESAO$^+$17]	–	✓$^{(1),(2)}$	–	–	–	✓	✓	–
Kwon et al. [KK17]	–	✓$^{(1),(2)}$	–	–	–	–	✓	–
Xiao et al. [XMZ18]	–	✓$^{(1),(2)}$	–	–	✓	✓	–	–
Liang et al. [LER18]	–	✓$^{(1),(2)}$	–	✓	–	–	–	–
Zhu et al. [ZSR18]	–	✓$^{(1)}$	–	–	–	✓	–	–
Haghighatkhah et al. [HMOK18]	–	✓$^{(1)}$	✓	–	–	–	✓	–
Chen et al. [CLZ$^+$18]	✓$^{(5)}$	✓$^{(2)}$	✓	✓	–	✓	✓	–
Alegroth et al. [AKR18]	–	✓$^{(1),(2)}$	–	–	–	–	✓	–
Marijan et al. [MLS18]	–	✓$^{(1),(2)}$	–	–	–	–	✓	–
Wen et al. [WYY18]	–	✓$^{(1),(2)}$	–	–	✓	–	–	–
Marijan et al. [MGL19]	✓	✓$^{(1)}$	–	–	–	✓	✓	–
Najafi et al. [NSR19]	–	✓$^{(1),(2)}$	–	–	–	–	✓	–
Pradhan [PWA$^+$19]	–	✓$^{(1),(2)}$	–	✓	–	–	✓	–
Yu et al. [YFM$^+$19]	–	✓$^{(1)}$	–	✓	–	–	✓	–
Lima and Vergilio [PLV20]	–	✓$^{(1)}$	–	✓	–	–	✓	–

Remark.

Studies in the category **History-based** consider:
(1) Failure History: information about which test cases failed previously, and
(2) Execution History: the time to execute the test cases.

Studies in the category **Coverage-based** consider the total number of:
(3) Statements covered (or not covered)
(4) Functions covered (or not covered)
(5) Test coverage

Table 3.2 represents the most common evaluation metrics considered in the previous Table 3.1. The Definition of each metric and their formula are described in the Background Chapter 2 under Section 2.3.2.

Table 3.2: Evaluation Metrics for the Test Case Prioritization

Evaluation Metrics	
APFD	Average Percentage of Faults Detected
APFDc	Average Percentage of Faults Detected per Cost
NAPFD	Normalized Average Percentage of Faults Detected
FD	Percentage of Faults Detected
Time	Time spent to execute the Prioritized Test Cases
Stmt	Minimum Percentage of Statements that must be considered to locate the Fault

Table 3.3: The Studied Challenges versus Recent Works

	Challenges		
	Tester's Domain Knowledge & Requirements[1]	**Test Volatility**[2]	**Diversification**[3]
Jiang et al. [JZTC09]	✗	✗	✗
Jiang et al. [JC10]	✗	✗	✗
Jiang et al. [JCT11]	✗	✗	✗
Yoo et al. [YNH11]	✗	✗	✗
Jiang et al. [JZC+12]	✗	✗	✗
Marijan et al. [MGS13]	✗	✗	✗
Elbaum et al. [ERP14]	✗	✓	✗
Marijan et al. [Mar15]	✗	✗	✗
Knauss et al. [KSM+15]	✗	✗	✗
Marijan [ML16]	✗	✗	✗
Busjaeger et al. [BX16]	✗	✗	✗
Cho et al. [CKL16]	✗	✗	✗
Strandberg et al. [SSA+16]	✗	✗	✗
Jiang et al. [JC16a]	✗	✗	✗
Spieker et al. [SGMM17]	✗	✓	✗
Kim et al. [KJL17]	✗	✗	✗
Marijan et al. [MLG+17]	✗	✗	✗
Marijan et al. [ML17]	✗	✗	✗
Strandberg et al. [ESAO+17]	✗	✗	✗
Kwon et al. [KK17]	✗	✗	✗
Xiao et al. [XMZ18]	✗	✗	✗
Liang et al. [LER18]	✗	✗	✗
Zhu et al. [ZSR18]	✗	✗	✗
Haghighatkhah et al. [HMOK18]	✗	✗	✗
Chen et al. [CLZ+18]	✗	✗	✗
Alegroth et al. [AKR18]	✗	✗	✗
Marijan et al. [MLS18]	✗	✗	✗
Wen et al. [WYY18]	✗	✗	✗
Marijan et al. [MGL19]	✗	✗	✗
Najafi et al. [NSR19]	✗	✗	✗
Pradhan [PWA+19]	✗	✗	✗
Yu et al. [YFM+19]	✗	✗	✗
Lima and Vergilio [PLV20]	✗	✓	✗

Remark.

[1] Tester's Domain Knowledge and Requirements
[2] Test Volatility
[3] Diversification: test case scheduling across all software components

In the following sections, we will discuss in details the numerous studies related to our work that have examines the realm of test case prioritization and selection.

Test Case Prioritization and Selection in Regression Testing:

Test case prioritization is defined and explained more in details in the Background Chapter 2. Existing works can be divided into two categories: heuristic-based and machine learning-based test prioritization.

Heuristic-Based TCP: This category group has mainly focused on information such as history, code coverage, models, and requirements. Pan et al.[PBGB21] conducts a systematic literature review and analyze 29 primary studies about test case selection and prioritization researches.

Code coverage information was widely used in several studies to prioritize test cases. Thus, techniques that are based on structural code coverage are able to increase the chances of increasing fault detection early [KP02]. Several approaches were introduced for prioritizing test cases using structural coverage measures. Rothermel et al. [RUCH99] presented empirical results assessing the efficacy of these approaches. Among the structural coverage measures are statement coverage [RUCH99], methods coverage [RUCH01], and modified condition coverage [JH03]. The code coverage information can be gathered using either static analysis or dynamic analysis. Static analysis techniques, as discussed by Jeffrey et al. [JG06], are inaccurate and overestimate the coverage data. Furthermore, Jeffrey et al. [JG06] presented that static analysis with reflection support can enhance the accuracy of coverage information significantly, however they are inefficient because of their high computation cost. Dynamic analysis techniques are also challenging and several studies discussed the reasons. We can summarized in three main reasons: (1) Maintainability: both Lima et al. [PLV20] and Memon et al. [MGN+17] presented that, in most actively developed projects, high code changes rates quickly make code coverage data obsolete, usually requires frequent updates [PBGB21]. (2) Computation Overhead: several works, such as Elbaum et al.[ERP14], Lima et al. [PLV20] and Memon et al. [MGN+17] showed that when dealing with a large code-base, code analysis and instrumentation can take a long time to complete. Memon et al. [MGN+17] confirmed that having to run a code instrumentation tool on Google's code-base at each milestone and collecting code coverage data, might impose excessively operational costs to be feasible. (3) Applicability: the non machine leaning based techniques leads to more efforts and customization, because they are often dependent on the language and on the platform [PBGB21]. Moreover, the extraction of code coverage requires traceability between code and test cases, such information is not easily accessible with system tests (i.e., black-box testing) [LMVAa20], [PBGB21].

System models were, on the other hand, also used from other researchers [KTH05] to select and generate test cases for the system's modified parts. Models are representations of the actual system. When compared to the execution of the actual system, Korel et al. [KK09] showed such abstractions help in making model execution for the entire test suite low cost and fast. Since the source code continuously evolves as developers make changes (e.g., adding new features, fixing bugs, refactoring existing code, etc.), updating the models to reflect the changes is necessary. When using model-based approaches, such updates create overhead [PBGB21]. Furthermore, because models are frequently extracted using source code analysis, they acquire the drawbacks of code-based approaches. Korel et al. [KTH05] described a *model-based prioritization approach* in which test cases are prioritized using the original and modified system models, as well as information about the system model and its behavior. Whereas code-based coverage approaches are more accurate compared to model-based approaches, they lead to practical challenges in terms of complexity and computational effort in terms of coverage.

Srikanth et al. [SWO05] introduced a value-driven approach for prioritizing system-level test cases based on *software requirement specifications*. They prioritized the test cases based on four factors after mapping test cases to software requirements. The four factors consist of fault-proneness of the requirements, customer priority, implementation complexity, and requirement volatility. In another work of Srikanth et al. [SW05], they also proposed a system-level technique for prioritization based on requirements according to the same factors. Their works are based on the idea that coverage-based white-box prioritization techniques can be time-consuming and are harder to apply on complex systems. Arafeen and Do [AD13] prioritized test cases based on the importance of the requirements to the clients. They clustered the requirements based on mining the texts and then utilizes the specification of the requirements test cases to cluster test cases. The test cases are prioritized in each cluster based on code metrics (e.g. McCabe Cyclomatic Complexity) and then based on the requirements.

History-based approaches is used to prioritize tests based on data from previous test runs. They are based on the assumption that previous test case failures are a good predictor of test cases with a high likelihood of failure in new releases. Kim and Porter [KP02] introduced a history-based method for ranking tests based on the average of previous execution results. According to Noor and Hemmati [NH15], a class of quality metrics that estimate test case quality based on their similarity to previously failing test cases from previous releases is defined. Park et al. [PRB08] formulated a history-based approach for analyzing the impact of test costs. Approaches based on history are less cost-effective than coverage-based and model-based approaches. Learning optimal test case prioritization policies solely based on test execution history, on the other hand, appears difficult, especially for complex software systems. They may also be unsuitable for constantly changing testing environments with frequent changes in code and test suites [PBGB21].

Fault-proneness and bug history-based approaches were proposed from several studies for test case prioritization. Anderson et al. [ASD14] and Engstroem et al. [ERL11] were among the studies that used the failure history information in software repositories to improve the effectiveness of testing process. Laali et al. [LLH$^+$16] studied a new online test case prioritization approach. Their method uses the location of previously identified faults in the source code to prioritize test cases. Marijan et al. [MGS13] proposed a case study of a test prioritization approach for continuous regression testing, called ROCKET. ROCKET prioritizes the set of test cases based on historical failure data, test execution time, and domain-specific heuristics. Kim et al. [106] [KB10] incorporates a fault localization technique to prioritize test cases by giving lower priority to test cases covering previous faults because they expect that faults are fixed after being detected, and thus those test cases will have lower fault detection probability. Wang et al. [WNT17] proposed a quality-aware test case prioritization method (QTEP) and addressed the limitation of existing coverage-based methods. They used static bug finders and unsupervised methods to identify fault-prone code units and then adapt existing coverage-based algorithms by considering the weight of fault-prone source code. Similar to Wang et al. work [WNT17], Paterson et al. [PCA$^+$19] proposed a test case prioritization strategy based on defect prediction. Another similar work to Wang et al. [WNT17] and Paterson et al. [PCA$^+$19], Mahdieh et al. [MMHM21] used the bug history to estimate the fault-proneness of code units, and prioritize test cases based on test case diversification and fault-proneness estimations.

These approaches have one major drawback, namely that they are not adaptive to rapidly changing environments, particularly when it comes to continuous integration environments.

Machine Learning-Based Test Case Prioritization: Applications of machine learning (ML) techniques to prioritize test cases are explored in this section. Many researchers apply ML techniques to test case prioritization in order to integrate data from various sources into predictive models [DDB+19]. It has been shown that ML approaches can significantly improve test selection and prioritization [DDB+19].

The literature on the selected papers is divided into three categories based on the used machine learning methods: Clustering, Ranking-based Models, and Natural Language Processing Models.

Clustering-based Test Case Prioritization: Clustering is an unsupervised machine learning technique. Clustering tends to group similar data points into groups (clusters). These clusters are then relatively easy to understand and to manipulate. A variety of clustering algorithms is currently available but their primary difference revolves around the number of configuration parameters and methods they require to calculate the distance between data instances (e.g. Euclidean Distance, Hamming Distance) [PBGB21]. In the context of test case prioritization, clustering is used to identify similar test cases based on their characteristics (e.g. coverage, fault detection, etc.). It guides then the prioritization of test cases.

Yoo et al. [YHTS09] were among the researchers that investigated the effectiveness of clustering method for prioritizing test cases. The method that Yoo et al. [YHTS09] developed combines clustering with Analytic Hierarchy Process. According to code coverage, they then clustered the test cases. The second step was to prioritize the test cases throughout each cluster based on code coverage and expert knowledge, and choose the test case with the highest priority from each cluster as the cluster's representative. Lastly, their representative test cases were used to perform inter-cluster prioritization. The final cluster priority assignment was then performed based on the assigned priorities from the third step, rearranging the clusters in a circular order. In Carlson et al.'s [CDD11] test prioritization strategy, code coverage, code complexity, and the test case's execution history determine the priority of test cases in each cluster. In the beginning, only code coverage was used to cluster test cases. Next, test cases were prioritized by code coverage, complexity, and history of executions features, separately. After that, they determined the priority based on the combination of code complexity and execution history. Lenz et al. [RPR13] used the execution of few examples of test cases to generate functional clusters of test cases. There are several researchers that used the K-means algorithm, which requires a predefined number of clusters in the beginning, among them are Chen et al.[CCZ+11], Wang et al. [WCF+12], Kandil et al. [KMB17] and Zumar et al. [KQ19]. Others used a hierarchical clustering algorithm, that also need the number of clusters as input, such as the above discussed works of Carlson et al. [CDD11] and Yoo et al. [YHTS09]. Other works such as Chen et al. [CCZ+11], Wang et al. [WCF+12], Kandil et al. [KMB17] and Zumar et al. [KQ19] used the K-means algorithm, which requires a predefined number of clusters in the beginning. Both algorithms require extensive experiments and tuning to decide on the right number of clusters. Carlson et al. [CDD11], Chen et al. [CCZ+11] and Wang et al. [WCF+12] used Euclidean Distance to calculate distance between data instances. In contrast, Kandil et al. [KMB17] and Yoo et al. [YHTS09] used the Hamming Distance.

These approaches showed that clustering algorithms could be beneficial for test prioritization. However, we believe that those algorithms have some drawbacks [PBGB21]: first, the effectiveness of clustering algorithms depends on the number of clusters and distance metrics, which must be adjusted accordingly. Second, it is extremely difficult to find the optimal solution to most clustering algorithms as they are NP-hard problems. With a large test suite, this can be a scalability concern. Third, the effectiveness of clustering algorithms is not as strong as it seems, particularly compared with other machine learning based test prioritization techniques

NLP-based Test Case Prioritization Natural Language Processing (NLP) techniques are being used to exploit information that is found in textual software development artifacts (e.g., defect descriptions) or sources of code that is considered as text information [PBGB21]. Thomas et al. [Tho14] generated vectors from test cases using an NLP topic modeling technique, and later calculated the distance between test cases. By using a greedy algorithm, the test cases were prioritized by maximizing their distance from already prioritized cases. Lachmann et al. [LSN+16] transformed test case descriptions into vectors based on word occurences, which they created of commonly occurring words in the description of all test cases. They employed a ranked Support Vector Machine (SVM) to prioritize test cases using the transformed textual data with code coverage and execution history. Other researchers, Aman et al. [AAYK20] used used three different kinds of NLP techniques, including topic modeling, Doc2Vec (PV-DM), and Doc2Vec (PV-DBoW) to vectorize test cases. They calculated the distance between pairs of test cases using 3 distance metrics; Angular Distance, Manhattan Distance, and Euclidean Distance. They gave the highest priority to the case which was most distant. Afterwards, they prioritized the remaining test cases in accordance with their distance from the prioritized testing set. Medhat et al. [MMBT20] used LSTM algorithm [HS97] to classify specifications, that are preprocessed using NLP. Those specifications describe the components of the system under test. Then, search-based approaches (genetic algorithms) are used to prioritize the test cases. Overall, despite its potential, the current use of Natural Language Processing is still not widely used in the context of test prioritization [PBGB21].

Test case selection is choosing a subset of test cases from the test suite, it aims to eliminate redundant or unnecessary test data. Test case selection is defined and explained more in details in the Background Chapter 2. We can divide the existing works into two categories: static test selection and dynamic test selection.

Static Test Selection: Many techniques have been proposed for selecting tests based on static analysis at different levels of granularity of source code. Kung et al. [KGH+93] were among the first that proposed static regression test selection based on class firewall, i.e., the statically computed set of classes that may be affected by a change. Ryder and Tip [RT01] proposed a call-graph-based static change-impact analysis technique and evaluated only one call-graph analysis on 20 revisions of one project [RST+04]. Skoglund and Runeson [SR05, SR07] performed a case study of class firewall, but they used dynamic coverage information together with class firewall to perform regression test selection, whereas Machalica et al. [MSPC19] apply on a Facebook project the class firewall purely statically. Despite the presence of many static regression test selection techniques in the literature, there are few systematic studies of these methods, and evaluations on modern open-source projects were lacking. Legunsen et al. [LHS+16] represent all all prior work on class-firewall-based analyses [KGH+93] and call-graph-based analyses [RT01]. Zhang [ZKK12] presented a test selection strategy based on method- and file-level analysis of test dependency and change information. In cases where a program's control flow crosses language boundaries, the given techniques are not easily extensible. Furthermore, scaling code dependency analysis in files with million lines of code presents scalability challenges.

Dynamic Test Selection: Several research studied the use of dynamic analysis for test selection strategies. Rothermel et al. [RH97] were among the first that dynamically collected coverage on old revision and investigated test selection using a control-flow graph analysis for C programs. Harrold et al. [HJL+01] extended Rothermel et al. [RH97] work and investigate test selection on Java programs. Other researchers, proposed a two-phase analysis, such as Harrold et al. [HJL+01], because control-flow graph could be time-consuming for large software systems. Some works like Gligoric et al. [GEM15] presented a method based on the granularity of files. Gligoric et al. [GEM15] proposed a method operating at the granularity of files. Celik et al. [CVMG17] described a technique whereby test execution may be traced across languages. The dynamic techniques described above require sufficiently fine-grained execution traces, which are not feasible at Facebook's scale as presented in Machalica et al. work [MSPC19]. Facebook's method is most closely related to Memon et al. [MGN+17] technique, which was developed from a similar industrial context. They analyzed build metadata and structural changes in code bases with an empirical observation that failing tests and changed parts of the code have small distances in build dependency graphs. In Facebook's approach, Machalica et al. [MSPC19] proposed a strategy for selecting tests, where they used distance as one of the features. They found that while distance is an important feature, it is not sufficient to select accurate tests.

Despite various approaches to test optimization for regression testing, the challenge of applying most of them in practice lies in their complexity and the computational overhead typically required to collect and analyze different test parameters needed for prioritization, such as age, test coverage, etc. By contrast, our approach based on RL is a lightweight method, which only uses historical results and its experience from previous CI cycles. Furthermore, our approach is adaptive and suited for dynamic environments with frequent changes in code and testing, and evolving test suites.

4. Quality Metrics for Fault Prediction

This chapter aims to direct the testing in areas in the software that are most likely to be failed since software testing consumes a considerable effort during software production. To raise the effectiveness and efficiency of this effort, it is wise to direct the testing to the code areas which need it most. Therefore, we need to identify those pieces of software that are the most likely to fail and require most of the developers' attention. This chapter presents a methodology that provides prediction models; a combined approach to create accurate failure predictors, to predict the probability of failure of each software component.

4.1 Problem Definition

Overall, software quality assurance is the most expensive activity in the development of safety-critical embedded software [RSHN14, HOBK17, OMS18]. As a result, given the size, complexity, time, and cost constraints in automotive development projects, increasing the effectiveness and efficiency of software quality assurance is becoming increasingly important. As a result, in order to improve the effectiveness and efficiency of software quality assurance tasks, we must first identify problematic code areas that are most likely to contain program flaws and then focus quality assurance tasks on such code areas. An early estimate of fault-proneness aids in making decisions on testing, code inspections, and design rework, as well as financially planning for potential delayed releases, and affordably guide corrective actions to the quality of the software [SPVV17, OMS18].

There are several sources to estimate fault-proneness of software components extracted from our research works [OMS18, OSM19]. It can be:

1. their failure history, that can be extracted from bug databases; a software component likely to fail in the past is likely to fail in the future [NBZ06]. Unfortunately, a long failure history is required in order to get accurate predictions. Such a long failure history, however, is usually not available. Moreover, maintaining long failure histories is usually avoided altogether [NBZ06],

2. the program code itself. Several case studies [BWIL99, TKC99, SK03, MBdNS17] have shown that static code analysis and code complexity metrics correlate with fault density. Static analysis evaluates software programs at compile time by exploring all possible execution paths [FMB$^+$16, ARB17]. In addition, static analysis tools can detect low-level programming faults such as potential security violations, run-time errors and logical inconsistencies [SSSJ17]. Code complexity metrics have been proposed in different case studies to assess software quality [NBZ06, RSHN14, YY17].

3. mining the software repositories to develop an accurate fault prediction model. Software repositories contain historical and valuable information about the overall development of software systems [OMS18, OSM19]. By mining the software git repositories, we can extract and analyze data about the system, such as code churn metrics, which represent the system's change history, and the bug fixes information [OSM19].

4.2 Research Goals

The major question in this approach is whether or not we can use code complexity metrics combined with static analysis fault density to predict pre-release fault density, that is: is combining static analysis metrics with code complexity metrics and code churn metrics a leading indicator of faulty code? The research questions that we want to confirm in this chapter are:

RQ1: Can static analysis fault density combined with code complexity metrics be used to predict pre-release fault density at statistically significant levels?;

RQ2: Can static analysis fault density combined with code complexity metrics be used to discriminate between fault-prone and not fault-prone components?

RQ3: Can the history of code changes between different releases enhance our prediction and classification models?

4.3 Approach

In this chapter, we apply a combined approach, represented in Figure 4.1, to create accurate fault predictors. Our process can be summarized in the following three steps:

4.3.1 Data Pre-Processing

First, we collect the data required to train and test the fault prediction models (the regressor and the classifier) for all software releases. The data required to train our fault prediction models are:

1. **Independent variables**: The independent variables are the input variables to the prediction models. (a) *Static analysis faults*: we execute static code analysis on each component for each release. We define the static analysis fault density of a software component as the number of faults found by static analysis tools, after reviewing and eliminating false positives, per KLOC (thousand lines of code). (b) *Code complexity metrics*: we compute different code complexity metrics for each component and for each release as describes in Table 4.2. (c) *Code churn metrics*: we mine the git repositories databases to extract several code churn metrics (e.g., added LOC, removed LOC, etc., see Table 4.2) for each component in each release.

2. **Dependent variable**: The dependent variable is the output that our prediction models will predict. We validate our approach within two experiments based on two different dependent variables:

    a) Pre-release faults: we mine the archives of several major software systems at Daimler and map their pre-release faults (faults detected during development) back to their individual components. We define the pre-release fault density of a software component as the number of faults per KLOC found by other methods (e.g., testing) before the release of the component, as we did not have access to the source code and git repositories at the beginning of our first experiment, described in Section 4.4.2.1.

Table 4.1: Software Projects Researched

Projects	# Components	Code Size
Project 1	8	2.026 MLOC
Project 2	4	1.762 MLOC
Project 3	9	4.795 MLOC
Project 4	3	3.555 MLOC
Project 5	21	5.070 MLOC
Project 6	2	1.664 MLOC
Project 7	4	2.215 MLOC
Project 8	3	2.710 MLOC

b) Bug Fix: We mine the git repositories using natural language processing techniques to parse and analyze commit messages mentioning bug fixes keywords (e.g., bug fix, bug fixing, etc.). The source code of the implemented tool "pygitminer" is available for peer review [1]. . Such bug fix commits are the indicator of the true known fault density of the software components for each release, described in Section 4.4.2.1.

4.3.2 Model Training

We train different machine learning models to learn the fault densities of each software component based on the independent variables: (a) static analysis faults densities, (b) code complexity metrics, and (c) code churn metrics. We use the standard statistical principal component analysis (PCA) technique on the static analysis fault densities, code complexity metrics, and code churn metrics to overcome the issue of multicollinearity associated with combining several metrics as input variables of the statistical models [OMS18, OSM19]. We split our data into two parts: (1) train data which accounts for three successive releases of all software components, and (2) test data representing the fourth release (the last release).

4.3.3 Model Prediction

The trained statistical models are used to (i) predict pre-release fault densities of software components (Regression) and (ii) discriminate fault-prone software components from the not fault-prone software components (Classification).

4.4 Study Design

This chapter aims to come up with fault predictors that evaluate our research questions, described in Section 4.2.

4.4.1 Researched Projects

Our experiments were carried out using eight software projects of an automotive head unit control system (Audio, Navigation, Phone, etc.) [OMS18, OSM19]. In the remainder of the work, we will be referring to these eight projects as Project 1 to 8. For reasons of confidentiality, we cannot disclose which number stands for which project. Each project, in turn, is composed of a set of components. The total number of components is 54. These components have a collective size of 23.797 MLOC (million LOCs without comments and spaces). All components use the object-oriented language C++. Table 4.1 presents a high-level outline of each project.

4.4.2 Data Collection

The data required to build our fault predictors are:

[1] Implementation available at `https://github.com/so1188/gitrepomining`

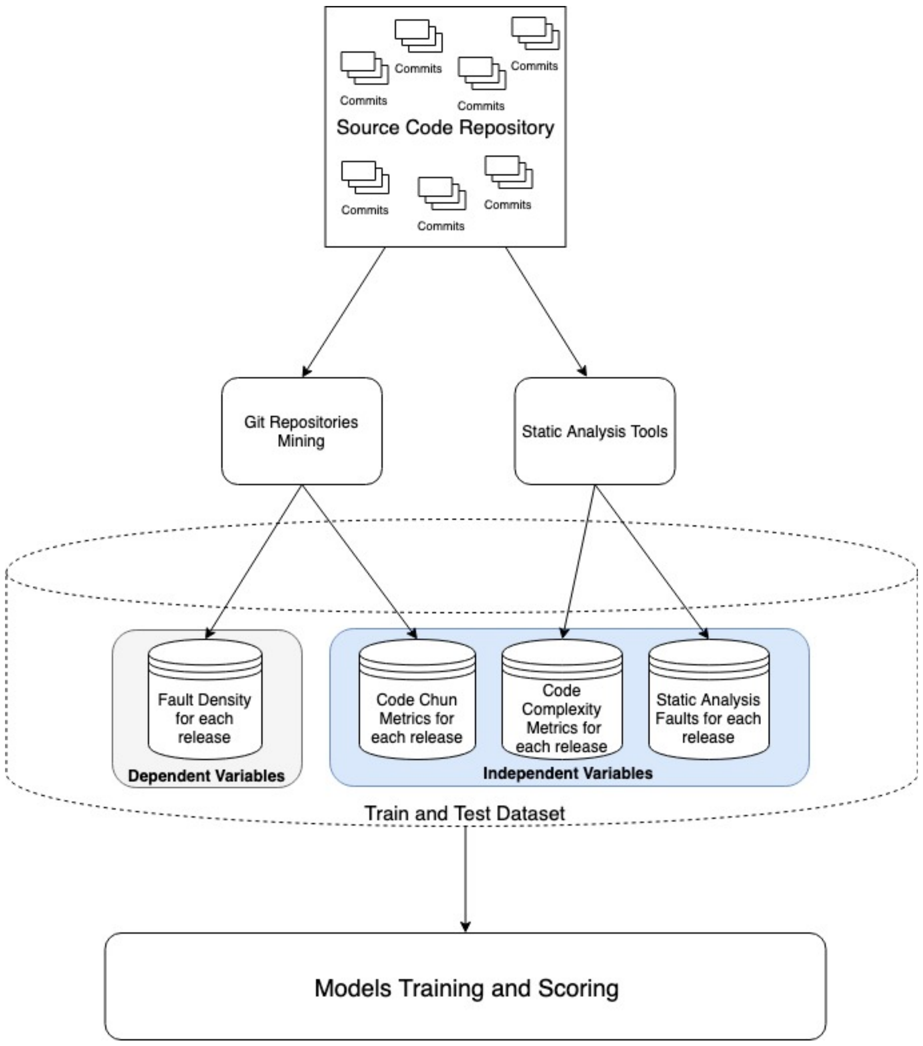


Figure 4.1: Overview of the fault prediction process

4.4.2.1 Faults Data

During our experiment, we identified and validated our work in two kinds of faults:

- Daimler systematically records all problems that occur during the entire product life cycle. In this work, we are interested in pre-release faults, that is, faults that have been detected before the initial release. For each component of the eight projects (Project 1 to Project 8 from Table 4.1), we extracted all bugs detected for the latest release from an internal fault database. The fault database is continuously updated from testing teams, integration teams, external teams, or third-party testers. Such faults do not include problems submitted by customers in the field, which are only found in post-release scenarios. The faults extracted from the fault database are then used to compute the pre-release fault density.
- Faults that have been detected during the development and mentioned as bug fixes in git commits. For each component, we extracted all detected bugs through mining the git repositories. The extracted faults are then used to compute the fault density.

4.4.2.2 Static Analysis Fault Density

We executed static analysis tools on each component and extracted the identified faults. These faults were then used to compute the static analysis fault density. We used commercial verifying and non-verifying static analysis tools (Astrée, Coverity, Polyspace, etc.).

4.4.2.3 Code Complexity Metrics

We compute several code complexity metrics for each of the components. The code complexity metrics are represented in Table 4.2. We limit our study to a set of selected metrics that have shown to provide significant quality indicators over a long period of time by the software quality assurance team at Daimler. Code complexity metrics have been shown to correlate with fault density in several case studies [SK03, TKC99, BWIL99], and they have been proposed in different case studies to assess software quality [RSHN14, NBZ06].

4.4.2.4 Code Churn Metrics

Software repositories contain historical information regarding the overall development of software program systems. Mining software databases is nowadays considered one of the most intriguing expanding areas within software engineering. Different recent works have used past changes as indicators for faults because the more changes are done to a part of the source code, the more likely it will contain faults [KZWJZ07, KWZ08]. Thus, we mine the software repositories databases to extract the churn metrics. We use the extracted code churn metrics, as described in Table 4.2 to predict software fault density.

4.4.2.5 Relative Code Churn Metrics

We compute a number of relative code churn metrics for each component, as described in Table 4.2. We show in this study that using relative code churn as a fault predictor is better than using (absolute) code churn predictors. Furthermore, combining relative code churn metrics with code complexity metrics and static analysis faults can accurately predict the fault density with a high degree of sensitivity.

Table 4.2: Metrics used for the study

Metrics	**Description**
Static Analysis Fault Density	# faults found by static analysis tools per KLOC (thousand lines of code).
Code Churn Metrics	
Added LOC	# lines of code added
Removed LOC	# lines of code deleted
Modified Files	# files modified
Files count	# files compiled to create a software component
Developers	# developers
Relative Code Churn Metrics	
Added LOC / Relevant LOC	We expect the larger the proportion of added code to the Relevant LOC, the higher is the probability of the presence of faults in the software component.
Removed LOC / Relevant LOC	We expect the larger the proportion of removed code to the Relevant LOC, the higher is the probability of the presence of faults in the software component.
Modified Files / Files count	We expect the larger the proportion of files in a component that get modified, the higher is the probability of these files introducing faults.
Code Complexity Metrics	
Relevant LOC	# relevant LOCs without comments, blanks, expansions, etc.
Complexity	cyclomatic complexity of a method
Nesting	# nesting levels in a method
Statements	# statements in a method
Paths	# non-cyclic paths in a method
Parameters	# function parameters in a method

4.5 Study Validation

The case study below details the experiments we executed to validate our research questions. Table 4.3 defines abbreviations used in this section. First, we introduce the evaluation metrics used in this experiment to validate both regression and classification models in Section 4.5.1. Second, the results and analysis of the studied research questions are discussed in Section 4.5.2 (**RQ1** and **RQ2**) and in Section (**RQ3**) 4.5.3. Finally, we discuss in Section 4.5.4 the threats to validity.

Table 4.3: Nomenclature

Abbreviations	Description
PCA	Principal component analysis (PCA) is a standard statistical procedure to convert a set of possibly correlated variables into a (typically smaller) set of linearly uncorrelated variables by using a coordinate transformation.
R^2	R squared: coefficient of determination, measures the variance in the predicted variable that is accounted by the regression built using the predictors (code metrics combined with static analysis fault density).
MSE	Mean squared error (MSE) is a measure of the unbiased error estimate of the error variance.
ROC curve	Receiver operating characteristic (ROC) curve, is a popular measure for evaluating classifier performance. The ROC curve is created by plotting the true positive rate against the false positive rate at various threshold settings.
AUC	Area under curve (AUC) equals the probability that the classifier predicts a randomly chosen true positive higher than a randomly chosen false negative. The larger the AUC, the more accurate is the classification model.

4.5.1 Evaluation Metrics

All defined metrics in this section are discussed in detail in the Background Chapter 2 in Section 2.3.

As a measure of the regression fits, we compute R^2. R^2 measures the variance in the predicted variable that is accounted by the regression built using the predictors. For evaluating and reporting the performance of our regression model, we use the commonly used error metric; the mean squared error (MSE) to measure the unbiased error estimate of the error variance.

As evaluation measures for the classification models, we use the accuracy, where a confusion matrix in Table 4.4 is used to store the correct and incorrect decisions made by a classification model. In order to compare the actual observed and predicted classes for each component, we categorized each predicted class into four individual categories, as shown in Table 4.4.

We compute precision and recall as (i) $precision = TP \div (TP + FP)$ and (ii) $recall = TP \div (TP + FN)$, as described in Chapter 2 in Section 2.3. All two measures are values between zero and one. A precision of one indicates that the classification model does not report any false positives. A recall of one implies that the model does not report any false negatives. The F-measure can be interpreted as a weighted average of the precision and recall, where an F-measure reaches its best value at one and worst at zero. The intuition behind precision, recall, and F-measure is:

- Precision: how many of the components classified by our classifiers as fault-prone are actually fault-prone.
- Recall: how many fault-prone components our classifiers were able to identify correctly as fault-prone.
- F-measure: measures the weighted harmonic mean of the precision and recall.

In order to evaluate the classifier output quality, we used the Receiver Operating Characteristic (ROC) metric. In binary classification, ROC curves are commonly used to investigate the output of a classifier. To apply the ROC curve and ROC area to multi-class or multi-label classification, the output must be binarized [ROC]. Therefore we binarize our output, that is, the pre-release defects density.

Table 4.4: Comparing observed and predicted component classes in a confusion matrix. Used to compute precision and recall values of classification model

		Observed class	
		fault prone	non-fault prone
Predicted class	fault prone	True negative (TN)	False negative (FN)
	non-fault-prone	False positive (FP)	True positive (TP)

4.5.2 Static Analysis and Code Complexity Metrics as Early Indicators of Software Defects

In this section, we validate the two following research questions:

RQ1: Can static analysis fault density combined with code complexity metrics be used to predict pre-release fault density at statistically significant levels?;

RQ2: Can static analysis fault density combined with code complexity metrics be used to discriminate between fault-prone and not fault-prone components?

4.5.2.1 Correlation Analysis

In order to confirm that static analysis fault density combined with code complexity metrics can be used as an early indicator of pre-release fault density, we investigate the possible correlations between the pre-release fault density and the code complexity metrics as well as the static analysis fault density. We applied a robust correlation technique, Spearman rank correlation. Spearman rank correlation has the advantage over other correlation techniques such as Pearson correlation to detect also non-linear correlations between elements [Wis05], [JCFdW16]. Table 4.5 summarizes the correlation results. It shows a statistically significant positive correlation between the static analysis fault density and the pre-release fault density. It also shows a statistically significant positive correlation between the pre-release fault density and some of the code complexity metrics. The correlations between the code complexity metrics (row 1 to row 6), as well as between the code complexity metrics and the static analysis fault density (row 7), are an early indicator of the existence of the multicollinearity problem when using both code complexity metrics and static analysis fault density as input parameters of statistical prediction models. The negative correlations are to be explained that the complex modules we are considering in this study are already in an advanced development stage and have been intensively tested.

In this work, we assume statistical significance at 99% confidence. Furthermore, all metrics are normalized before computing the correlations.

Table 4.5: Correlation results of pre-release fault density with code metrics and static analysis fault density (All correlations are significant at the 0.01 level (2-Tailed))

Metric	Statements	Parameters	Nesting	Paths	Complexity	R_LOC	Static Analysis Fault Density	Pre-Release Fault Density
Statements	1							
Parameters	0.55	1						
Nesting	-0.32	0,042	1					
Paths	-0.079	0.33	0.84	1				
Complexity	0.3	0.42	-0.13	-0.055	1			
Relevant_LOC	-0.3	- 0.13	0.79	0.46	-0.13	1		
Static Analysis Fault Density	-0.22	0.13	0.37	0.067	0.31	0.52	1	
Pre-Release Fault Density	**0.7**	**0.82**	-0.15	0.079	**0.55**	-0.13	**0.69**	1

4.5.2.2 Predictive Analysis

In order to estimate the pre-release fault density, we applied statistical regression techniques where the dependent variable is the pre-release fault density, and the independent variables are the code complexity metrics combined with the static analysis fault density. However, one difficulty associated with combining several metrics is the issue of multicollinearity. For instance, (see Table 4.5) the Statement, Parameters and Complexity metrics not only correlate with pre-release fault density, but they also strongly correlated with each other. To overcome the problem of multicollinearity between the independent variables (see Table 4.5 where correlations between code metrics have been identified), we used the standard statistical principal component analysis (PCA) technique. Multicollinearity might lead to an inflated variance in the estimation of the dependent variable, that is the pre-release fault density.

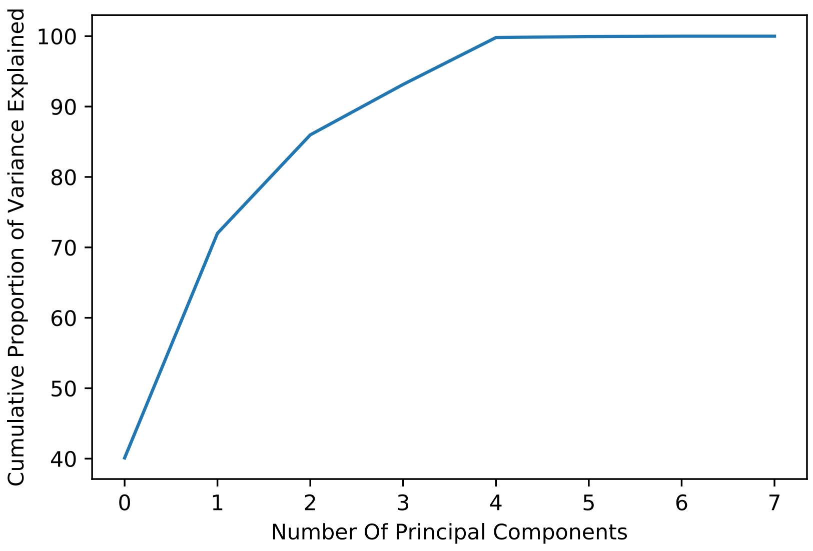


Figure 4.2: Extracted Principal Components

We extracted the principal components out of the 7 independent variables which include the six complexity metrics and the static analysis fault density. Fig. 4.2 shows that 4 principal components result in variance close to 98%. Therefore, in this study, we selected the number of principal components as 4, which we used to model our prediction model. We split our data into two parts: (i) train data which accounts for 70% of the available data, and (ii) test data representing the remaining 30%. We first transform both train and test data into 4 components which explained 98% of the total data variance using PCA. Then, we fit several models to the code complexity data, and the static analysis fault density separately as predictors and the pre-release fault density as the dependent variable. We then combined both code metrics and static analysis fault density as predictors for the pre-release fault density. The models we tested include linear, exponential, polynomial regression models, as well as support vector regressions and random forest. As a measure of the regression fits, we compute R^2. R^2 measures the variance in the predicted variable that is accounted by the regression built using the predictors. As a measure of the unbiased error estimate of the error variance, we use the mean squared error (MSE), as described in Section 4.5.1.

Table 4.6 shows that when using both the code complexity metrics and the static analysis fault density as predictors, we obtain the best fit using the random forest model; the R^2 value increases to 0.783, and the MSE decreases to 0.216. Therefore, we conclude that combining code metrics and static analysis is more beneficial to explain pre-release faults. We do not present the regression equations to protect proprietary data. The validation of the model goodness is repeated 10 times using the 10-fold cross-validation technique.

Table 4.6: Regression Fits

Predictors	R^2	MSE
Static Analysis Fault Density **alone**	0.676	0.323
Complexity Metrics **alone**	0.507	0.493
Both Complexity Metrics and the Static Analysis Fault Density	**0.783**	**0.216**

In order to address the fact that the above results are not by chance, we repeated the data split (train: 70% and test: 30% of the data) as well as the model fitting several times. To quantify the sensitivity of the results, we applied the Spearman rank correlation between the predicted and the actual pre-release fault densities. Table 4.7 reports the correlations as well as the accuracy (R^2) of the prediction when applied on three random splits of the data. The Spearman correlation shows a strong positive correlation, always stronger when both complexity metrics and static analysis are combined.

Table 4.7: Summary of fit and correlation results of random model sampling

	R^2			Correlation Results (Spearman)			MSE		
	Complexity Metrics	Static Analysis	Both (Proposed Model)	Complexity Metrics	Static Analysis	Both (Proposed Model)	Complexity Metrics	Static Analysis	Both (Proposed Model)
Split 1	0.485	0.694	0.895	0.842	0.935	0.946	0.514	0.306	0.104
Split 2	0.625	0.838	0.915	0.791	0.918	0.957	0.374	0.161	0.084
Split 3	0.506	0.590	0.729	0.847	0.789	0.880	0.493	0.409	0.270

4.5.2.3 Classification Analysis

In this section, we discuss the experimental results showing how well the combination of code complexity metrics with static analysis fault density performs with respect to categorizing software components based on their fault-proneness.

In order to classify software components into fault-prone and not fault-prone components, we applied several statistical classification techniques. The classification techniques include random forest classifiers, logistic regression, passive aggressive classifiers, gradient boosting classifiers, K-neighbors classifiers and support vector classifiers. The dependent variables for the classifiers are the code complexity metrics and the static analysis fault density, the independent variable is the result of binarizing (i.e. fault-prone vs. not fault-prone) the pre-release fault density. We binarized the pre-release fault-density to create a binary classification problem (i.e. fault-prone vs. not fault-prone). We split the overall data into $2/3$ training and $1/3$ testing instances using stratified sampling. In order to address the fact that the classification results were not by chance we repeated the data splitting experiments. Our experiments showed that logistic regressions delivered the most accurate classifiers. Table 4.8 shows the accuracy results of the classification based on four data splits when using the logistic regression.

Table 4.8: Precision and recall values for the classification model on four random data splits

	Precision	**Recall**
Split 1	0.739	0.375
Split 2	0.717	0.358
Split 3	0.723	0.357
Split 4	0.725	0.365

We now want to determine the quality of our classification model. The accuracy of a classification model is characterized by misclassification rates. In this work, a Type I misclassification, also called false positive is when the model predicts that a module is not fault-prone when it is. Type II misclassification, also called false negative is when the model predicts that a module is fault-prone when it is not. In order to compare the actual observed and predicted classes for each component, we categorized each predicted class into four individual categories as shown in Table 4.4 in Section 4.5.1. As evaluation measures, we compute precision and recall defined in Section 4.5.1. The intuition behind precision and recall is the following:

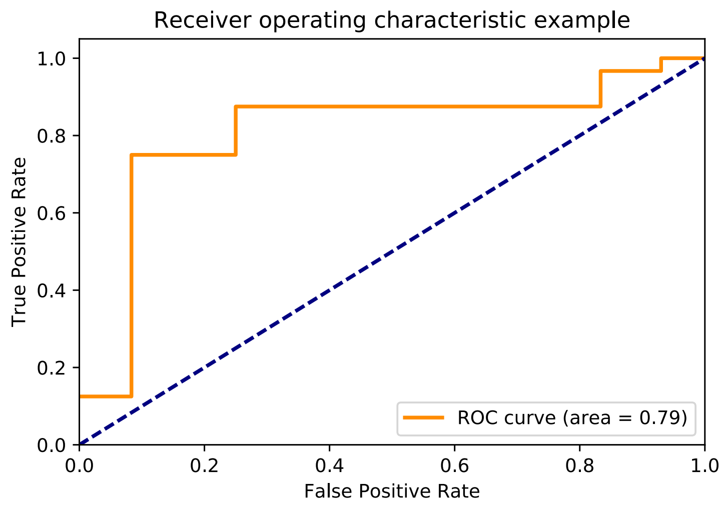
- Recall: how many fault-prone components our classifiers were able to identify correctly as fault-prone.
- Precision: how many of the components classified by our classifiers as fault-prone are actually fault-prone

All two measures are values between zero and one. Table 4.8 reports the recalls and precisions for the classification model on four random data splits. The mean precision over all splits lies at 0.726, the mean recall lies at 0.419 . High precision relates to a low false positive rate, meaning (according to Table 4.4) the probability to classify true fault prone components as non-fault prone ones is low. Conversely, high recall relates to a low false negative rate; meaning low probability to classify true non-fault prone components as fault prone. The recall value of our classification model is modest and needs to be further improved. Nevertheless, our model still delivers a safe classification; non-fault prone components would get more software quality assurance attention, while these components are truly non-fault prone. A visual representation of the performance of the classification model is provided in Fig. 4.3, which plots the Receiver Operating Characteristic (ROC) metric. The area under the ROC curve (AUC) equals the probability that the classifiers predicts a randomly chosen true positive higher than a randomly chosen false negative. The larger the AUC, the more accurate is the classification model. As shown in Fig. 4.3, the accuracy of the classification model lies at 79% (AUC).

Extended Validation

The validation of our study is based on the metrics validation methodology proposed by Schneidewind [Sch92]. Following the Schneidewind's validation scheme, the quality indicator is the pre-release fault density (**F**) and the metric suite (**M**) is the combination of the static analysis

Figure 4.3: ROC curve for logistic regression using code complexity metrics and static analysis fault density to classify software components



fault density together with the code complexity metrics. The six validation criteria are the following:

1. ***Association:*** "This criterion assesses whether there is a sufficient linear association between **F** and **M** to justify using **M** as an indirect measure of **F**" [Sch92]. We identified a linear correlation between the pre-release fault density and the combination of the static analysis fault density with the code complexity metrics at a statistically significant level warranting the association between **F** and **M**.

2. ***Consistency:*** "This criterion assesses whether there is sufficient consistency between the ranks of **F** and the ranks of **M** to warrant using **M** as an indirect measure of **F**" [Sch92]. We demonstrated the consistency between the pre-release faults and the combination of static analysis fault density and code complexity metrics in Section 4.5.2.2 in Table 4.6.

3. ***Discriminative Power:*** "This criterion assesses whether **M** has sufficient discriminative power to warrant using it as an indirect measure of **F**" [Sch92]. This is satisfied based on the results in Section 4.5.2.3, where we showed that discriminant analysis can classify effectively fault-prone from not fault-prone components.

4. ***Tracking:*** "This criterion assesses whether **M** is capable of tracking changes in F (e.g., as a result of design changes) to a sufficient degree to warrant using **M** as an indirect measure of **F**" [Sch92]. Table 4.7 justify the ability of **M** to track **F**.

5. ***Predictability:*** "This criterion assesses whether **M** can predict **F** with required accuracy" [Sch92]. The correlation analysis results as well as the prediction analysis results in Section 4.5.2.2.

6. ***Repeatability:*** "This criterion assesses whether **M** can be validated on a sufficient percentage of trials to have confidence that it would be a dependable indicator of quality in the long run" [Sch92]. We demonstrated the repeatability criterion by using random splitting techniques in Section 4.5.2.2. A limitation of our study with respect to repeatability is that all data used are from one software system.

4.5.3 An Enhanced Fault Prediction Model Based on Code Churn, Complexity Metrics, and Static Analysis Results

We validate in this section the following research question:

RQ3: Can the history of code changes between different releases enhance our prediction and classification models?

4.5.3.1 Experimental Setup

Our experiment is composed of two steps: data pre-processing and model training. First, we collect the data required to train and test the fault prediction models (the regressor and the classifier) out of a git versioned software project. Git versioning allows us to capture the required data for all software releases. The data required to train our fault prediction models is:

- Independent variables which are the input variables to the studied prediction models: (a) Static analysis faults: we execute static code analysis on each component for each release. We define the static analysis fault density of a software component as the number of faults found by static analysis tools, per KLOC (thousand lines of code). (b) Code complexity metrics: we compute different code complexity metrics for each of the components and for each release as describes in Table I. (c) Code churn metrics: we mine the git repositories databases to extract several code churn metrics (e.g, added LOC, removed LOC, etc., see Table I) for each release.
- Dependent variable which is the output that will be predicted by our prediction models. We mine the git repositories using natural language processing techniques to parse and analyze commit messages mentioning bug fixes keywords (e.g, bug fix, bug fixing, etc.). Such bug fix commits are the indicator of the true known fault density of the software components for each release.

All studied metrics are discussed in details in the Data Collection Section 4.4.2 in 4.4 and they are represented in Table 4.2.

Second, We train different machine learning models to learn the fault densities of each software component based on the independent variables: (a) static analysis faults densities, (b) code complexity metrics, and (c) code churn metrics. We split our data into two parts: (1) train data which accounts for 3 successive releases of all software components, and (2) test data representing the fourth release (the last release). We split our data into two parts: 1) train data which accounts for 80% of the available data, and 2) test data representing the remaining 20%. We tested our data on the four main regression models families (Generalized Linear models, Deep Learning Models, Random Forest Models and Boosted Models). The boosted models include RGBoost (also known as regularized gradient boosting), Distributed Random Forests (DRF) as well as Gradient Boosting Machines (GBM). We will shortly explain the model that we used in this study to predict the fault density. RGBoost is a supervised learning algorithm that implements a process called boosting to yield accurate models [CBH$^+$16]. Boosting refers to the ensemble learning technique of building many models sequentially, with each new model attempting to correct for the deficiencies in the previous model [Zho21]. In tree boosting, each new model that is added to the ensemble is a decision tree. RGBoost provides parallel tree boosting that solves many data science problems in a fast and accurate way. For many problems, RGBoost is one of the best gradient boosting machine frameworks today [CBH$^+$16]. Both RGBoost and GBM follows the principle of gradient boosting. There are, however, differences in modeling details. Specifically, RGBoost uses a more regularized model formalization to control over-fitting, which gives it better performance, especially when the correlation between the independent variables is non-linear. Distributed Random Forest (DRF) is a powerful classification and regression tool. When given a set of data, DRF generates a forest of classification or regression trees, rather than a single classification or regression tree [GBT18]. The validation of the model goodness is repeated 10 times using the 10-fold cross-validation technique.

Table 4.9: Regression Fits of three Predictors and their Combination

		RGBoost		DRF		GBM	
		R^2	MSE	R^2	MSE	R^2	MSE
Predictors	Absolute Code Churn Metrics **alone**	0.473	0.235	0.325	0.337	0.592	0.195
	Relative Code Churn Metrics **alone**	0.730	0.113	0.651	0.267	0.694	0.221
	Relative Code Churn Metrics **Combined With** Code Complexity Metrics and Static Analysis Fault Density	**0.857**	**0.015**	0.683	0.103	0.784	0.067

4.5.3.2 Model Fitting and Regression Analysis

In this section we compare predictive models built using the different metrics presented in Table 4.2 in order to find the best model for accurate fault prediction to enhance the our model described in Section 4.5.2. We fit several models to the absolute code churn data as well as the relative code churn data separately as predictors, with the fault density as the dependent variable. We tested our data on the four main regression models families (Generalized Linear models, Deep Learning Models, Random Forest Models and Boosted Models). The experiment shows that boosted models are showing the best fitting and generalized accuracy. This result can be explained by the fact that the relation between the independent variables is highly non-linear. The boosted models include RGBoost (also known as regularized gradient boosting), Distributed Random Forests (DRF) as well as Gradient Boosting Machines (GBM). As a measure of the regression fits, we compute R^2 and as a measure of the unbiased error estimate of the error variance, we use the mean squared error (MSE). The evaluation measures are introduced in Section 4.5.1 and discussed in more details in the Background Chapter 2. The regression model fit for absolute code churn metrics has an R^2 value of 0.473, an MSE value of 0.235. Nevertheless, using the relative code churn metrics as fault predictors shows a better fit; the R^2 value increases to 0.730, the MSE decreases to 0.113. We then combined relative code churn metrics with code complexity metrics and with static analysis fault density as predictors for the fault density.

Table 4.9 shows that when using the combination relative code churn metrics with code complexity metrics and with static analysis fault density as fault predictors, we obtain the best fit using the Regularized Gradient Boosting (RGBoost) model; the R^2 value increases to 0.857, the MSE decreases to 0.015. Therefore, we conclude that it is more beneficial to combine relative code churn metrics with code complexity metrics and static analysis fault density to explain software faults. The validation of the model goodness is repeated 10 times using the 10-fold cross-validation technique. A benefit of using ensembles of decision tree methods like regularized gradient boosting is that they can automatically provide estimates of feature importance from a trained predictive model, as presented in Figure 4.4.

4.5.3.3 Fault-Proneness Analysis

In order to classify software components into fault-prone and not fault-prone components, we applied several statistical classification techniques. The classification techniques include the same techniques that we considered for the regression; RGBoost, DRF and GBM. The independent variables for the classifiers are the relative code churn metrics combined with the code complexity

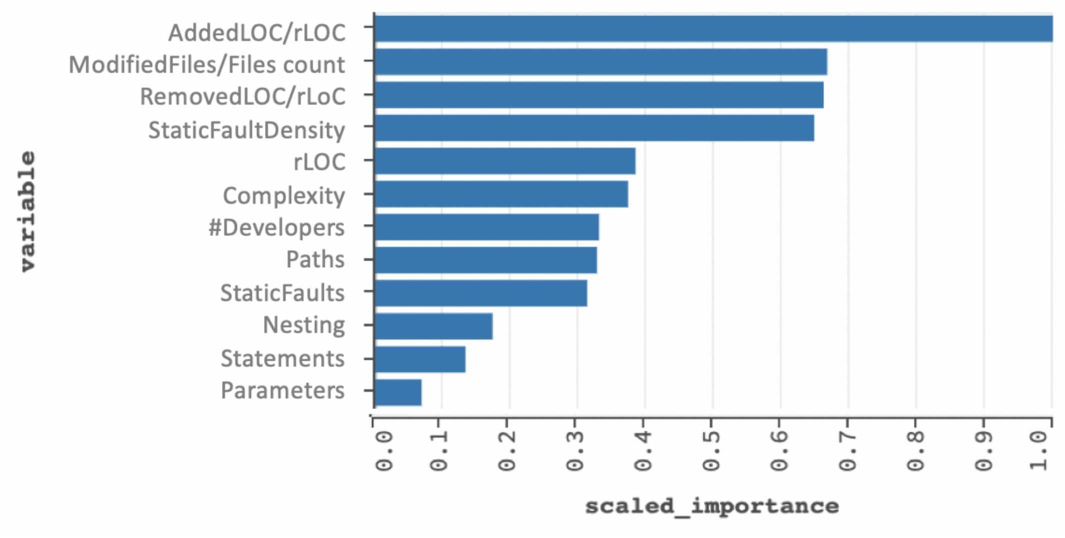


Figure 4.4: Variable Importance

Table 4.10: Classification Performance for Our Approach using RGBoost, DRF and GBM Models

	RGBoost	DRF	GBM
Precision	0.94	0.33	0.45
Recall	0.91	1	0.96
F-measure	0.92	0.496	0.613
AUC	0.88	0.6	0.73

metrics and the static analysis fault density. The dependent variable is the result of binarizing (i.e., fault-prone vs. not fault-prone) the fault density. A confusion matrix, as defined in Table 4.4 in Section 4.5.1, is used to store the correct and incorrect decisions made by a classification model.

For instance, if a component is classified as fault-prone when it is truly fault-prone, the classification is true positive (TP). If the component is classified as fault-prone when it is actually clean (not fault-prone), then the classification is a false positive (FP). If the file is classified as clean when it is in fact fault-prone, the classification is a false negative (FN). Finally, if the issue is classified as clean and it is, in fact, clean, the classification is true negative (TN). In order to compare the actual observed and predicted classes for each component, we categorized each predicted class into four individual categories as shown previously in Table 4.4. As evaluation measures, we compute precision, recall, and F-measure defined in Section 4.5.1 as:

- Precision: how many of the components classified by our classifiers as fault-prone are actually fault-prone.
- Recall: how many fault-prone components our classifiers were able to identify correctly as fault-prone.
- F-measure: measures the weighted harmonic mean of the precision and recall.

Furthermore, we investigate the use of the area under the receiver operating characteristic (ROC) curve (AUC) as a performance measure for our approach. The area under the ROC curve (AUC) equals the probability that the classifiers predict a randomly chosen true positive higher than a randomly chosen false negative. The larger the AUC, the more accurate is the classification model, as defined in Section 4.5.1.

As shown in Table 4.10 and in Figure 4.5, the classification model which uses RGBoost as the classifier produced an impressive result with all four performance indicators (Precision, Recall, F-measure and AUC) being well above 0.9. Using DRF or GBM achieved very high recall, but at the same time it appeared to produce many false positives, and thus their precision is much lower than the precision produced by RGBoost. All studied classifiers achieved an AUC well above the 0.5 threshold; 0.89 for RGBoost, 0.6 for DRF and 0.73 for GBM.

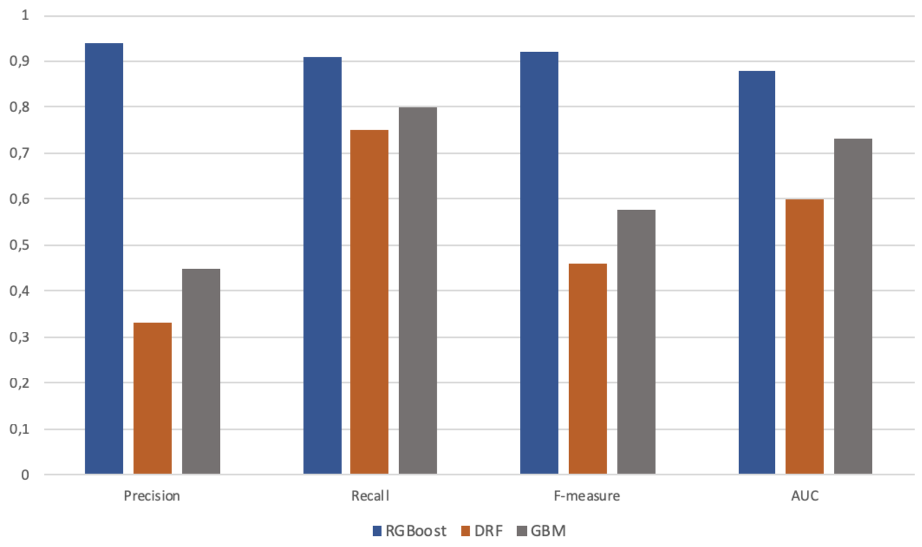


Figure 4.5: Classification Performance of Our Approach

Summary:

We verified in this study whether the faults detected by static analysis tools combined with code complexity metrics can be used as software quality indicators and to build pre-release fault prediction models. The combination of code complexity metrics with static analysis fault density was used to predict the pre-release fault density with an accuracy of 78.3%. This combination was also used to separate high and low quality components with a classification accuracy of 79%. Moreover, we extends this study, where we demonstrated that the combination of code complexity metrics together with static analysis results allows accurate prediction of fault density and to build classifiers discriminating faulty from non-faulty components. The extension presented in this study augments our predictor and classifier with code churn metrics. We applied our methodology to $C++$ projects from Daimler's head unit development. In experiments to separate fault-prone from non-fault-prone components, our new approach achieved a classification accuracy of 89%, and the regressor predicted the fault density with an accuracy of 85.7%. This is an improvement of 7.5% with respect to the accuracy of fault density prediction, and an improvement of 10% to the accuracy of fault classification compared to our previous approach that did not take code churn metrics into account.

4.5.4 Threats to Validity

The results reported in our study (i) are heavily dependent on the quality of the static analysis tools used at Daimler as well as the quality of the manual reviews executed to eliminate false positives, and (ii) might not be repeatable with the same degree of confidence with other tools. The defects detected by static analysis tools might be false positive. In our study, an internal review of the faults identified by static analysis tools has been executed, and only true positive faults have been used in this study. Moreover, it is possible that static analysis tools do not detect all faults during the development process. Such missed faults can be found by testing, which would increase the pre-release fault density and consequently might perturb the correlation. In order to mitigate possible skewness of the correlation, our hypothesis was that combining the static analysis fault density with code complexity metrics and code churn metrics would account for faults not identified by static analysis tools.

4.6 Related Work

This section presents the existing works related to our first contribution described in this Chapter 4. Software code complexity metrics were initially suggested by Chidamber and Kemerer [CK94]. Basili et al. [BBM96], and Briand et al. [BWIL99] were among the first to use such metrics to validate and evaluate fault-proneness. Subramanyam and Krishnan [SK03], and Tang et al. [TKC99] showed that these metrics can be used as early indicators of external software quality. Nagappan et al. [NBZ06] empirically confirmed that code complexity metrics can predict post-release faults. Based on a study on five large software systems, they showed that 1) for each software, there exists a set of complexity metrics that correlates with post-release faults, 2) there is no single set of metrics that fits all software projects, 3) predictors obtained from complexity metrics are good estimates of post-release defects, and 4) such predictors are accurate only when obtained from the same or similar software projects. Our work builds on the study of Nagappan et al. [NBZ06], and focuses on pre-release faults while taking into consideration not only the code complexity metrics but also the faults detected by static analysis tools to build accurate pre-release fault predictors.

In this work, we used the faults detected by static analysis tools to predict the pre-release fault density. Our basic hypothesis is that while static analysis tools only find a subset of the actual faults in the program's code, it is highly likely that these detected faults, combined with code complexity metrics would be a good indicator of the overall code quality. This is explained by the fact that static analysis tools can find faults that occur on paths uncovered by testing. On the other hand, testing has the ability to discover deep functional and design faults, which can be hardly discovered by static analysis tools. In other words, code complexity metrics would complement the static analysis fault detection capabilities to account for the type of faults that cannot be detected by static analysis tools, and hence such a combination can form accurate predictors of pre-release faults.Nagappan et al. [NWH$^+$04] showed at Nortel Networks on an 800 KLOC commercial software system, that automatic inspection faults detected by static analysis tools were a statistically significant indicator of field failures and is effective to classify fault-prone components. Nagappan et al. [NB05a] applied static analysis at Microsoft on a 22 MLOC commercial system and showed that the faults found by static analysis tools were a statistically significant predictor of pre-release faults and can be used to discriminate between fault-prone and non fault-prone components. Again, our approach does not only make use of the faults detected by static analysis, but also uses code complexity metrics; it goes beyond the works of Nagappan et al. by not only using the faults detected by static analysis tools as an indicator of pre-release faults, but also combines these faults with code complexity metrics in a mathematical model which delivers a more accurate predictor and classifier of pre-release faults.

Furthermore, faults are closely related to changes made in the software systems and studying the changes that take place during software evolution via code churn is also important. Khoshgoftaar et al. [KAG$^+$96] were among the first to use past changes for bug prediction. Their objective was to classify the modules as fault-prone or not. Therefore, they identified modules where debug code churn exceeded a threshold. They showed, by studying the change history of two consecutive releases of a large legacy software system of telecommunications, that a high code churn, i.e., a high amount of lines added and removed, is a good indicator of fault-prone modules. The system studied contain over $38,000$ procedures in 171 modules. Ohlsson et al. [OvMMW], Graves et al. [GKMS00] studied the evolution of changes in the software systems to understand their relationship with software quality. Based on a study on eight large-scale open source systems (Eclipse, Postgres, KOFFICE, gcc, Gimp, JBOSS, JEdit and Python), Zimmermann et al. [ZPZ07] mined the version histories and predicted the location of future changes in systems with an accuracy of 70%. Closely related to our study is the work performed by Nagappan and Ball [NB05b] on predicting defect density in software systems using relative code churn metrics, i.e., code churn weighted by lines of code. They analyze different code churn measures in isolation, and show that relative code churn is better than absolute code churn values to predict defects at statistically significant levels. Their approach is similar to ours in the sense that we are also considering relative churn variables to predict fault potential. However, we focus on predicting fault density on an extended number of variables including code complexity metrics and the faults detected by static analysis tools.

To the best of our Knowledge, this work is the first to combine code churn metrics with code complexity metrics and with static analysis results to predict software defect density for each software component.

In the following sections, we will discuss in details the numerous studies that have examined the realm of software fault prediction in the recent decades.

Software Fault Prediction

Fault prediction is an active research area in the field of software engineering. Many techniques and metrics have been developed to improve fault prediction performance. Figure 4.6 briefly shows the history of software fault prediction studies in about the last 20 years.

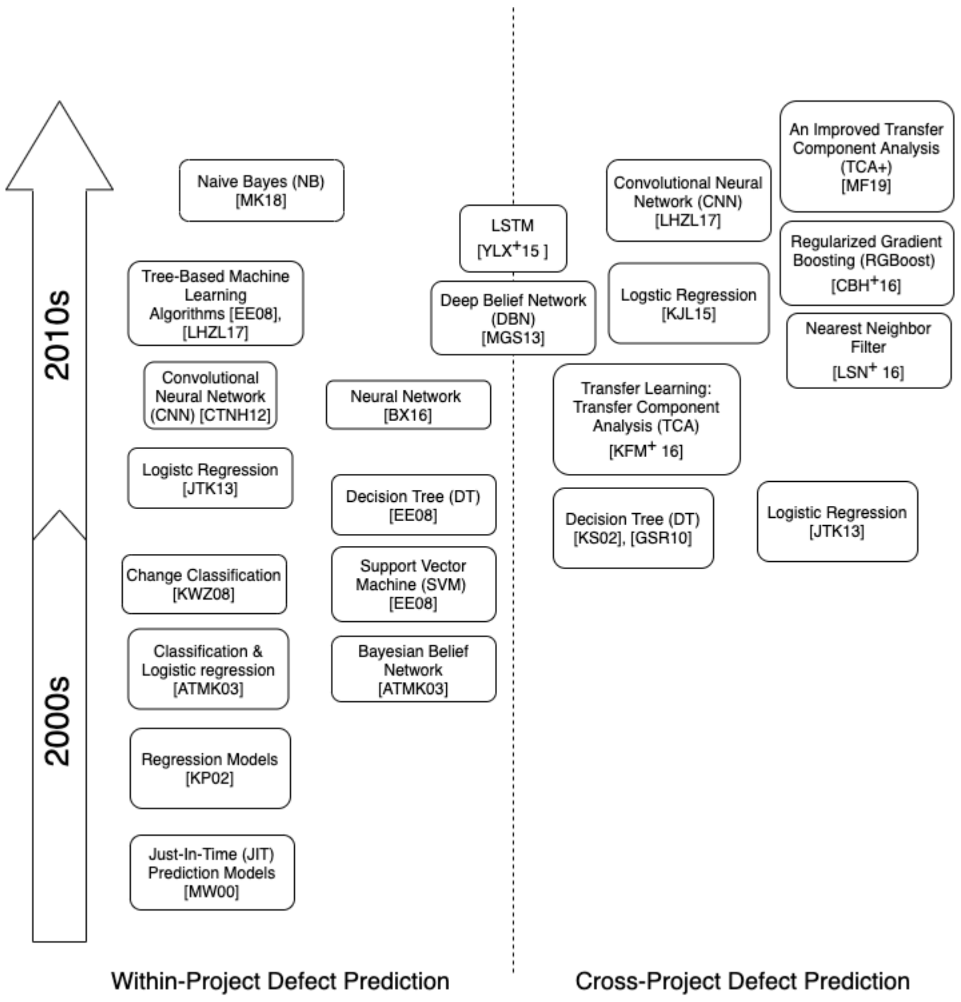

2010s
2000s
Naive Bayes (NB) [MK18]
Tree-Based Machine Learning Algorithms [EE08], [LHZL17]
Convolutional Neural Network (CNN) [CTNH12]
Logistc Regression [JTK13]
Change Classification [KWZ08]
Classification & Logistic regression [ATMK03]
Regression Models [KP02]
Just-In-Time (JIT) Prediction Models [MW00]
LSTM [YLX+15]
Deep Belief Network (DBN) [MGS13]
Neural Network [BX16]
Decision Tree (DT) [EE08]
Support Vector Machine (SVM) [EE08]
Bayesian Belief Network [ATMK03]
Convolutional Neural Network (CNN) [LHZL17]
An Improved Transfer Component Analysis (TCA+) [MF19]
Regularized Gradient Boosting (RGBoost) [CBH+16]
Logstic Regression [KJL15]
Nearest Neighbor Filter [LSN+ 16]
Transfer Learning: Transfer Component Analysis (TCA) [KFM+ 16]
Decision Tree (DT) [KS02], [GSR10]
Logistic Regression [JTK13]
Within-Project Defect Prediction
Cross-Project Defect Prediction

Figure 4.6: History of Software Defect Prediction

Within-Project Defect Prediction uses training data and test data that are from the same project. Many machine learning algorithms have been adopted for within-project defect prediction, including Support Vector Machines (SVM) [EE08], Bayesian Belief Networks [ATMK03], Naive Bayes (NB) [WL], Decision Trees (DT) [GSR10], [KS02], [WSC], Neural Networks (NN) [EMG12], or Dictionary Learning [JYZ+14]. Elish et al. [EE08] evaluated the feasibility of SVM in predicting defect-prone software modules, and they compared SVM against eight statistical and machine learning models on four NASA datasets. Their results showed that SVM is generally better than, or at least competitive with other models, e.g., Logistic Regression, Bayesian techniques, etc. Amasaki et al. [ATMK03] used a Bayesian Belief Network to predict the final quality of a software product. They evaluated their approach on a closed project, and the results showed that their proposed method can predict bugs that the Software Reliability Growth Model (SRGM) cannot handle. Wang et al. [WSC] and Khoshgoftaar et al. [KS02] examined the performance of tree-based machine learning algorithms on defect prediction. Their results indicate that tree-based algorithms can generate good predictions. Tao et al. [WL] proposed a Naive Bayes based defect prediction model, and they evaluated the proposed approach on 11 datasets from the PROMISE defect data repository. Their experimental results showed that the Naive Bayes based defect prediction models could achieve better performance than $J48$ (decision tree) based prediction models. Jing et al. [JYZ+14] introduced the dictionary learning technique to defect prediction. Their cost-sensitive dictionary learning based approach could significantly improve defect prediction in their experiments. Wang et al. [WLT16] used a Deep Belief Network (DBN) to generate semantic features for file-level defect prediction tasks. In Wang et al.'s work [WLT16], to evaluate the performance of DBN-based semantic features as well as traditional features, they built prediction models by using three typical machine learning algorithms, i.e., ADTree, Naive Bayes, and Logistic Regression. Their experimental results show that the learned DBN-based semantic features consistently outperform the traditional defect prediction features on these machine learning classifiers. Most of the above approaches are designed for file-level defect prediction. For change-level defect prediction, Mockus and Weiss [MW00] and Kamei et al. [KSA+13] predicted the risk of a software change by using change measures, e.g., the number of subsystems touched, the number of files modified, the number of added lines, and the number of modification requests. Kim et al. [KWZ08] used the identifiers in added and deleted source code and the words in change logs to classify changes as being fault-prone or not fault-prone. Jiang et al. [JTK13] and Xia et al. [XLWY16] built separate prediction models with characteristic features and meta features for each developer to predict software defects in changes. Tan et al. [TTDM15] improved change classification techniques and proposed online defect prediction models for imbalanced data. Their approach uses time sensitive change classification to address the incorrect evaluation introduced by cross-validation. McIntosh et al. [MK18] studied the performance of change-level defect prediction as software systems evolve. Change classification can also predict whether a commit is buggy or not [PDS+15], [PP14], [HP19]. In Wang et al.'s work [WLT16], they also compare the DBN-based semantic features with the widely used change-level defect prediction features, and ther results suggest that the DBN-based semantic features can also outperform change-level features. However, sufficient defect data is often unavailable for many projects and companies. This raises the need for cross-project bug localization, i.e., the use of data from one project to help locate bugs in another project.

Cross-Project Fault Prediction: Due to the lack of data, it is often difficult to build accurate models for new projects. Recently, more and more papers studied the *cross-project defect prediction* problem, where the training data and test data come from different projects.

Some studies ([KMT07], [MMT+10], [ZNG+09]) have been done on evaluating cross-project defect prediction against within-project defect prediction and show that cross-project defect prediction is still a challenging problem. He et al. [HPMY13] showed the feasibility to find the best cross-project models among all available models to predict defects on specific projects. Turhan et al. [TMBDS09] proposed a nearest-neighbor filter to improve cross-project defect prediction. Zimmermann et al. [ZNG+09] evaluated the performance of cross-project defect prediction on 12 projects and their 622 combinations. They found that the defect prediction models at that time could not adapt well to cross-project defect prediction. Li et al. [LHZL17] proposed defect prediction via convolutional neural networks (DP-CNN). Their work differs from the above-mentioned approaches in that they utilize deep learning technique (i.e., CNN) to automatically generate discriminative features from source code, rather than manually designing features which can capture semantic and structural information of programs. Their features lead to more accurate predictions. The state-of-the-art cross-project defect prediction is proposed by Nam et al. [NPK13], who adopted a state-of-the-art transfer learning technique called Transfer Component Analysis (TCA). They further improved TCA as TCA+ by optimizing TCA's normalization process. They evaluated TCA+ on eight open-source projects, and the results show that their approach significantly improves cross-project defect prediction. Xia et al. [XLWY16] proposed HYDRA, which leverages a genetic algorithm and ensemble learning (EL) to improve cross-project defect prediction. HYDRA requires massive training data and a portion (5%) of labeled data from test data to build and train the prediction models. TCA+ [NPK13] and HYDRA [XLWY16] are the two state-of-the-art techniques for cross-project defect prediction. However, in Wang et al.'s work [WLNT18], they only use TCA+ as baseline for cross-project defect prediction. This is because HYDRA requires that the developers manually inspect and label 5% of the test data, while in real-world practice, it is very expensive to obtain labeled data from software projects, which requires the developers' manually inspection, and the ground truth might not be guaranteed. Most of the above existing cross-project approaches are examined for file-level defect prediction only. Recently, Kamei et al. [KFM$^+$16] empirically studied the feasibility of change level defect prediction in a cross-project context. Wang et al. [WLNT18] examines the performance of Deep Belief Network (DBN)-based semantic features on change-level cross-project defect prediction tasks. The main differences between this and existing approaches for within-project defect prediction and cross-project defect prediction are as follows. First, existing approaches to defect prediction are based on manually encoded traditional features which are not sensitive to the programs' semantic information, while Wang et al.'s approach automatically learns the semantic features using a DBN and uses these features to perform defect prediction tasks. Second, since Wang et al.'s method requires only the source code of the training and test projects, it is suitable for both within-project and cross-project defect prediction. The semantic features can capture the common characteristics of defects, which implies that the semantic features trained from one project can be used to predict a different project, and thus is applicable in cross-project defect prediction. Deep learning-based approaches require only the source code of the training and test projects, and are therefore suitable for both within-project and cross-project defect prediction. In the next session, we explain, based on recent research, how effective and accurate fault-prediction models developed using deep learning techniques are.

Deep Learning in Software Fault Prediction: Recently, deep learning algorithms have been adopted to improve research tasks in software engineering. The most popular deep learning techniques are: Deep Belief Networks (DBN), Recurrent Neural Networks, Convolutional Neural Networks and Long Short Term Memory (LSTM), see Table 4.11. Yang et al. [YLX$^+$15] propose an approach that leverages deep learning to generate new features from existing ones and then use these new features to build defect prediction models. Their work was motivated by the weaknesses of logistic regression (LR), which is that LR cannot combine features to generate new features. They used a Deep Belief Network (DBN) to generate features from 14 traditional change level features, including the following: number of modified subsystems, modified directories, modified files, code added, code deleted, lines of code before/after the change, files before and after the change, and several features related to developers' experience [YLX$^+$15]. The work of Wang et al. [WLNT18] differs from the above study mainly in three aspects. First, they use a DBN to learn semantic features directly from source code, while Yang et al. use relations among existing features. Since the existing features cannot distinguish between many semantic code differences, the combination of these features would still fail to capture semantic code differences. For example, if two changes add the same line at different locations in the same file, the traditional features cannot distinguish between the two changes. Thus, the generated new fea-

Table 4.11: Common machine learning and deep learning techniques used in software defect prediction

Techniques	Definition	Advantages	Drawbacks	Ref.
RNN	RNNs are called recurrent because they perform the same task for every element of a sequence, with the output being depended on the previous computations.	- Possibility of processing input of any length - Model size not increasing with size of the input - Computation takes into account historical information	- Slow computation - Difficulty of accessing information from a long time ago - Cannot consider any future input for the current state	[WZ18]
LSTM	A long short-term memory (LSTM) network is a type of RNN model that avoids the vanishing gradient problem by adding 'forget' gates.	- Remembering information for a long periods of time	- It takes longer to train - It requires more memory to train	[DPN+19], [DPN+18]
CNN	CNN is a class of deep neural network, it uses convolution in place of general matrix multiplication in at least one of their layers.	- It automatically detects the important features without any human supervision.	- need a lot of training data. - High computational cost.	[LHZL17], [MLZ+16], [PNB18]
Stacked Auto-Encoder	A stacked autoencoder is a neural network consist several layers of sparse autoencoders where output of each hidden layer is connected to the input of the successive hidden layer.	- Possible use of pre-trained layers from another model, to apply transfer learning - It does not require labeled inputs to enable learning	- Computationally expensive to train - Extremely uninterpretable - The underlying math is more complicated - Prone to overfitting, though this can be mitigated via regularization	[MF19], [TLW17]
DBN	DBN is an unsupervised probabilistic deep learning algorithm.	- Only needs a small labeled dataset - It is a solution to the vanishing gradient problem	- It overlooks the structural information of programs	[WLT16]
Logistic Regression	LR is used to describe data and to explain the relationship between one dependent binary variable and independent variables.	- Easy to implement - Very efficient to train	- It cannot combine different features to generate new features. - It performs well only when input features and output labels are in linear relation	[KSA+13]
SVM	SVM is a supervised learning model. It can be used for both regression and classification tasks.	- Using different kernel function it gives better prediction result - Less computation power	- Not suitable for large number of software metrics	[EE08]
Decision Tree	DT is a decision support tool that uses a tree-like graph or model of decisions and their possible consequences.	Tree based methods empower predictive models with high accuracy, stability and ease of interpretation.	- Construction of decision tree is complex	[GSR10], [KS02], [WSC]

tures, which are combinations of the traditional features, would also fail to distinguish between the two changes.

How to explain deep learning results is still a challenging question in the AI community. To interpret deep learning models, Andrej et al. [KJL15] used character level language models as an interpretable testbed to explain the representations and predictions of a Recurrent Neural Network (RNN). Their qualitative visualization experiments demonstrate that RNN models could learn powerful and often interpretable long-range interactions from real-world data. Radford et al. [RJS17] focus on understanding the properties of representations learned by byte-level recurrent language models for sentiment analysis. Their work reveals that there exists a sentiment unit in the well-trained RNNs (for sentiment analysis) that has a direct influence on the generative process of the model. Specifically, simply fixing its value to be positive or negative can generate samples with the corresponding positive or negative sentiment. The above studies show that to some extent deep learning models are interpretable. However, these two studies focused on interpreting RNNs on text analysis. Wang et al. [WLNT18] leverages a different deep learning model, Deep Belief Networks (DBN), to analyze the ASTs of source code. The DBN adopts different architectures and learning processes from RNNs. For example, an RNN (e.g., LSTM) can, in principle, use its memory cells to remember long-range information that can be used to interpret data it is currently processing, while a DBN does not have such memory cells. Thus, it is unknown whether DBN models share the same properties (w.r.t interpretability) as RNNs. Many studies used a topic model [BNJ03] to extract semantic features for different tasks in software engineering ([CTNH12], [NNP11], [XZYW12]). Nguyen et al. [NNP11] leveraged a topic model to generate features from source code for within-project defect prediction. However, their topic model handles each source file as an unordered token sequence.

Thus, the generated features cannot capture structural information in a source file. A just-in-time defect prediction technique was proposed by Kamei et al. which leverages the advantages of Logistic Regression (LR) [KSA+13]. However, logistic regression has two weaknesses. First, in logistic regression, the contribution of each feature is calculated independently, which means that LR cannot combine different features to generate new ones. For example, given two features x and y, if $x \times y$ is a highly relevant feature, it is not enough to input only x and y because logistic regression cannot generate the new feature $x \times y$. Second, logistic regression performs well only when input features and output labels are in linear relation. Due to these two weaknesses, the selection of input features becomes crucial when using logistic regression. The bad selection of features may result in a non-linear relation for output labels, leading to bad training performance or even training failure. This severe problem leads some studies to adopt Deep Belief Network (DBN), which is one of the state-of-the-art deep learning approaches. The biggest advantage of DBN, as shown in Table 4.11, over logistic regression is that DBNs can generate a more expressive feature set from the initial feature set. We summarizes in Table 4.11 the most commonly used machine learning and deep learning techniques in software defect prediction.

5. Learning to Rank Test Case Prioritization

In Continuous Integration (CI) environments, the productivity of software engineers depends strongly on the ability to reduce the round-trip time between code commits and feedback on failed test cases. Test case prioritization is popularly used as an optimization mechanism for ranking tests by their likelihood in revealing failures. However, existing techniques are usually time and resource intensive making them not suitable to be applied within CI cycles. This chapter formulates the test prioritization problem as an online learn-to-rank model based on foundational results of learning-to-rank from the field of Information Retrieval (IR), as defined by Li [Li14] and described in Figure 5.1, and applying reinforcement learning to make the ranking approach adaptive and suitable to our CI context. Our approach minimizes the testing overhead and continuously adapts to the changing environment as new code and new test cases are added in each CI cycle.

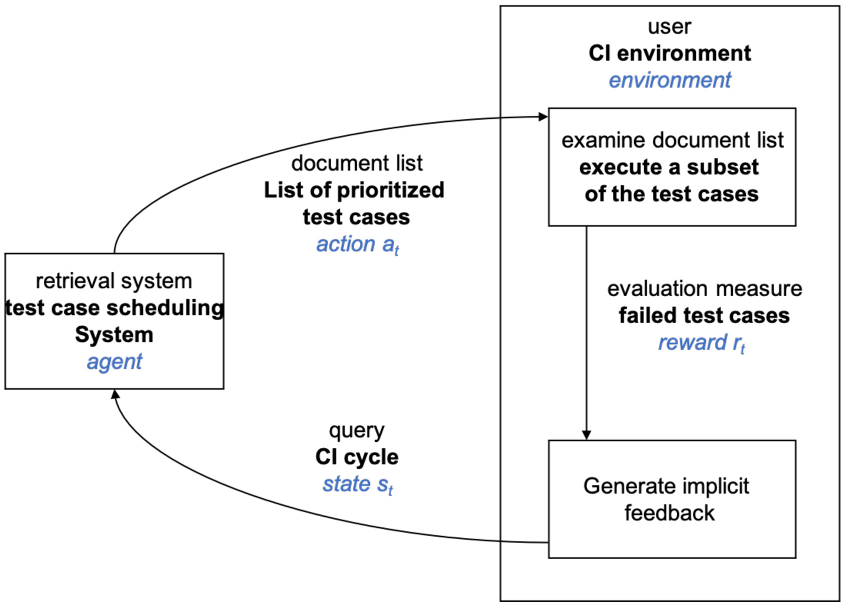


Figure 5.1: The test case prioritization problem modeled as a contextual bandit problem with IR terminology in black, test case prioritization in bold and RL terminology in blue italics.

5.1 Motivation

Continuous Integration (CI) environments automate the process of building and testing software where software engineers integrate their code changes at frequent time intervals with the mainline code base. CI reduces integration problems and shortens release time significantly. However, CI environments are facing scalability challenges; Amazon is conducting $136,000$ builds per day [LER18], and Google engineers wait 45 minutes to 9 hours to receive testing results [MGN+17]. These facts introduce new challenges for the testing activities due to (i) the dynamic environment resulting from frequent code changes and (ii) time constraints since testing should be fast enough to enable frequent builds and testing of the source code.

In order to reduce the cost of testing, different test optimization techniques such as test prioritization and selection have been developed. Test selection aims at reducing testing to the set of tests affected by code changes. However, test selection techniques often underestimate or overestimate the needed set of tests making such techniques not attractive because they omit test cases that could have exposed faults if they were executed [KP02]. Test prioritization aims to reorder the test cases to maximize early fault detection and can be used for test selection in resource constrained environments by running the top k ranked tests.

In CI environments, traditional test prioritization techniques can be difficult to apply. Most traditional techniques require gathering code coverage data or performing static analysis which makes such techniques (i) time intensive (e.g., not applicable within CI cycles) and (ii) limited from a practicality point of view as modern software is often written in different programming languages.

To address such challenges, existing approaches have to be improved to deal with the dynamic nature and timing constraints of CI and the heterogeneity of modern software applications. Applicable test prioritization techniques must be fast enough to avoid delays in the quick build cycles that justifies CI.

Recent test prioritization approaches shifted to using historical data about failures and successes of test cases based on the hypothesis that test cases that have failed in the past are more likely to fail in the future [KP02, MGS13]. Such techniques are based on machine learning techniques which do not allow incremental learning but regularly construct new models from scratch making them (i) time consuming, (ii) leading to potentially outdated models, and (iii) not well adapted to changes in the execution environment. In fact, it is frequent to remove test cases from some CI cycles as they test obsolete features, or to add new test cases to test new features. Moreover, some test cases might be important at some point of time because they test features important for the customers and later loose their prevalence because of a testing focus shift. To summarize, non-adaptive techniques might miss changes in the importance of test cases over others because they are based on systematic prioritization approaches.

In order to address these challenges, we propose a new test case prioritization approach in CI based on foundational results of learning to rank from the field of information retrieval (IR) [SMPB11] and reinforcement learning. In particular, we use historical test results and information about each code change to learn a ranking model used to predict rankings for all test cases. The model is based on reinforcement learning principles allowing us to design an adaptive method capable of learning from the execution environment. Adaptiveness means in our context that our approach can progressively improve its effectiveness after each test case's execution cycle. Unlike other test prioritization approaches, our technique is able to adapt to situations where test cases are added or deleted, or when testing priorities change because of changing failure indications in different code regions as the code matures or as the requirements change. Moreover, our technique does not require computationally intensive operations during the ranking process. It uses knowledge about the execution history of the test cases at each CI cycle and updates this knowledge from feedback provided by a reward function.

The contributions of this chapter are:

1. Formulation of test prioritization as an online learning-to-rank problem based on reinforcement learning principles.
2. Implementation of an online prioritization model without any previous training phase into the continuous integration process showing that our approach can learn to prioritize test cases better than traditional techniques after only around 150 CI cycles (which corresponds to around five month of data if there is only one CI cycle per day).
3. An empirical evaluation of our approach on an industrial data set from a web-based asset management software applications gathered from more than three years of continuous integration. This shows that our technique is applicable in real-world industrial settings.

5.2 Approach

In this section, we introduce our approach for the test case prioritization problem using an online learning-to-rank method based on reinforcement learning, called ***Lea**rning-to-**R**a**n**k **Te**st **C**ases* (LeaRnTeC).

5.2.1 Problem Formulation

We formulate the test prioritization problem as an online learn-to-rank model using reinforcement learning techniques. The following problem formulation for test prioritization is based on foundational results of learning-to-rank from the field of Information Retrieval (IR) as defined by Li [Li14], and applying reinforcement learning to make the ranking approach adaptive and suitable to our CI context. We first introduce necessary notations used in the rest of the chapter and then present the addressed problem in a formal way. Let T be a set of test cases $\{t_1, t_2, ..., t_N\}$ to be prioritized at a CI cycle, and $\mathcal{C} = \{c_1, c_2, ..., c_M\}$ a set of CI cycles. In IR, $\mathcal{C}$ corresponds to queries, and T corresponds to documents. For each CI cycle $c \in \mathcal{C}$ the task is to rank the test cases in T using information about c and T. The ranking is computed by a scoring function $f(c,t) : \mathcal{C} \times T \rightarrow \Re$. Scores are indicators of the failure likelihood of a test t for a given CI c. In our context of test cases prioritization within a CI cycle, test cases that are likely to fail based on previous executions might not fail when the software is bug-free, even if their failure would be predicted. Therefore, we need to formulate our learning-to-rank model with an online-learning approach able to adapt and tolerate such test cases continuously. Our formulation of the test cases prioritization differs from most other works in that we consider the continuous adaptation of the priority scores during the software lifetime as new code changes are added and CI cycles are executed. A natural fit for our problem are formalizations from reinforcement learning, where an algorithm learns by trying out actions (e.g., test suites) that generate reward (e.g., evaluation measures such as test recall) from its environment (e.g., CI test execution environment). Figure 5.1 shows the CI cycle. A developer commits code changes that trigger a CI cycle, the test case scheduling system generates a prioritized test suite. A subset of the test cases will be executed in the given order until a time-out is reached (it is usually not feasible to execute all available test cases within a CI cycle). As the test cases are executed, some will fail and others will not. The goal of our prioritization/ranking algorithm is to learn an optimal ordered sequence of test cases that reveal failures as early as possible. Our algorithm uses the information about which test cases failed to infer feedback about the quality of the learned scores. This problem formulation translates to an RL problem in which our algorithm based on test cases execution results tries to maximize a hidden reward that corresponds to some evaluation measures. We assume that CI cycles are independent, i.e., CI cycles are the result of code changes committed by different developers. This renders the overall problem to a well-studied type of RL, which is the contextual bandit problem [Lan08].

Our online prioritization algorithm can observe feedback only on the list of test cases that have been executed. Therefore, effective learning is possible only if our algorithm experiments with new prioritization ranks. Hence, our algorithm should balance exploration and exploitation to improve the prioritization while learning. As is common in RL, we measure the cumulative reward, i.e., the sum of rewards over all CI cycles during learning. We assume an infinite horizon problem that includes a discount factor $\gamma \in [0, 1)$ that gives more weight to immediate rewards than future rewards. The intuition behind the discount factor is to suppose that there is a $1 - \gamma$ probability that the CI cycle will terminate at each timestep. Consequently, rewards are weighted according to the probability that the CI cycle will last long enough. The cumulative reward is defined as a discounted infinite sum of rewards r_i: $\mathcal{R} = \sum_{i=1}^{\infty} \gamma^{i-1} r_i$.

5.2.2 Baseline Algorithm

Our approach is based on a gradient-based policy search algorithm called Dueling Bandit Gradient Descent (DBGD) [YJ09]. DBGD is suitable for online learning of test case ranks because it can generalize over CI cycles requiring only relative evaluations of the quality of two test cases lists (two test suites), and can infer such comparison from the execution of the two test suites. In order to rank a set of test cases T given a CI cycle c, feature vectors $\Phi = \left\{\phi_1, \phi_2, ..., \phi_{|T|}\right\}$ that describe the relationship between T and c are produced. The ranking scores for each test case t are produced using $f(c, t) = w\Phi$.

Algorithm 1 summarizes our online learning approach. It requires as input a comparison method $\Delta(\mathcal{TS}_1, \mathcal{TS}_2)$ that compares two test suites, two step size parameters α and β, and an initial weight vector w_0. At each CI cycle c_i we generate ranked test suites: one exploitative and one exploratory. The exploitative test suite is generated from the exploitative weight vector w_i, performing best up to CI cycle c_i. The exploratory test suite is generated from the exploratory weight vector w'_i by moving w_i in a random direction u_i with a step size β. Both exploitative and exploratory test suites are compared using $\Delta(\mathcal{TS}_1, \mathcal{TS}_2)$. The exploitative weight vector w_i is updated by moving it towards w'_i by a step size α if w'_i is judged to have produced a better test cases ranking.

Algorithm 1 Test Ranks Learning based on DBGD [YJ09]

```
1:  Input:
2:  $\Delta(\mathcal{TS}_1, \mathcal{TS}_2), \alpha, \beta, w_0$
3:  for CI cycle $c_i (i = 1..I)$ do
4:      Sample unit vector $u_i$ uniformly.
5:      $w'_i \leftarrow w_i + \beta u_i$ // generate exploratory $w$
6:      if $\Delta(\mathcal{TS}(w_i), \mathcal{TS}(w'_i))$ then
7:          $w_{i+1} \leftarrow w_i + \alpha u_i$ // update exploitative $w$
8:      else
9:          $w_{i+1} \leftarrow w_i$
10:     end if
11: end for
12: Output
13: $w_{i+1}$
```

5.2.3 Comparison Procedure

We present in Algorithm 2 a comparison function $\Delta(\mathcal{TS}_1, \mathcal{TS}_2)$ that balances exploration and exploitation during the online learning process of test cases ranks. We use a method inspired by ϵ-greedy, where an RL agent explores actions with probability ϵ and selects greedy action with probability $1 - \epsilon$.

Our algorithm takes as input two test suites $\mathcal{TS}_1$ and $\mathcal{TS}_2$ and an exploration rate ϵ. For each rank of the result test suite to be filled, we pick of the two result lists biased by the exploration rate ϵ. From the selected list (test suite), the highest rank test case that is not yet in the combined result list is added at that rank. The result test suite is then sent to the CI environment for execution. Each test failure is attributed to that test suite if the test case is in the top N of the test suite, where N is the total number of failures detected after execution. The exploration rate $\epsilon \in [0.0, 0.5]$ determines the probability of selecting a test suite to contribute to the result list at each rank. As ϵ decreases, the exploitative list contributes more test cases, which introduces bias to re-executing already failed test cases. We partially compensate for this bias since $E[tf_2] = \frac{n_1}{n_2} * E[tf_1]$, where $E[tf_i]$ is the expected number of test failures within the top N of test suite TS_i, and n_i is the number of test cases from TS_i that were selected in the top N of the result list. This procedure compensates for the expected number of test failures but leaves some bias in the expected number of times each test suite is selected. One solution to perfectly compensate for such bias is to make probabilistic updates, but this would introduce additional noise creating a bias/variance trade-off.

Algorithm 2 Comparison Function $\Delta(\mathcal{TS}_1, \mathcal{TS}_2)$

```
1:  **Input:**
2:  $\mathcal{TS}_1, \mathcal{TS}_2, \epsilon$
3:  initialize empty result list $R$
4:  **for** rank $r$ in $(1.. \mid \mathcal{TS}_1 \mid)$ **do**
5:      $TS \leftarrow \mathcal{TS}_1$ with probability $\epsilon$, $\mathcal{TS}_2$ with probability $1 - \epsilon$
6:      $R[r] \leftarrow$ first element of $TS \notin R$
7:  **end for**
8:  execute the test cases in $R$ and log the failed test cases $TF$
9:  $N = \mid TF \mid$
10: $tf_1 = tf_2 = 0$
11: **for** $i$ in $(1..N)$ **do**
12:     **if** $TF[i] \in \mathcal{TS}_1[1:N]$ **then**
13:         $tf_1 = tf_1 + 1$
14:     **end if**
15:     **if** $TF[i] \in \mathcal{TS}_2[1:N]$ **then**
16:         $tf_2 = tf_2 + 1$
17:     **end if**
18: **end for**
19: $n_1 = \mid \mathcal{TS}_1[1:N] \mid \cap R[1:N]$
20: $n_2 = \mid \mathcal{TS}_2[1:N] \mid \cap R[1:N]$
21: $tf_2 = \frac{n_1}{n_2} * tf_1$
22: **Output**
23: $tf_1 < tf_2$
```

5.2.4 Integration within a CI Process

We integrated our prioritization algorithm into our CI environment, which is based on Gitlab CI/CD as a build step that extends our current CI pipeline, as illustrated in Figure 5.2

5.3 Experimental Evaluation

In this section, we present an experimental evaluation of our method to address the following two research questions:

1. **RQ1:** How efficient can our approach conduct priority-based test selection?
2. **RQ2:** How effective can our approach conduct test prioritization?

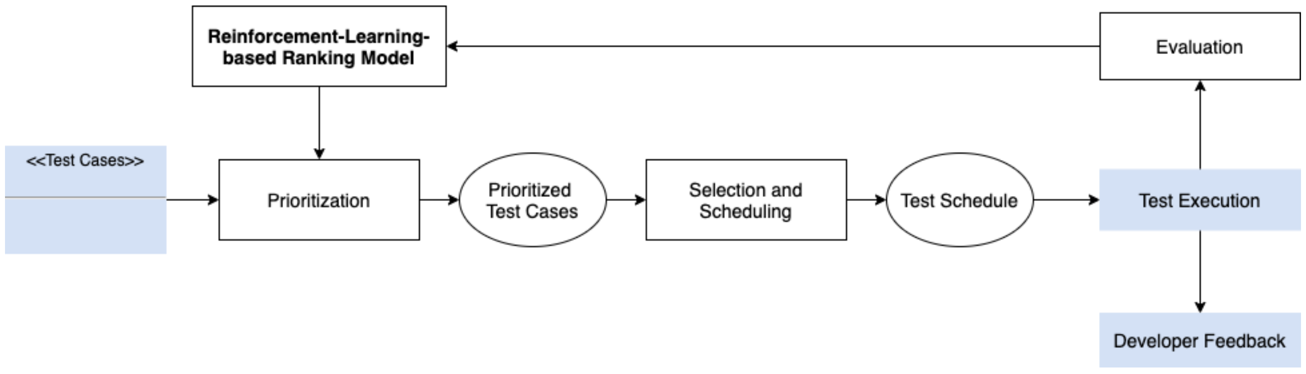


Figure 5.2: Testing in CI process (adopted from [SGMM17]): LeaRnTec uses test execution results for learning-to-rank test case prioritization (solid boxes: Included in LeaRnTec, blue colored boxes: Interfaces to the CI environment).

We first define and give an overview of the evaluation metrics used in this work in Section 5.3.1. Afterward, we introduce the experimental setup in Section 5.3.2. Then, we discuss the results of our experiments based on the research questions in Section 5.3.3. Finally, a discussion of possible threats is given in Section 5.3.4.

5.3.1 Evaluation Metrics

The presented evaluation metrics in this section are described in detail in Chapter 2. Our approach learns priority ranks for all test cases. Our approach, when applied in the CI context, might terminate before all test cases are executed. This behavior is similar to the behavior of test selection techniques. We used the following standard commonly used metrics in software test selection to assess the effectiveness of our approach when not all test cases are executed [PP21, BX16, MSPC19]:
Test recall: Intuitively, the test recall approximates the empirical probability of a particular test selection strategy catching an individual failure. The formula is described in Chapter 2, Section 2.3.2.

For the evaluation of the effectiveness of our test case prioritization approach, we use the standard evaluation metric:
Average Percentage of Faults Detected (APFD): APFD was introduced in [RUCH99] to measure the effectiveness of test case prioritization techniques. It measures the quality via the ranks of failure-detecting test cases in the test execution order. It ranges from 0 to 1, with higher numbers implying faster fault detection. The formula is described in Chapter 2, Section 2.3.2.

Furthermore, we used an extension of the APFD metric:
Normalized Average Percentage of Faults Detected (NAPFD): NAPFD is the ratio between detected and detectable failures within the test suite $\mathcal{TS}$. The NAPFD of a prioritized test suite $\mathcal{TS}'$ is calculated using the formula which is described in Chapter 2, Section 2.3.2.

5.3.2 Experimental Setup

To evaluate the efficiency and the effectiveness of our method, we compare it against four test case prioritization methods:

1. Elbaum's Approach [ERP14]: we were able to re-implement the approach since a detailed algorithm is available,
2. RETECS (Reinforced Test Case Selection) [SGMM17]: the first online learning approach, to the best of our knowledge, using reinforcement learning in the context of test prioritization and selection in continuous integration. RETECS considers as input the test case

duration, historical failure data, and previous last execution. We execute the implementation of RETECS available in the literature[1] by using ANN, which obtained the best results compared to with a Tableau representation [SGMM17].

3. COLEMAN (Combinatorial VOlatiLE Multi-Armed BANdit) [PLV20]: we compare the results (we have the results of the two evaluation metrics APFD and NAPFD but not the Test Recall metric [PLV20]) on the same projects since they are available, but neither the implementation of the approach is available, nor the algorithm to re-implement it.

4. Random: we use random test case prioritization as a baseline method.

To account for the influence of randomness within the experimental evaluation, all experiments are repeated 30 times and reported results show the mean, if not stated otherwise. Our approach is implemented in Python and scikit-learn and the source code, as well as the used data, is available for peer review[2]. All the experiments are performed on an Intel ® Xeon ® $E5-2640$ v3 with 2.60 GHz CPU, 94GB RAM, running Linux Ubuntu 18.04.1 LTS.

5.3.2.1 Algorithm Configuration

In all experiments, we initialize the starting weight vector w_0 randomly and use the parameter settings suggested in [YJ09]: $\beta = 1$ and $\alpha = 0.01$. In order to fix the exploration rate ϵ of our algorithm, we took 1000 CI cycles from our industrial asset management example described in Section 5.3.2.2. As an evaluation metric of the performance of our approach, we used the test recall metric defined in Equation 2.17.

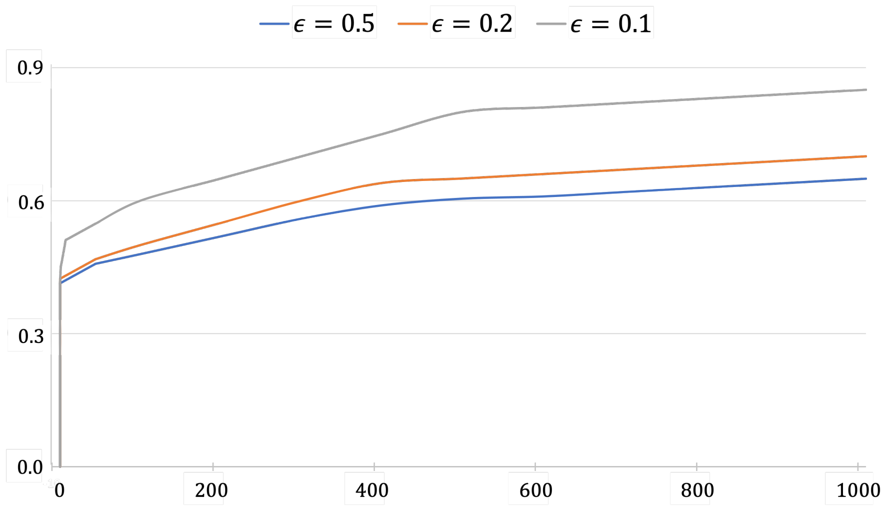


Figure 5.3: Testing Record Performance over CI Cycles for $\epsilon \in \{0.1, 0.2, 0.5\}$.

The results are depicted in Figure 5.3 and show that $\epsilon = 0.1$ would be a good exploration rate setting balancing a learning process between exploiting historical test cases execution and exploring in 10% of the times new test cases. Figure 5.3 also shows that our approach is capable to learn a test case prioritization scheme with a test recall of over 60% after only learning from 150 CI cycles. We set the discount factor $\gamma = 0.995$. This choice can be justified in two ways: (i) it is a typically used discount factor when evaluating RL methods [SB18]; a value close to 1 means future rewards have significant weight, and consequently, the algorithm must explore to perform well. (ii) with $\gamma = 0.995$ a cumulative performance can be accurately estimated with

[1]Implementation available at `https://bitbucket.org/helges/RETECS`
[2]Implementation available at `https://github.com/so1188/test-case-scheduler`

the number of CI cycles in our data sets. Rewards after 1000 iterations have a weight of 1% of less, consequently, our finite runs are good approximations of true cumulative performance.

In all our experiments in the following sections, when comparing our approach to the baseline approaches, we repeat all runs 30 times and report results averaged over folds. We test for significant differences with the baseline approaches using a two-sided student's t-test with ($p < 0.05$).

5.3.2.2 Studied Data Sets

We used publicly available industrial data sets from ABB Robotics Norway, used by Spieker et al. [SGMM17]. The data sets contain CI logs about test executions of two industrial robots: "Paint Control" and "IOF/ROL". We also used the Google Shared Data set of Test Suite Results (GSDTSR) presented firstly by Elbaum et al. [ERP14]. Moreover, we consider eight other systems already used in the literature and they were used to evaluate COLEMAN [PLV20]. The data sets are detailed in Table 5.1. The data of the experiment is available online [3]. The data sets include test verdicts, duration of test execution, and their corresponding CI cycles.

Table 5.1: Data Sets Overview

Data Set	CI Cycles	Test Cases	Failures
GSDTSR	259388	5555	3208
Paint Control	20711	1980	4956
IOF/ROL	2392	1941	9289
Druid	286	2391	270
DSpace	6309	211	13413
Deeplearning4j	3410	117	777
Retrofit	3719	206	611
Guava	2011	568	7659
ZXing	961	124	68
Fastjson	2710	2416	940
OkHttp	9919	289	9586
QioTec Asset	10829	36173	8593

We extend our validation on a web-based industrial asset management application (QioTec Asset) where we used three years of historical CI execution logs to validate our approach further. Table 5.1 gives an overview of the data sets' structure, where all columns show the total amount of data in the data set.

5.3.3 Results and Analysis

To quantify the accuracy of test selection and test prioritization, we measure recall at various cut-off points, recall at a various percentages of the required time, APFD, and NAPFD, respectively. The approaches under comparison are our approach denoted as "LeaRnTeC" and previous approaches, defined above.

5.3.3.1 RQ1: Priority-based test case selection efficiency.

We first evaluated our approach on the 11 publicly available data sets [3]. The results are summarized in Table 5.2. Recall is measured as the percentage of test failures detected if tests ranked before the cut-off point are executed, as defined in Section 5.3.1. The experiments show that

[3]Datasets available at `https://github.com/so1188/data`

our approach can reach a recall of 0.95 after only executing 13% to 35% of the total available test cases. For this reason, we extended to a real-world industrial application where we have access to all CI logs. We compare our approach against the following methods; Random, Elbaum's Approach and RETECS. We could not compare our approach against COLEMAN since the Test Recall metric was not considered by Lima et al. [PLV20] and therefore we don't have available results on the same studied projects. Moreover, neither the source code of COLEMAN [PLV20] is available nor the algorithm to re-implement it. Figure 5.4 shows average recall across all changes in the test datasets at varying cut-off points below 14000 ($\sim 40\%$) for all the approaches under comparison for the industrial asset management application. Each point on the curve is average recall across all changes in the test data sets at that cut-off point. Our approach has a higher average recall at all cut-off points than any other previous approach.

Furthermore, the data sets of the studied projects do not contain a time limit for each CI cycle. Therefore, a fixed percentage of 50% of the required time is used for the time limit for each CI cycle. We evaluated how this percentage affects the test recall (e.g., the percentage of detected test failures). We set the time limit of each CI cycle to 50% of the execution time of the overall test suite $\mathcal{TS}_{c_i}$. Figure 5.5 shows the results on the industrial application (QioTec Asset). The test recall result is averaged over all CI cycles to compare the performance of our approach against other methods (Random, Elbaum's Approach, and RETECS). In all the studied test case prioritization methods, the performance decreases with a lower time limit. Since RETECS is based on reinforcement learning agents, a decreased time of test case execution implies fewer test cases can be executed and, therefore, a limited data for learning the actions and their rewards, which leads to a slower learning process. Our approach shows a good performance under different time limits.

Table 5.2: Average Recall by Number of Tests for the Studied Projects

	Percentage of Detected Test Failures											
	Random			**Elbaum's Approach**			**RETECS**			**LeaRnTeC**		
	50%	75%	95%	50%	75%	95%	50%	75%	95%	50%	75%	95%
GSDTSR	1519 (27%)	2858 (∼ 51%)	3524 (∼ 63%)	944 (∼ 17%)	1791 (∼ 32%)	2062 (∼ 37%)	635 (∼ 11%)	1198 (∼ 21%)	1685 (∼ 30%)	**519 (∼ 9%)**	**908 (∼ 16%)**	**1277 (∼ 23%)**
IOF/ROL	405 (∼ 19%)	779 (∼ 37%)	1115 (∼ 53%)	233 (∼ 12%)	412 (∼ 21%)	549 (∼ 28%)	195 (∼ 9%)	338 (∼ 16%)	447 (∼ 21%)	**155 (∼ 7%)**	**238 (∼ 11%)**	**354 (∼ 17%)**
Paint Control	356 (∼ 18%)	634 (∼ 32%)	851 (∼ 43%)	283 (∼ 14%)	198 (∼ 10%)	345 (∼ 17%)	178 (∼ 9%)	217 (∼ 11%)	297 (∼ 15%)	**138 (∼ 7%)**	**198 (∼ 9%)**	**257 (∼ 13%)**
Druid	479 (∼ 20%)	932 (∼ 39%)	1865 (∼ 78%)	358 (∼ 15%)	669 (∼ 28%)	1338 (∼ 56%)	263 (∼ 11%)	549 (∼ 23%)	1148 (∼ 48%)	**239 (∼ 10%)**	**358 (∼ 15%)**	**549 (∼ 23%)**
DSpace	63 (∼ 32%)	120 (∼ 57%)	168 (∼ 81%)	48 (∼ 23%)	87 (∼ 42%)	143 (∼ 68%)	37 (∼ 18%)	78 (∼ 37%)	113 (∼ 54%)	**25 (∼ 12%)**	**65 (∼ 31%)**	**61 (∼ 29%)**
Deeplearning 4j	31 (∼ 27%)	51 (∼ 43%)	62 (∼ 52%)	32 (∼ 28%)	42 (∼ 36%)	54 (∼ 47%)	24 (∼ 21%)	43 (∼ 37%)	51 (∼ 43%)	**12 (∼ 11%)**	**28 (∼ 24%)**	**36 (∼ 31%)**
Retrofit	39 (∼ 19%)	81 (∼ 39%)	89 (∼ 43%)	31 (∼ 15%)	67 (∼ 33%)	81 (∼ 39%)	24 (∼ 12%)	63 (∼ 31%)	76 (∼ 37%)	**16 (∼ 8%)**	**56 (∼ 27%)**	**65 (∼ 32%)**
Guava	176 (∼ 31%)	312 (∼ 55%)	431 (∼ 76%)	153 (∼ 27%)	221 (∼ 39%)	357 (∼ 63%)	142 (∼ 25%)	221 (∼ 39%)	335 (∼ 59%)	**107 (∼ 19%)**	**142 (∼ 25%)**	**193 (∼ 34%)**
ZXing	28 (∼ 23%)	46 (∼ 37%)	58 (∼ 47%)	23 (∼ 19%)	34 (∼ 27%)	45 (∼ 36%)	21 (∼ 17%)	28 (∼ 23%)	38 (∼ 31%)	**13 (∼ 11%)**	**21 (∼ 17%)**	**34 (∼ 27%)**
Fastjson	441 (∼ 18%)	994 (∼ 41%)	1956 (∼ 81%)	314 (∼ 13%)	507 (∼ 21%)	1135 (∼ 47%)	265 (∼ 11%)	459 (∼ 19%)	1063 (∼ 44%)	**242 (∼ 10%)**	**338 (∼ 14%)**	**652 (∼ 27%)**
OkHttp	101 (∼ 35%)	147 (∼ 51%)	182 (∼ 63%)	89 (∼ 31%)	124 (∼ 43%)	162 (∼ 56%)	81 (∼ 28%)	106 (∼ 37%)	135 (∼ 47%)	**49 (∼ 17%)**	**61 (∼ 21%)**	**101 (∼ 35%)**
QioTec Asset	15573 (∼ 44%)	24961 (∼ 69%)	29214 (∼ 81%)	11937 (∼ 33%)	22065 (∼ 61%)	26406 (∼ 73%)	10321 (∼ 28%)	19763 (∼ 55%)	24235 (∼ 67%)	**5368 (∼ 15%)**	**11937 (∼ 33%)**	**14469 (∼ 40%)**

Table 5.2 summarizes how many tests need to be selected by each approach to achieve a given recall average for all studied projects. Our approach requires 5368 (15%) top-ranked tests for the Industrial Asset Management Application to detect 50% of the test failures, compared to at least 10321 (28%) tests when using the previous approaches. Our approach requires 11937 (33%) top-ranked tests to detect 75% of the test failures, compared to at least 19763 (55%) tests using previous approaches. To detect 95% of the test failures, ou method "LeaRnTeC" requires 14469 (40%) top-ranked tests, whether "RETECS" needs about 24235 (67%) tests.

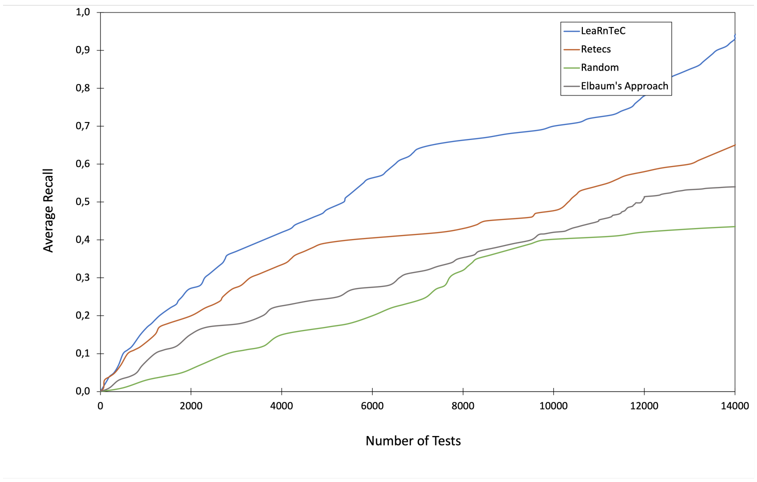


Figure 5.4: Average Recall for the Industrial Asset Management Application.

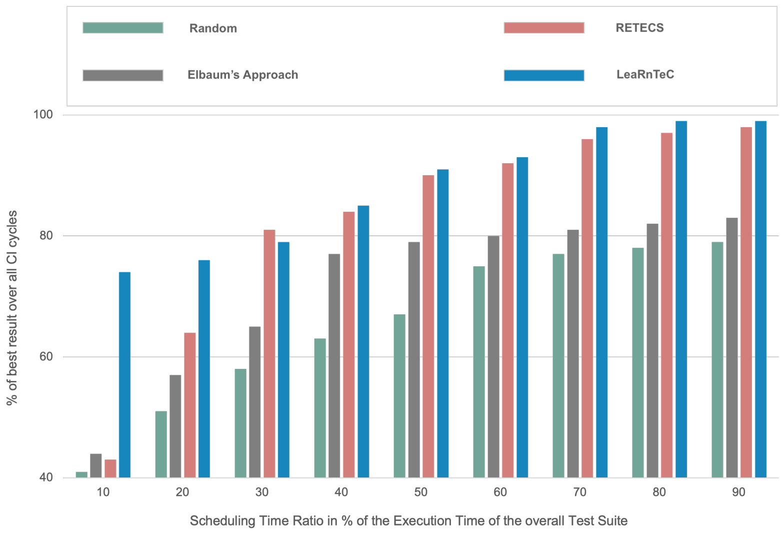


Figure 5.5: Relative Performance under Different Time Limits.

5.3.3.2 RQ2: Test case prioritization effectiveness.

We compare four test case prioritization methods with our method. We use the quality indicators: APFD and NAPFD to assess the effectiveness of our approach. We detail NAPFD and APFD results regarding the budget of 50% (Table 5.3). Based on Lima et al. [PLV20] (COLEMAN's approach), and Spieker et al. [SGMM17] (RETECS's approach), a time budget of 50% provides a constraint that allows for better comparison while retaining the inherent difficulty of the problem. The average is computed using results from 30 independent executions found by each approach in each of the studied projects. We highlighted the best values in bold. We compare our approach against the COLEMAN's approach [PLV20] since the results are available on the same studied projects by choosing the best-performing COLEMAN configuration with the Fitness-Rate-Rank (FRRMAB) policy, where the FRRMAB policy works with a sliding window. Elbaum et al. [ERP14] were among the first that used the sliding window principle in test case prioritization. We were able to re-implement Elbaum's approach since a detailed description of the algorithm is available. Therefore, we compare our approach against Elbaum's approach. Moreover, to compare our approach with RETECS, we choose the best-performing Network-based RL agent (with Test Case Failure reward). Table 5.3 shows the APFD and NAPFD results of three approaches compared with our approach on each of the twelve data sets. Our approach outperforms almost all studied approaches. We highlighted the best values in bold. For the industrial application (QioTec Asset), we could not compare our approach with COLEMAN [PLV20] because of the unavailability of the source code. However, we report the results as provided in [PLV20]. Among all the studied projects, the Random method has the worst performance. For IOF/ROL, Druid and ZXing, the approach had the worst performance. To understand this behavior, we analyze the failures detected over the cycles. In those systems, we verify peaks in the failure detection in a few commits and long periods without failures. Moreover, Druid has the lowest number of CI Cycles among the systems evaluated, and ZXing has the lowest number of failures reported.

Table 5.3: Mean and Standard Deviation APFD and NAPFD: Our Approach against RETECS, COLEMAN and Elbaum

	Average Percentage of Faults Detected (APFD)					**Normalized Average Percentage of Faults Detected (NAPFD)**				
	Elbaum's Approach	**RETECS**	**COLEMAN**	**Random**	**LeaRnTeC**	**Elbaum's Approach**	**RETECS**	**COLEMAN**	**Random**	**LeaRnTeC**
GSDTSR	0.9891 ± 0.000	0.9811 ± 0.000	0.9894 ± 0.000	0.4892 ± 0.000	**0.9896 ± 0.000**	0.9891 ± 0.000	0.9911 ± 0.000	0.9893 ± 0.000	0.4891 ± 0.000	**0.9894 ± 0.000**
Paint Control	0.9155 ± 0.000	0.9131 ± 0.000	0.9162 ± 0.000	0.4145 ± 0.000	**0.9162 ± 0.000**	0.9145 ± 0.000	**0.9150 ± 0.000**	**0.9150 ± 0.000**	0.4145 ± 0.000	**0.9150 ± 0.000**
IOF/ROL	0.4796 ± 0.002	0.5175 ± 0.008	0.5081 ± 0.002	0.4788 ± 0.002	**0.5223 ± 0.002**	0.4892 ± 0.002	0.5101 ± 0.007	0.5046 ± 0.002	0.4786 ± 0.002	**0.5189 ± 0.002**
Druid	0.4477 ± 0.110	0.4147 ± 0.102	**0.9486 ± 0.016**	0.4506 ± 0.014	0.6087 ± 0.009	0.4472 ± 0.110	0.4851 ± 0.134	**0.9333 ± 0.013**	0.4464 ± 0.014	0.6710 ± 0.008
DSpace	0.9713 ± 0.002	0.9683 ± 0.001	0.9737 ± 0.009	0.5587 ± 0.001	**0.9767 ± 0.009**	0.9592 ± 0.002	0.9568 ± 0.001	0.9724 ± 0.009	0.5581 ± 0.001	**0.9766 ± 0.008**
Deeplearning4j	0.8314 ± 0.004	0.8331 ± 0.041	0.8106 ± 0.001	0.6404 ± 0.016	**0.8334 ± 0.001**	0.8066 ± 0.003	0.7049 ± 0.070	0.7890 ± 0.001	0.6381 ± 0.011	**0.8200 ± 0.000**
Retrofit	0.9852 ± 0.001	0.9762 ± 0.002	0.9850 ± 0.000	0.4712 ± 0.001	**0.9885 ± 0.000**	0.9717 ± 0.001	0.9706 ± 0.002	0.9853 ± 0.000	0.4710 ± 0.002	**0.9893 ± 0.000**
Guava	0.9770 ± 0.006	0.9767 ± 0.008	0.9687 ± 0.003	0.3611 ± 0.002	**0.9872 ± 0.007**	0.9668 ± 0.006	0.9502 ± 0.015	0.9653 ± 0.004	0.3603 ± 0.002	**0.9675 ± 0.007**
ZXing	0.4876 ± 0.001	0.4954 ± 0.000	**0.9862 ± 0.000**	0.4893 ± 0.001	0.6969 ± 0.000	0.4873 ± 0.002	0.4878 ± 0.000	**0.9846 ± 0.000**	0.4892 ± 0.002	0.6897 ± 0.000
Fastjson	0.9291 ± 0.006	0.9326 ± 0.005	0.9186 ± 0.021	0.5193 ± 0.002	**0.9340 ± 0.027**	0.9101 ± 0.005	0.8714 ± 0.007	0.9174 ± 0.021	0.5176 ± 0.003	**0.9178 ± 0.028**
OkHttp	0.8994 ± 0.005	0.8878 ± 0.015	0.9177 ± 0.000	0.4549 ± 0.002	**0.9246 ± 0.000**	0.8941 ± 0.005	0.8812 ± 0.010	0.9192 ± 0.000	0.4513 ± 0.002	**0.9317 ± 0.000**
QioTec Asset	0.5767 ± 0.008	0.6891 ± 0.000	—	0.4193 ± 0.002	**0.9896 ± 0.000**	0.5567 ± 0.008	0.6861 ± 0.000	—	0.4193 ± 0.002	**0.9973 ± 0.000**

Summary:

We formulate the test case prioritization problem as an online learn-to-rank model using reinforcement learning techniques. Our approach minimizes the testing overhead and continuously adapts to the changing environment as new code and new test cases are added in each CI cycle. The experimental evaluation results show fast learning and adaptation of our approach. It shows a good performance under different time limits. We validated our approach on an industrial case study showing that over 95% of the test failures are still reported back to the software engineers while only 40% of the total available test cases are being executed.

5.3.4 Threats of Validity

Internal. The first threat to internal validity for LeaRnTec is the influence of random decisions on the results. To mitigate the threat, we repeated our experiments 30 times and reported averaged results, and we tested for significant differences with a two-sided student's t-test. A further threat is that our approach (LeaRnTeC) can be parameter sensitive, and a set of parameters appropriate for one problem environment may not work as well for another. In our experiments, the parameters initially chosen for different problems were not changed to allow for better comparison. In a real environment, these parameters can be adjusted to adapt the approach to the specific environment. Finally, the existence of faults within our implementation. We approached this threat by applying established components, such as scikit-learn, within our software.
External. Our evaluation is based on three industrial data sets, which is a limitation regarding the wide variety of CI environments and failure distributions. Two of these data sets are publicly available, but according to our knowledge, it has only been used in one publication [SGMM17]. From what we have analyzed, there are no further public data sets available, including the required data, especially test verdicts over time. This threat was addressed by executing our validation on a further real-world industrial application.

5.4 Related Work

Test Failure Prediction: Recently, test failure prediction have been adopted to identify test cases that are more likely to fail with the goal of reducing the overall testing effort. In a study carried in Facebook, Machalica et al. [MSPC19] considered the effects of flaky tests and used change-level, target-level, and other features to predict test case results. Anderson et al. [ASD15] presented an approach that predicts each test case as passing or failing before executing these tests. However, these approaches rely on computationally program analysis techniques, which are challenging to apply in the CI context and even more challenging in the context of polygot software applications where different programming languages are used (i.e., for each programming language usually a specific program analysis tool and technique is required). Pan and Pradel [PP21] proposed a supervised model for test suite failure prediction instead of single test case prediction. Their prediction model uses features of the code change, the test suite, and the development history. They adopt some of the features from test case failure prediction approaches. However, their approach relies on discriminating test cases from the test suites based on the learned classifier model, which in case of false positives can lead to missing whole test suites that might be fault revealing. Moreover, their approach is not optimized for the CI environment and needs to be retrained after changes in the execution environment. In contrast, our work could decide whether to run the test suite at all, and if this decision is positive, we optimize which test cases to run by predicting the probability of failure of each test case.

Ranking-based Test Case Prioritization: Some research has considered the test prioritization as a ranking problem via training machine learning models to provide the probability of failure of each test case. Recently, Bertolino et al. [BGM^{+}20] evaluated the effectiveness of several models: Random Forest (RF), Multiple Additive Regression Tree (MART), L-MART, RankBoost, RankNet, Coordinate ASCENT (CA) for test prioritization. They showed that a pairwaise ranking model was the most accurate, which is (MART). Previous researchers used Support Vector Machine rank [Joa02], such as Lachmann et al. [LSN+16], others, like Tonella et al. [TAS06] used Rankboost [FIS+03] to rank test cases.

These ranking-based approaches used machine learning techniques, assuming that before training, the full data is available, and thus incremental learning is not supported (i.e., integrating new data into already constructed models); however, new models are constructed from scratch [PBGB21]. Especially in CI environments, this can lead to potentially outdated models, which are very inefficient and very time-consuming.

In the context of Continuous Integration environment, machine learning techniques are often infeasible because prediction models need to adapt to new data quickly and continuously, taking into account changes in the system by each cycle and test suites. In recent years, researchers have incorporated reinforcement learning methods into their research to deal with this issues, discussed below.

Reinforcement Learning-based Test Case Prioritization: Several recent studies have used Reinforcement Learning (RL) for both test prioritization and selection, mostly in the context of Continuous Integration environments. In these studies, the main aim is to benefit from the capabilities of RL to integrate new data into already constructed models without retraining them from scratch and to adapt seamlessly to the dynamic nature of CI, where frequent changes in systems and test suites occur. Most of those studies create the reinforcement learning environment using CI logs and train an RL agent through interaction with the environment and rewards. This agent can then prioritize test cases. More details on the RL technique are discussed in Chapter 2. Most researchers that used RL to solve test prioritization problems mainly differ on the choice of the reward function and the ways the agents learn optimal policy to prioritize and select test cases.

Spieker et al. [SGMM17] were among the first that used RL in the context of test case prioritization and selection in continuous integration. They defined 3 reward functions; failure count reward, test case failure reward, and time-ranked reward, and prioritize test cases according to their execution time and previous execution and failure history. In contrast, Shi et al. [SXW20] defined one reward function, which was weighted based on the entire execution history. Based on Spieker et al. [SGMM17] work, Bagherzadeh et al. [BKB21] performed a comprehensive investigation of RL techniques by guiding the RL agent according to three different ranking models: pairwise, listwise, and pointwise ranking. In a very recent work, Bertolino et al. [BGM+20] examined the performance of ten machine learning algorithms, including three reinforcement learning (RL) algorithms, for test prioritization in continuous integration. An experimental analysis shows that Non-RL-based approaches are more affected by code changes, while the RL-based algorithms are more robust, in the context of test case prioritization. Similar to Spieker et al. [SGMM17], Bertolino et al.'s [BGM+20] application of RL is based on the pointwise ranking model. Their results show that their specific RL configuration is significantly less accurate than the best ranking algorithms based on supervised learning (e.g., MART). Furthermore, Bertolino et al. [BGM+20] evaluated different ML models as policy models. They considered both a multi-layer perceptron and random forest, and a shallow network for policy model. Rosenbauer et al. [RSM+20] used a a rule-based evolutionary machine learning method; XCS classifier system (XCS) [Wil95] as policy model. In situations where the number of test case features is relatively small, and the most important features are also known and intuitive, simple ML models (other than Deep Neural Networks (DNNs)) can be used to achieve better accuracy at a lower cost in terms of training data and computation time. Other researchers used a Multi-Armed Bandit (MAB) approach for the test prioritization problem in continuous integration environment. Lima et al. [LMVAa20] showed that they outperform reinforcement learning with an ANN policy model by evaluating their approach on eleven case studies. In another work, Lima et al. [LV20] a similar comparison was performed using a Genetic Algorithm (GA). According to their results, MAB approach can perform similarly to GA in terms of percent of faults detected, Root Mean Square Error (RMSE), and Prioritization Time in 90% of the cases. Our approach, however, continuously learns a rank for each test case, where a rank corresponds to the failure likelihood of a test for a given CI cycle. This allows executing test cases by descending priority until a CI time-limit is reached or all test cases are executed. Moreover, our approach considers the test cases volatility.

6. Learning to Schedule Test Cases across Software Components based on Testers' Domain Knowledge

Testing in Continuous Integration (CI) involves test case prioritization and execution at each cycle with the goal to accelerate detecting faults and the release of new features. However, existing prioritization techniques are usually time and resource intensive to be applied within CI cycles. Moreover, existing techniques do not allow to capture algorithmically the testers' domain knowledge and requirements which are essential to speedup and control the expensive testing and release process. This chapter formulates the test prioritization problem as a sequential decision-making process using reinforcement learning. Our approach uses graph neural networks to represent the states of our reinforcement learning problem and capture the testers' domain knowledge as well as their requirements in the form of a probabilistic graph annotated with the preferences where to focus testing. In a constantly changing environment as new test cases are added and release requirements are changing, our approach continuously learns to prioritize error prone test cases.

6.1 Motivation

Modern software evolves constantly as developers change the source code by adding new features, refactoring existing code or fixing bugs. Continuous Integration (CI) is a common practice to allow developers to integrate their work into the mainline code base and regression testing is a crucial technique to ensure that the changes do not introduce new bugs at each CI cycle.

By default, regression testing executes all test cases including the new added test cases as well as all previously executed test cases to ensure that new changes do not break existing functionalities. However, the huge volume of test cases and code changes make regression testing very expensive in terms of both time and resources. Intuitively, prioritizing test cases that might reveal bugs would give the software engineers more time to fix the bugs and speed up the software release. In this context, test case prioritization techniques have been proposed [EMR01, LHH07, MHZ$^+$12, MB16, MB17, RUCH99, SZKP15, TAS06, YH12]. Test prioritization techniques reorder test cases to explore software faults earlier and have been widely adopted in the industry as by Salesforce.com [BX16] or Microsoft [CDD11].

However, the existing test prioritization techniques have some limitations; (i) they are time and resource intensive to be executed with CI cycles in an online and dynamic fashion, (ii) do not allow to capture algorithmically the testers domain knowledge and preferences, (iii) ignore the modular architecture of software applications and the relationships between its components.

In fact, a seasoned tester might have knowledge about which components are more error-prone. Moreover, the testing activities might sometimes require to prioritize some components over others because of release requirements (e.g., a particular feature or functionality is expected to be more used than other features).

In order to address these challenges, we propose a new lightweight test case prioritization approach in CI environments that captures testers requirements and domain knowledge and learns to schedule test cases across the software components while considering the structure of the software application. In particular, we formulate the test prioritization problem as a sequential decision-making process using reinforcement learning. We use graph neural networks to represent the states of our reinforcement learning problem and capture the structure of the software application as well the domain knowledge and requirements of the testers. The model is based on reinforcement learning principles allowing us to design an adaptive method capable of learning from the execution environment. Adaptiveness means in our context, that our approach can progressively improve its efficiency after each test case's execution cycle. Unlike other test prioritization approaches, our technique is able to adapt to situations where test cases are added or deleted or when testing priorities change because of changing failure indications in different code regions as the code matures or as the requirements change. Moreover, our technique does not require computationally intensive operations during the prioritization process. The contributions of this chapter are:

1. Formulation of test prioritization as an online sequential decision-making process based on reinforcement learning principles and using GNNs to capture the structure of the software applications as well as the domain knowledge and requirements of the testers
2. Implementation of an online prioritization model without previous training phase into the continuous integration process showing that our approach can learn to prioritize test cases better than traditional techniques after around 1000 CI cycles (which corresponds to around 2 month of data if there are only 2 CI cycles per day), described in the Validation Chapter 7.
3. A theoretical cost model that describes when our prioritization and selection approach is worthwhile
4. An empirical evaluation of our approach on 2 industrial data sets from 2 software applications gathered over a range from 1 to 3 years of continuous integration. This shows that our technique is applicable in real-world industrial settings, described in the Validation Chapter 7.

6.2 Problem Definition

Assume we have a software application composed of a set of components as described in Figure 1.1 in Chapter 1. The software components are developed in different programming languages (e.g Python for the data engineering components, JavaScript for the User Interfaces, Java for backend workflows, $C++$ for some compressing algorithms, etc). For each software component, a set of test cases has been created, and test cases might be added or modified after each change of the application's source code. For each software component we can compute a probability of failure based on our approach described in Chapter 1. Moreover, the software tester can add probabilities of execution on the edges connecting the components. Such probabilities can be extracted (i) after profiling how the application's clients are using the application, or (ii) based on preferences and domain expertise of the testers. The tester might decide to focus testing on:

1. the components highly to be executed

2. the components highly to contain faults
3. the components that are both highly to contain faults and highly to be executed by the application's clients.

We assume a limited testing budget is available at each CI cycle which does not allow to run all test cases. The limited testing budget is explained by the following facts:

1. As software components evolve the number of test cases keeps on growing making it practically impossible to execute all test cases [MGN+17].
2. The test cases for each component are executed on specific testing machines which are pre-configured based on the components programming language and setup requirements. For example, the test cases of a Python-based component are usually not executed on the same execution machine as the test cases of a JavaScript-based component. Moreover, such test execution machines might need to be shared between different applications within the enterprise in order to reduce infrastructure and maintenance cost.
3. The round-trip time between code commits and the feedback about failing test cases should be reduced as possible to enable quick fix and integration of the newly added source code with the mainline code base.

Our goal is to schedule the test cases of the software components for execution on the available test execution machines while satisfying the tester's preferences and the time constraint for each CI cycle. In each CI cycle, the test cases in each component are ranked from the highest likely to fail to lowest ones using our approach described in Chapter 5.

In order to solve our test case scheduling challenge, we formulate our problem to a Parallel Machine Scheduling Problem (PMSP) [KIL21].

Definition 8. PMSP is a scheduling problem with m machines, and n jobs with $m \leq n$, and:

- $w_{i\{i=1,\ldots,n\}}$ penalty weight for each job
- $p_{i\{i=1,\ldots,n\}}$ processing times
- each job $\mathbf{j}$ belongs to a job class $\mathcal{C}_\mathbf{j} \in \{1, \ldots, c\}$, with $c \leq n$
- if job $\mathbf{j}$ is scheduled to process immediately after job $\mathbf{j}'$ on the *same* machine, then there is an additional incurred *setup cost* $\mathcal{M}[\mathcal{C}_{\mathbf{j}'}, \mathcal{C}_\mathbf{j}]$, with $\mathcal{C}_{\mathbf{j}'} \neq \mathcal{C}_\mathbf{j}$, where M is a matrix with non-negative entries, and zeros in the diagonal.

The objective is to minimize the total *weighted* completion time (i.e., learn to dispatch jobs to machines step by step such that the total *weighted* completion ($\mathcal{WC}$) time is minimized), which consists of the waiting, setup, and processing times of all arrived jobs. [KIL21]

We map the test case scheduling problem to the PMSP problem as following:

- jobs: test cases
- job weight w_i: weight of the test case $\mathbf{j}_i$. The weight of test cases is defined in Section 6.4
- processing time p_i: execution time of the test case $\mathbf{j}_i$
- job class $\mathcal{C}_\mathbf{j}$: the software component to which the test case belongs
- setup cost $\mathcal{M}[\mathcal{C}_{\mathbf{j}'}, \mathcal{C}_\mathbf{j}]$: the weight of the software component to which the test case $\mathbf{j}_i$ belongs. The weight of software component is defined in Section 6.4

PMSP is a well known NP-hard combinatorial optimization problem that has been intensively studied over the past several decades. Regardless of the growing computing resources and the available efficient solvers, many practical problems are computationally expensive and problem specific heuristics and approximations have been deployed [WS11], [Lom65], [GKU97], [Yim99], [Gup06]. However, such techniques are usually difficult to apply to dynamic scheduling problems where the conditions of the problem continuously change since they are designed to compute the entire schedule assignments for given initial conditions. Consequently, re-execution of these methods whenever the scheduling problem changes is required making such techniques impractical because of the expensive time required to find a solution. Recent researches have studied the application of reinforcement learning techniques for solving combinatorial problems and specifically also scheduling problems [ZSC+20]. Unlike analytical methods, reinforcement learning based learned policies can be evaluated in real-time and consequently enable fast response times without sacrificing the quality of the solutions. Recent advances in reinforcement learning have trained policies on expressive representations [ZSC+20] of the scheduling problem using Graph Neural Networks (GNN) making the trained policy capable of solving problems of different sizes.

In our context of scheduling the execution of test cases across the software components in each CI cycles, the scheduling method requires the following properties:

1. It should deal with the dynamic changes happening during the software development and CI cycles. It should adapt to situations where test cases are added or deleted, and when testing priorities change because of changing failure indications in different code regions as the code matures or as the testers' requirements and preferences change.
2. It should have fast response times to be applied within each CI cycle without adding considerable time overhead. We aim at getting a scheduling solution in less than one minute.

In order to satisfy the above requirements, we propose a novel reinforcement learning based method for solving PMSP problems, as described in Figure 6.1.

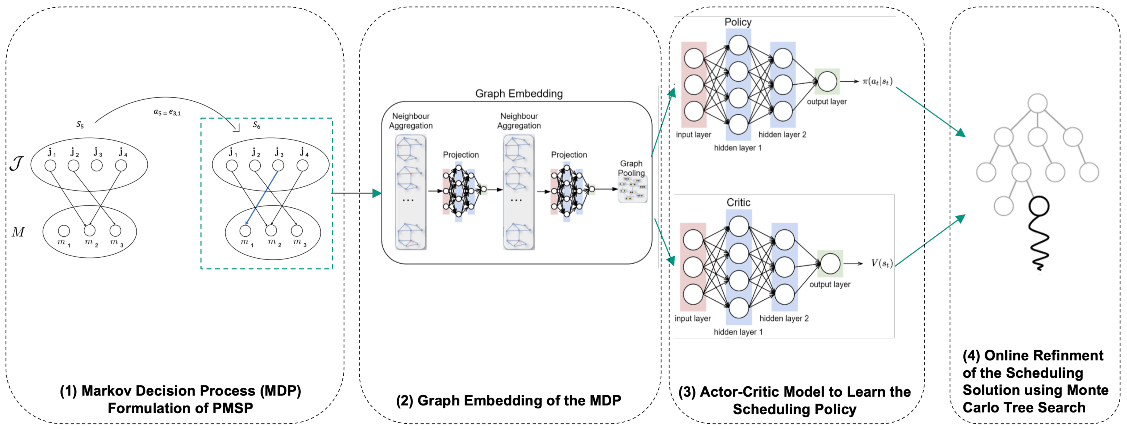


Figure 6.1: Test Case Scheduling Process.

We propose to model the PMSP state-action space as a graph representing the structure of the problem and learn a GNN policy that operates on it. GNNs offer properties such as invariance to node neighborhood sizes, node permutations, and graph sizes. In our case, GNNs allow us to use the same learned compact network to handle PMSP problems of different sizes. In order to deal with dynamic changes in the scheduling problem, we represent each separate solution as a graph. Concretely, we model a sequential solution process of the PMSP problem instance where at each time t a partial solution is extended, using a finite horizon Markov Decision Process (MDP; [Put94]) of T steps. At each time t, the state s_t corresponds to a partial solution, an action a_t corresponds to a feasible extension of s_t, a reward $r_{t+1} = r(s_t, a_t)$, and a transition probability $p(s'|s_t, a)$. The action distribution is set by a policy $\pi(a|s)$. This leads to a distribution over *trajectories* $\rho = (\langle s_t, a_t, r_{t+1}\rangle)_{t=0,\ldots,T-1}$, $p(\rho) = p(s_0)\prod_{t=0}^{T-1}\pi(a_t|s_t)p(s_{t+1}|s_t, a_t)$. The Q-function is defined as

$$Q(s_t, a_t) \triangleq \mathbb{E}_\rho\left[\sum_{i=0}^{N-t} r(s_i, a_i)\middle| s_0 = s_t, a_0 = a_t\right],$$

where the agent's objective is to find an optimal policy $\pi^*(a|s) = \arg\max_a Q(s,a)$ [SB18].

We encode the problem instance induced by a partial solution s_t, as a graph $G_t = (V_t, E_t, \mathcal{E}_t^v, \mathcal{E}_t^e)$ where:

- $\mathcal{E}_t^v$ maps nodes to feature vectors
- $\mathcal{E}_t^e$ maps edges to feature vectors

Combinatorial optimization problem like the PMSP are sensitive to perturbations in their solutions [HP04]. Search procedures such as Monte Carlo Tree Search (MCTS) have been recently applied in different fields such as playing Go [SHM+16] to increase the robustness of the optimization solutions. We improve the solution predicted from our GNN-based policy by guiding a Monte Carlo Tree Search (MCTS) using our trained policy, which allows us to mitigate possible degradation of the performance of the learned solutions.

6.3 Method for Solving PMSP

In this section, we present the rationale of our approach. We first formulate Markov Decision Process Model of the PMSP. Second, we design a method to train the scheduling policy based on a GNN, and third we improve our policy at solution time by using the trained policy to guide Monte Carlo Tree Search to ensure a more robust solution.

6.3.1 Markov Decision Process Formulation

Graph States:

In PMSP, a state s_t consists of:

- currently n_t pending jobs
- set of m machines
- remaining processing times $r_t^{(i)}{}_{\{i=1,\ldots,m\}}$
- last or currently assigned job classes (for computing setup times), $(\mathcal{C}_{t,i}^{last})_{i=1,\ldots,m}$ of the machines

We represent this as a complete bipartite graph $\mathcal{G}_t = (V_t, E_t, \mathcal{E}_t^v, \mathcal{E}_t^e)$, where

- $V_t = (\mathcal{J}_t, M)$
- $\mathcal{J}_t$: sets of job
- M: sets of machine nodes
- $E_t = \mathcal{J}_t \times M$ is the complete set of edges, connecting every job to every machine.
- $\mathcal{E}_t^v$: maps nodes to feature vectors
- $\mathcal{E}_t^e$: maps edges to feature vectors

As described in Figure 6.2, a job $\mathbf{j}_{i\{i=1,\ldots,n_t\}} \in V_t$, has a feature vector $\mathcal{E}_t^v(\mathbf{j}_i)$ represented as follows:

- p_i: processing time
- w_i: weight
- a_i: arrival time
- $\mathcal{C}_{\mathbf{j}_i}$: the class of the job

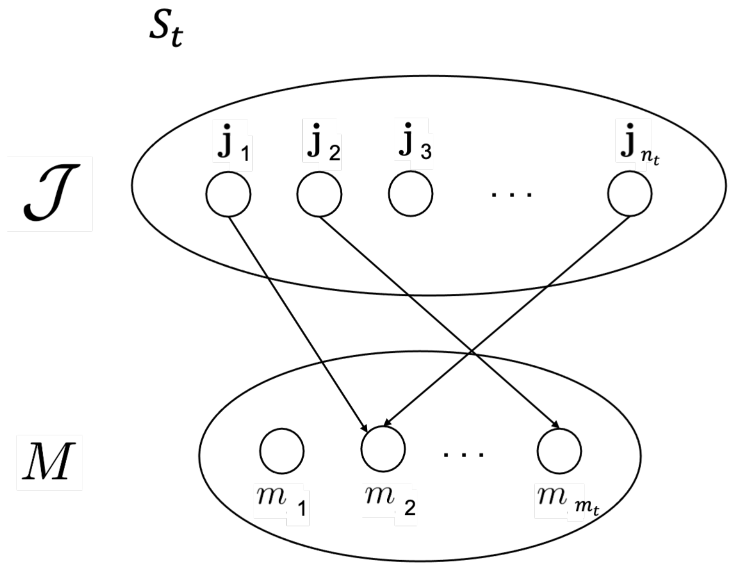


Figure 6.2: The GNN representation of a state S_t with n_t waiting jobs and m_t machines.

For a machine node $m_j \in M, j = 1, \ldots, m_t$, its feature vector is represented as follows:

- $\mathcal{C}_{t,j}$: the class of the last executed job
- $r_t^{(j)}$: the remaining processing time of the machine

The feature of edges $e_{ij} \in E_t$ connecting the job $\mathbf{j}_i$ to a machine m_j is:

- $\mathcal{M}[\mathcal{C}_i, \mathcal{C}_{t,j}] = setuptime$
- $\mathcal{M}[\mathcal{C}_i, \mathcal{C}_{t,j}] = 0$ if machine m_j was not processing a job.

Actions:

The Actions are specified by the edges that connect jobs to machines. An edge e_{ij} corresponds to assigning a job $\mathbf{j}_i$ to machine m_j at state s_t at time t when the machine m_j is free and we have at least one waiting job $\mathbf{j}_i$. We also allow special null action that executes no assignment to machines. Each action transitions our Markov Decision Process (MDP) at time t from state s_t to s_{t+1}, as shown in Figure 6.3.

Reward Function:

The goal is to learn to dispatch jobs to machines step by step such that the total *weighted* completion ($\mathcal{WC}$) time is minimized. We designed a reward function $R(s_t, a_t)$ as the quality difference between the partial solution at state s_t and s_{t+1}. We defined

$$R(s_t, a_t) = \mathcal{Q}(s_t) - \mathcal{Q}(s_{t+1}) \tag{6.1}$$

where $\mathcal{Q}(.)$ is the quality measure. We defined

$$\mathcal{Q}(s_t) = \sum_{i=\{1,\ldots,\alpha\}} w_i \cdot \sum_{i=\{1,\ldots,\alpha\}} p_i \tag{6.2}$$

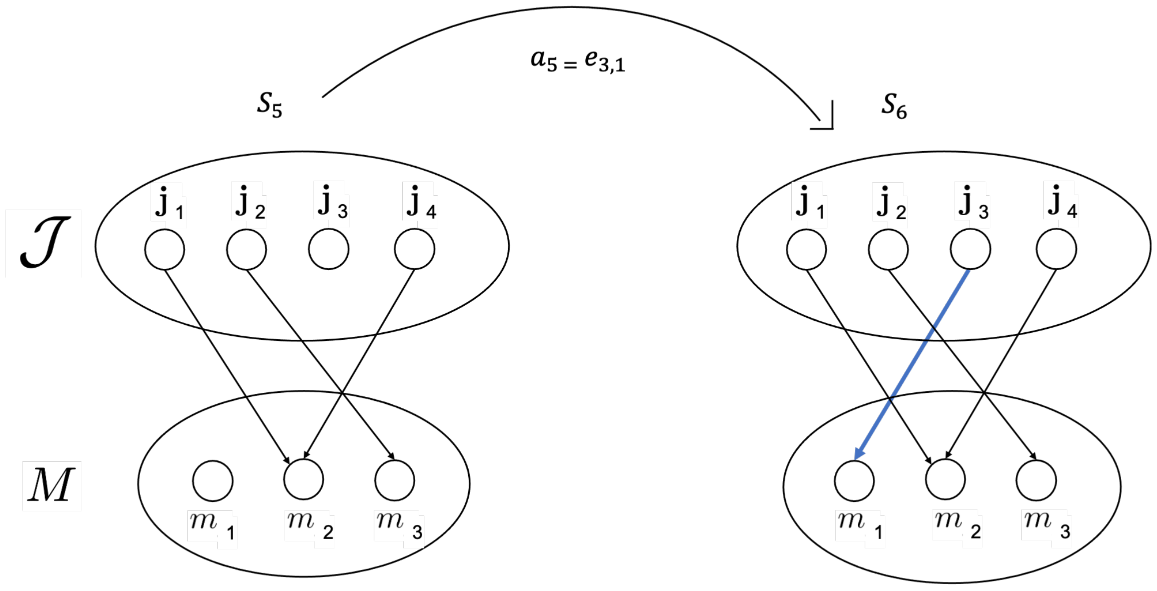


Figure 6.3: Example of State Transition.

where α is the number of jobs scheduled for execution until time t at state s_t. Consequently,

$$R(s_t, a_t) = \sum_{i=\{1,\dots,\alpha\}} w_i \cdot \sum_{i=\{1,\dots,\alpha\}} p_i - \sum_{i=\{1,\dots,\alpha+1\}} w_i \cdot \sum_{i=\{1,\dots,\alpha+1\}} p_i = -w_{\alpha+1} p_{\alpha+1} \tag{6.3}$$

meaning that at each time t a cost is incurring since our problem is minimization problem.

6.3.2 Parameterizing the Policy

The MDP formulation in Section 6.3.1 provides a holistic view of the scheduling states and the sequential solution construction process. This motivates the parametrization of the stochastic policy $\pi(a_t \mid s_t)$ as a graph neural network with trainable parameter θ, i.e. $\pi_\theta(a_t \mid s_t)$. Such parametrization enables a size-agnostic generalization.

Graph Embedding:

Graph Neural Networks (GNN) [BHea18] are a family of deep neural networks able to learn the representation of graph structured data, by extracting feature embedding of the graph nodes in an iterative and non-linear manner. Concretely in our approach, we use the Graph Isomorphism Network (GIN) [XHLJ19] which is a recent implementation of the GNN approach. Given a graph $\mathcal{G}(V, E)$ with node feature vector X_v for $v \in V$, GIN executes K iterations to compute an embedding vector for each node $v \in V$, and at each iteration k the embedding $h_v^{(k)}$ of node v is expressed as:

$$h_v^{(k)} = MLP_{\theta_k}^{(k)} \left(\left(1 + \epsilon^{(k)}\right) \cdot h_v^{(k-1)} + \sum_{u \in \mathcal{N}(v)} h_v^{(k-1)} \right) \tag{6.4}$$

where

- $h_v^{(0)} = X_v$
- $MLP_{\theta_k}^{(k)}$: a Multi-Layer Perceptron (MLP)

- θ_k: parametrization of the MLP at iteration k
- ϵ: a learnable parameter
- $\mathcal{N}(v)$: the neighborhood of node v

An average pooling function [XHLJ19], adapted from [ZSC+20]

$$L(\left\{h_v^{(K)} : v \in V\right\}) = \frac{1}{\mid V \mid} \sum_{v \in V} h_v^{(K)} = h_{\mathcal{G}}^{(K)} \tag{6.5}$$

takes the embeddings of all nodes as input and outputs a d-dimensional embedding vector for the whole graph, $h_{\mathcal{G}}^{(K)} \in \mathbb{R}^d$

Action Selection:

The selection of an action a_t at state s_t is decided based on a probability distribution over the action space from which a_t can be sampled. This is done via an action selection network that uses the extracted graph embedding $h_{\mathcal{G}}^{(K)}$. An MLP is used to compute the score for each action a_t with $score(a_t) = MLP_{\theta_\pi}([h_{a_t}^{(K)}, h_{\mathcal{G}}^{(K)}])$, with [.] a concatenation operator. A softmax function is applied to output a distribution $P(a_t)$ over the computed scores, which is used to sample actions for training. During execution a greedy approach is used to pick the actions with maximum probability.

6.3.3 Learning Algorithm

We use the Proximal Policy Optimization (PPO) algorithm to train the policy network. PPO is an actor-critic algorithm, where the actor is the policy network π_θ described in Section 6.3.2, and the critic shares the same GIN network with the actor and uses an MLP MLP_{θ_v} with input $h_{\mathcal{G}}^{(K)}$ and as output an estimate of the cumulative rewards at state s_t.

Generalization Remarks: The proposed policy network is not bounded by the instance size $\mid \mathcal{J} \mid$ and $\mid M \mid$ (number of jobs and machines), because all parameters are shared across all nodes of the graph. This enables generalization of the trained policy to problem instances of different sizes without retraining [ZSC+20]. Moreover, it allows us to deal with dynamics and uncertainties such us new jobs or machines breakdown or new machines by adding or removing nodes or edges into our graph [ZSC+20].

6.3.4 Monte Carlo Tree Search

The Monte Carlo Tree Search approach is applied at runtime after training the policy as described in Section 6.3.2. We use this online search technique to optimize the action selection process based on the states observed in real time. More concretely, we use the Upper Confidence Bound applied to Trees (UCT) technique [KS06] together with $\tilde{Q}$-value trained in Section 6.3.2. The online search procedure samples rollouts iteratively from the root state s_t until a terminal state is reached. At each iteration of the online search, the Q-value of all previously traversed state-action pairs is estimated as

$$Q^{traversed}(s, a) = \sum_{i=1}^{n(s,a)} \frac{\bar{r}_i}{n(s, a)} \tag{6.6}$$

with

- $n(s, a)$: number of times action a was sampled in state s

- $\bar{r_i}$: the accumulated reward from state s over all traversings that passed through (s, a)

We select actions during the rollout procedure according to the UCT policy, where new actions that have not been yet sampled in the current state are selected, and if such action does not exist we select the one that maximizes:

$$Q^{traversed}(s,a) + \gamma\sqrt{\frac{\log(n(s))}{n(s,a)}} \tag{6.7}$$

with $n(s) = \sum_a n(s,a)$, and γ is the exploration factor.

The UCT algorithm guarantees asymptotic convergence to the optimal decision by sampling each action infinitely. In order to improve the practical efficiency of the online UCT based search, the search tree is expanded with at most one state at each iteration, and during rollout, if the visited state is not yet in the tree, the action that maximizes the $\tilde{Q}$-value is chosen.

6.4 Solving Test Case Scheduling

Scheduling the execution of test cases across the software components can be mapped to the PMSP as follows:

- jobs: test cases
- job weight w_i: weight of the test case $\mathbf{j}_i$. We will define later the weight based on the testers' preferences
- processing time p_i: execution time of the test case $\mathbf{j}_i$
- job class $\mathcal{C}_\mathbf{j}$: the software component to which the test case belongs
- machines: test case execution machines usually for different programming languages and different languages specific execution machines are required
- setup cost $\mathcal{M}[\mathcal{C}_{\mathbf{j}'}, \mathcal{C}_\mathbf{j}]$: the weight of the software component to which the test case $\mathbf{j}_i$ belongs. We will define later the weight based on the testers' preferences.

As illustrated in Figure 1.1 in Chapter 1, a software application can be composed of different software components. The software components might be developed in different programming languages (e.g., Python for Data Engineering, Java for Backend, JavaScript for User Interface, etc). The software components are connected to each other via required and provided interfaces [RBB11]. This creates a call graph describing the interactions between all components. The software tester can use the diagram in Figure 1.1 in Chapter 1 to add her preferences and domain knowledge as follows:

- the edges connecting the components can be annotated with the probability of execution, for example, as depicted in Figure 1.1 starting from component 1 the probability to execute component 3 is $\mathbb{P}_{execution}(component_1, component_3)$. Such a probability can be approximated based on the testers' domain knowledge of the application, for example, the tester might know based on profiling information that some functionalities are more likely to be executed than others. Another source to approximate such probability can be the release requirements.
- the components can annotated with the probability of failure $\mathbb{P}_{failure}(component_j)$. Such a probability can be approximated using our approach as presented in Chapter 4. The probability of failure can be also approximated based on the testers' assumptions about the maturity of the software component.

The preferences and the domain knowledge of the software tester is used to define the weights of the software components as well as the test cases. The software tester might be interested in focusing testing on the software components:

1. highly to be executed by the end user after release
2. having the highest probability of failure
3. both highly to be executed by the end user after release and having the highest probability of failure

Each software component has a set of test cases that are continuously added or updated as new code is integrated into the components' main source code. Using the approach presented in Chapter 5, each test case t_i has a rank

$$rank(t_i) \in (0, 1] \tag{6.8}$$

In the following sections, we present how we map the conditional probability of execution on the edges in Figure 1.1 to the probability of execution of each component (Section 6.4.1). Then, we describe in Section 6.4.2 how the weights of the test cases and the software components are computed.

6.4.1 Computing the Components Probability of Execution

The graph described in Figure 1.1 can be mapped to a first-order Markov Chain [Nor98]. A Markov Chain is a discrete time stochastic process that processes from one state to another with certain probabilities represented by graph and a state transition matrix P. Let s_i be a state of the Markov Chain which corresponds to the component $component_i$ in Figure 1.1. The transition matrix P is filled with the probability of execution annotated at each edge. For the edges that has not been annotated by the software tester, we assume a uniform distribution across the output edges of the components.

Consequently, the probability of execution of each component can be approximated by the stationary distribution π of the Markov Chain defined as follows: Based on the Markov Property:

$$s_{i+1} = s_i \cdot P$$

Recursively,

$$s_1 = s_0 \cdot P,$$
$$s_2 = s_1 \cdot P = (s_0 \cdot P) \cdot P = s_0 \cdot P^2,$$
$$s_n = s_0 \cdot P^n,$$

The stationary distribution π is:

$$\pi = s_0 \cdot P^n; n \to \infty$$

The stationary distribution can be solved algebraic or via simulation (example simulating from a multinomial distribution) [Nor98].

6.4.2 Computing the Weights of Test Cases and Components

The computation of the weights for the test cases as well as the software components depends on the software testers' preferences:

1. **Preference 1**: focus on testing the components highly to be executed by the end user after release
2. **Preference 2**: focus on testing the components having the highest probability of failure
3. **Preference 3**: focus on testing the components that are both highly to be executed by the end user after release and having the highest probability of failure

Preference 1:

We define the penalty weight of a test case t_i that belongs to a component $component_j$ as $w_{t_i} = (1 - rank(t_i)) \cdot (1 - \mathbb{P}_{execution}(component_j))$, where $rank_{t_i}$ computed as described in Chapter 5 and $\mathbb{P}_{execution}(component_j)$. The rationale behind the weight definition is to penalize low-ranked test cases in the components less likely to fail, which in term means to prioritize the ranked test cases in the component high likely to fail.

The weight of a component $component_j$ is defined as $w_{component_j} = 1 - \mathbb{P}_{execution}(component_j)$, where $\mathbb{P}_{execution}(component_j)$ is the probability of execution as described in Section 6.4.1.

Aligned with the PMSP formulation, for two components $component_j$ and $component_{j'}$, where the execution of $component_{j'}$ is triggered after $component_j$ in the graph, as described in Figure 1.1, the setup cost is defined as $\mathcal{M}[component_j, component_{j'}] = w_{component_{j'}}$. The rationale behind it is that we penalize the execution of test cases from the components with the lowest probability of execution.

Preference 2:

In analogy to **Preference 1**, we define:

- $w_{t_i} = (1 - rank(t_i)) \cdot (1 - \mathbb{P}_{execution}(component_j))$
- $w_{component_j} = 1 - \mathbb{P}_{failure}(component_j)$
- $\mathcal{M}[component_j, component_{j'}]$ as in **Preference 1**

Preference 3:

We define the weights as follows:

- $w_{t_i} = (1 - rank(t_i)) \cdot (1 - \mathbb{P}_{execution}(component_j)) \cdot (1 - \mathbb{P}_{failure}(component_j))$
- $w_{component_j} = (1 - \mathbb{P}_{failure}(component_j))(1 - \mathbb{P}_{execution}(component_j))$
- $\mathcal{M}[component_j, component_{j'}]$ as in **Preference 1**

6.5 Theoretical Cost Model

In this section, we present a model for the theoretical analysis of the cost effectiveness of our test case prioritization and scheduling approach to help decide when and whether our approach is beneficial in specific project and organization during continuous integration.

We use our cost model to reason about two strategies; running test cases (i) scheduled and prioritized by our approach, (ii) based on a random decision.

The model relies on the following parameters: (i) the cost of running the test cases, (ii) the cost of missing the execution of the failing test cases, and (iii) the failure rate of the executed test cases.

Our model is similar to the work presented in [Her19] where, for example, quality assurance, initialization, and defect costs were used to create a cost model for software defect prediction. Our approach, however, uses a set of parameters suitable for the continuous integration environment instead of defect prediction. A similar cost model in the context of continuous integration was proposed by [PP21] but was restricted for the binary classification (fail or pass) of the test suites assuming that each CI cycle executes one test suite.

6.5.1 Model Formulation:

Assume at a CI cycle c_i, a set of test cases $\mathcal{TS}_i = \{t_1, t_2, ..., t_{|\mathcal{TS}_i|}\}$ is scheduled for execution, where $\mathcal{TS}_i$ is a subset of the set of available test cases. We define:

- TP: the true positives defined as: $TP \subseteq \mathcal{TS}_i$, where $\forall t_i \in TP$, t_i **fails** at c_i.
- FP: the false positives defined as: $FP \subseteq \mathcal{TS}_i$, where $\forall t_i \in FP$, t_i **did not fail** at c_i.
- TN: the true negatives defined as: $TN \nsubseteq \mathcal{TS}_i$, and **will not fail** if they have been executed. This represents the set of test cases that we did not schedule for execution at CI cycle c_i, and will not fail if they have been executed at c_i
- FN: the false negatives defined as: $FN \nsubseteq \mathcal{TS}_i$, and **will fail** if they have been executed. This represents the set of test cases that we did not schedule for execution at CI cycle c_i, and will fail if they have been executed at c_i

Let $cost_{t_i}$ be the cost of execution of a test case t_i and $cost_{missing}$ the cost of not executing failure-inducing test cases at CI cycle c_i. We will give in Section 6.5.2 guidelines from literature to determine both cost parameters.

Our model tries to balance between the cost of execution $cost_{execution} = \sum_{i=1,...,|\mathcal{TS}_i|} cost_{t_i}$ and the cost of missing failure-inducing test cases leading to the following cost formulation

$$cost_{approach} = \sum_{t_i \in FP} cost_{t_i} \cdot \mid FP \mid + \sum_{t_i \in TP} cost_{t_i} \cdot \mid TP \mid + cost_{missing} \cdot \mid FN \mid \tag{6.9}$$

We use the random strategy as a baseline for our approach. Our approach tries to approximate at each CI cycle the true failure rate of the overall available test cases by sampling a subset $\mathcal{TS}_i$ of test cases likely to fail. The failure rate is defined as

$$failurerate = \frac{\mid FN \mid + \mid TP \mid}{N} \tag{6.10}$$

where $N = \mid FN \mid + \mid TP \mid + \mid TN \mid + \mid FP \mid$

For the random strategy, we define

- $\mid TP_{random} \mid = N \cdot (failurerate)^2$: the probability that the random model produces a true positive is the product of the probability of scheduling a test case for execution and the probability of seeing an actual failure after execution
- $\mid FP_{random} \mid = N \cdot (failurerate) \cdot (1 - failurerate)$: the probability that the random model produces a false positive is the product of the probability of scheduling a test case for execution and the probability of not seeing an actual failure after execution
- $\mid FN_{random} \mid = N \cdot (1 - failurerate) \cdot (failurerate)$: the probability that the random model produces a false negative is the product of the probability of not scheduling a test case for execution and the probability of seeing an actual failure after execution

Similarly to our cost model, we define the cost of the random model as

$$cost_{random} = \sum_{t_i \in FP_{random}} cost_{t_i} \cdot \mid FP_{random} \mid + \sum_{t_i \in TP_{random}} cost_{t_i} \cdot \mid TP_{random} \mid + cost_{missing} \cdot \mid FN_{random} \mid \tag{6.11}$$

The execution cost of the test cases $\sum cost_{t_i}$ can be approximated to an overall cost $cost_{all}$ (for example 2 person-hours) to simplify both cost Equations 6.9 and 6.11 as follows:

$$cost_{approach} = cost_{all} \cdot (\mid FP \mid + \mid TP \mid) + cost_{missing} \cdot \mid FN \mid \tag{6.12}$$

$$cost_{random} = cost_{all} \cdot (\mid FP_{random} \mid + \mid TP_{random} \mid) + cost_{missing} \cdot \mid FN_{random} \mid \tag{6.13}$$

Consequently, our approach is beneficial over the random strategy when

$$cost_{approach} - cost_{random} < 0 \tag{6.14}$$

which reduces to the following boundary condition, meaning when:

$$\frac{cost_{all}}{cost_{missing}} < \frac{\mid FN \mid - N \cdot failurerate \cdot (1 - failurerate)}{N \cdot failurerate - \mid FP \mid - \mid TP \mid},$$
$$if \mid FN \mid - N \cdot failurerate \cdot (1 - failurerate) < 0$$

$$\frac{cost_{all}}{cost_{missing}} > \frac{\mid FN \mid - N \cdot failurerate \cdot (1 - failurerate)}{N \cdot failurerate - \mid FP \mid - \mid TP \mid},$$
$$if \mid FN \mid - N \cdot failurerate \cdot (1 - failurerate) > 0$$

6.5.2 Cost Parameters

The cost of running test cases can be approximated by the following factors

- the computational cost of test case runs which depends on the size of the project and the complexity of test cases. For example, Amazon is conducting 136000 builds per day [LER18], and Google runs daily 150 million test cases [MGN+17] which corresponds to millions of dollars on hardware spent only for execution [HTH+16]
- the human cost of developers maintaining the test infrastructure or waiting for the result of the test case execution to integrate the new source code. At Facebook, around 5% of the engineers are dedicated to developing and maintaining the CI environment [SDG+16]. [BGZ17] reports that the mean execution time of CI cycles for open source projects are around 20 minutes

The cost, $cost_{missing}$ of not scheduling failure-inducing test cases for execution can be explained by the following: it is usually expected for the developers to require more time to fix and localize faults which have been revealed later after the inducing code changes have been already integrated into the main source code. Depending on the fault severity, the cost of missing faults during CI cycle is estimated between 1.5 to 22 person-hours [DR06]. [SBB+02] reports that finding faults later than immediately during new code integration can increase the fixing time by factors between 2 and 100.

6.6 Experimental Validation

Our approach is implemented in Python and scikit-learn and the source code, as well as the used data, is available for peer review [1]. In our experiments, we considered five sets of benchmarks originally proposed by Dunstall and Wirth [DW05a, DW05b] and used by Kramer et al. [KIL21] to analyze and improve Branch and Bound algorithms to solve PMSP. We selected out of the studied benchmarks in [KIL21] the ones where it took Branch and Bound algorithms more than one minute to find the solution. All benchmark instances and their corresponding solutions can be found under [pms]. The benchmark instances were created by Kramer et al. [KIL21, pms] as follows:

[1]Implementation available at `https://github.com/so1188/test-case-scheduler`

- job processing times and weights were randomly drawn from the intervals $[p_{min}, p_{max}] = [1, 100]$ and $[w_{min}, w_{max}] = [1, 10]$, respectively
- The family setup times were randomly generated in the range $[0, 50]$. For each combination of (n, m, f), 100 instances were created.

In the experimental validation of our scheduling algorithm, we consider its two flavors; (i) by only using the trained policy for the reinforcement agents, (ii) by applying an online refinement of the trained policy using the Monte Carlo Tree Search (MCTS).

We compare our algorithm to:

1. **SC**: state-of-the-art Mixed Integer Linear Program (MILP) algorithm Set Covering (SC) based on the reported results from Kramer et al. [KIL21]
2. **Zhang**: state-of-the-art Deep Reinforcement Learning Job Shop Scheduling algorithm proposed by Zhang et al. [ZSC+20]

For the SC algorithm, we did not run the benchmarks because of the unavailability of the source code. However, we report the results as provided in [KIL21, pms].

We modified the authors' implementation of the Deep Reinforcement Learning Job Shop Scheduling algorithm proposed by Zhang et al. [ZSC+20] in order to apply it to Parallel Machine Scheduling Problem (PMSP). We represent each schedule as a single list ordered by machines, starting with an entry for the machine, then entries for the jobs scheduled to that machine.

The results are summarized in Table 6.1. Column gap $\%$ represents the average gap between the found scheduling solution and the best known feasible solution as provided in [pms]. The scheduling solution is characterized by the makespan (time required for the execution of all jobs). Column $t(s)$ reports the average computation time. The average of gap $\%$ and $t(s)$ for each benchmark (represented as a row in Table 6.1) is computed over the 100 instances. We trained a policy for each of the three benchmarks (i) $(20, 3, 3)$, (ii) $(20, 8, 5)$, (iii) $(40, 5, 5)$. The two remaining benchmarks $(80, 8, 5)$ and $(80, 12, 5)$ were used to evaluate the generalization performance of our algorithm.

Table 6.1: Results on PMSP Benchmarks with and without MCTS

Size	**SC** (MILP)		**Zhang**		Ours without MCTS		Ours with MCTS		Ours (50,4, 4) with MCTS **(Generalization)**	
(#jobs, #classes, # machines)	gap %	t(s)	gap %	t(s)	gap %	t(s)	gap%	t(s)	gap%	t(s)
20, 3, 3	<0.1	∼2000	26%	∼2	20%	∼2	16%	60		
20, 8, 5	<0.1	∼2000	23%	∼2	23%	∼2	15%	60		
40, 5, 5	<0.1	∼2800	25%	∼12	25%	∼10	15%	60		
80, 8, 5	<0.1	∼3000	41%	∼28	-	-	-	-	15%	60
80, 12, 5	<0.1	∼3000	59%	∼28	-	-	-	-	13%	60

As shown in Table 6.1, our trained policy is slightly better than Zhang with comparable average computation time. Compare to SC, we are at least 280 times faster with an average gap of around 23%. We are able to reduce this gab to 15% by activating our MCTS based solution refinement algorithm with a search timeout of 60 seconds.

Next we evaluated the performance of our algorithm in terms of generalizing to large instances. More specifically, we used the policy trained on a $(50, 4, 4)$ scheduling problem with 50 jobs 4 job families and 4 machines to solve $(80, 8, 5)$ and $(80, 12, 5)$ benchmarks. The last column in Table 6.1 shows that our algorithm is able to extract knowledge from small sized instances to solve large-scale ones with an average gap of 14% to the best known feasible solution in only one minute compared to the 50 minutes required by SC. In summary, our algorithm shows desirable properties for practical applications.

Summary:

In the experimental validation, we first consider our scheduling algorithm by only using the trained policy for the reinforcement agents. It shows a better performance compared to two state-of-the-art algorithms. Second, we applied an online refinement of the trained policy using Monte Carlo Tree Search. Our scheduling algorithm shows the ability to reduce the gab between the found scheduling solution and the best known feasible solution to 15% by activating our Monte-Carlo-Tree-Search (MCTS) based solution refinement algorithm with a search timeout of 60 seconds. Moreover, our algorithm shows good performance in terms of generalizing to large instances.

6.6.1 Threats to Validity

A first threat is that our approach can be parameter sensitive, and a set of parameters appropriate for one problem environment may not work as well for another. In our experiments, the parameters initially chosen for different problems were not changed to allow for better comparison. In a real environment, these parameters can be adjusted to adapt the approach to the specific environment. A further threat is that combinatorial optimization problem like the PMSP are sensitive to perturbations in their solutions [HP04]. Search procedures such as Monte Carlo Tree Search (MCTS) have been recently applied in different fields such as playing Go [SHM+16] to increase the robustness of the optimization solutions. We improve the solution predicted from our GNN-based policy by guiding a Monte Carlo Tree Search (MCTS) using our trained policy, which allows us to mitigate possible degradation of the performance of the learned solutions.

6.7 Related Work

Reinforcement Learning Applied on Job Shop Scheduling Problems Deep Reinforcement Learning (DRL) as an end-to-end solution to combinatorial optimization problems has recently received a lot of attention. The majority of them are concerned with solving routing problems (for example, the travelling salesman problem) [VFJ15, NOST18, KvHW19, WSC+19, LZY20, ZPD20], graph optimization problems [KDea17, LCK18], and the satisfiability problem (SAT) [SLB+18, AMW19, YP19]. Scheduling problems, on the other hand, which have numerous real-world applications, are relatively unexplored, particularly for Job Shop Scheduling Problems (JSSP).

Several existing works investigate simple job scheduling problems, in which jobs are treated as elementary tasks with no internal operation dependencies, which are required by JSSP. Mao et al. [MAea16] proposes a DRL agent to learn job scheduling policies for a compute cluster. To capture the status of resources and jobs, a 2-D image-based state representation scheme is used. Chen and Tian [CT19] used DRL to learn local search heuristics for a similar problem, where the states are represented by a Directed Acyclic Graph (DAG) describing the temporal relationships among jobs in the corresponding schedule. The state representation in these works is hard-bounded by some factors (e.g., look ahead horizon, size of job queue or slot) and is not scalable to arbitrary numbers of jobs and machines (resources). This limitation is mitigated in the work of Zheng et al. [ZGS19], which also uses an image-based representation but with a transfer learning method to reconstruct the trained policies on problems of varying sizes. Nonetheless, policy transition is still relatively expensive and inconvenient. Our method, on the other hand, is completely size-agnostic, and the trained policy can be applied directly to larger problems without the need for transfer. The work closest to ours is Zhang et al. [ZSC+20], it focuses, however, on the general job shop scheduling problem and not on the parallel machines scheduling problem.

A DRL method for task scheduling in a cloud computing environment is proposed by Mao et al. [MSV+19]. The policy network can scale to an arbitrary number of tasks because GNN is used to extract the embedding of each task represented as a DAG. The underlying problem, however, is not JSSP, and the task DAG only describes the necessary temporal dependencies among its subtasks. Because resource information is encoded as node features, the number of resources is fixed. In Zhang et al. [ZSC+20] a GNN was used to embed the disjunctive graph with directed disjunctive arcs reflecting processing order on each machine and is size-independent in terms of both jobs and machines. Our GNN, on the other hand, embeds the sequential solution refinement process allowing us to sequentially solve the PMSP problem.

A RL method combined with GNNs is proposed to accelerate computation in a distributed system [SGW+20], similar to [MSV+19]. Sun et al. [SGW+20] presented another example of using GNNs to solve real-world scheduling problems. They used an imitation learning algorithm to solve robotic scheduling problems in manufacturing.

Combining search algorithms and machine learning techniques can benefit each other by either using search to accelerate learning or learning better models for the search to use, or both. Waschneck et al. [WRB+18] used RL methods to solve scheduling problems using a multi-agent cooperative approach. Their experiments, however, did not show any clear advantages over heuristic algorithms. Chen and Tian [CT19] took a different approach, employing a DQN in the solution to improve a local search heuristic. Zhuwen et al. [ZQV18] labeled nodes in a graph using GNN and supervised learning to determine whether they belong to a Maximal Independent Set. This prediction was used in a tree search algorithm to find the best feasible solution predicted by the network. Due to superhuman play levels in board games such as Go, Chess, and Shogi, MCTS combined with RL methods has recently gained a lot of traction [SD17]. Laterre et al. [LFJ+18] integrated MCTS into an RL loop and used ranking rewards to solve the Bin Packing Problem (BPP) in two and three dimensions. In our work, we also refine the solution approximated by our RL trained policy using a MCTS with a time limit as a stopping criteria.

7. Validation

This thesis developed a new lightweight test case prioritization in CI environments. Our approach first predicts the fault density of each component to guide the testing effort to the components likely to contain the largest number of faults. Moreover, it minimizes the test prioritization overhead through formulating the test case prioritization problem as a computational efficient online learn-to-rank model using reinforcement learning techniques. In addition, our approach developed a test case execution scheduling model that captures the testing requirements as well as the testers' domain knowledge and learns to schedule test cases across the software components while considering the structure of the software application.

In this Chapter, we experimentally evaluate the goal of our approach in scheduling test cases across all software components for execution on the available test execution machines while satisfying the tester's preferences and the time constraint for each CI cycle. We experimentally evaluate the ability of our approach to reduce the testing overhead, taking into account the impact of the probability of failure of each software component and the testers' domain knowledge on the effectiveness and the efficiency of the test prioritization, as the results are dependent on the quality of the tools used (i.e., defects detected by static analysis tools used to predict the fault density of the software component might be false positive) and they are dependent on the availability of testers' domain knowledge.

7.1 Research Questions

We present an experimental evaluation of our method to address the following research questions:

1. **RQ1:** How efficient can our approach conduct priority-based test selection?
   - **RQ1.1:** Does the probability of failure of each software component impact the priority-based test selection' efficiency?
   - **RQ1.2:** Does the domain knowledge impact the priority-based test selection' efficiency?
2. **RQ2:** How effective can our approach conduct test case prioritization?
   - **RQ2.1:** Does the probability of failure of each software component impact the test case prioritization' effectiveness?
   - **RQ2.2:** Does the domain knowledge impact the test case prioritization' effectiveness?

3. **RQ3:** Is our approach applicable in the CI development context?

To evaluate the efficiency and the effectiveness of our method, we compare it against three test case prioritization methods:

1. Elbaum's Approach [ERP14]: this approach uses sliding time window to select test suites to be applied during pre-submit testing by tracking their history, then prioritizes test suites based on such window to be performed subsequent post-submit testing. Elbaum et al, were among the first that uses this technique in the context of test case prioritization in continuous integration. We were able to re-implement their approach since a detailed algorithm is available,

2. RETECS (Reinforced Test Case Selection) [SGMM17]: this approach is the first online learning approach, to the best of our knowledge, using reinforcement learning in the context of test prioritization and selection in continuous integration. RETECS considers as input the test case duration, historical failure data, and previous last execution. We execute the implementation of RETECS available in the literature [1] by using ANN [SGMM17].

3. Random: we use random test case prioritization as a baseline method.

To account for the influence of randomness within the experimental evaluation, all experiments are repeated 30 times, and the reported results show the mean. Our approach is implemented in Python and scikit-learn and the source code, as well as the used data, is available for peer review [2]. All the experiments are performed on an Intel Ⓡ Xeon Ⓡ $E5-2640$ v3 with 2.60 GHz CPU, 94GB RAM, running Linux Ubuntu 18.04.1 LTS.

7.2 Studied Data Sets

We validate our approach on two Web-based industrial analytical applications, where we had access to the CI execution history logs for over three years as well as the test cases execution history. We had also access to static analysis reports that were run on daily nightly builds using the tool SonarQube [3].

The two studied applications are:

1. QioTec Asset:
   - LoC: $\sim$ 20 MLoC
   - # Test Cases: $\sim$ 36K
   - # Components: 27
   - Programming Languages: Java, Python, Javascript, C++
   - Description: a web-based analytical application for ingesting, analyzing and visualizing industrial assets performance data. The visualization is written in Javascript. The backend logic is in Java. The data pipelines are in Python, and the asset performance algorithms are in C++ and Python.

2. QioTec Energy:
   - LoC: $\sim$ 23 MLoC
   - # Test Cases: $\sim$ 47K

[1] Implementation available at `https://bitbucket.org/helges/RETECS`
[2] Implementation available at `https://github.com/so1188/test-case-scheduler`
[3] `https://www.sonarqube.org/`

- # Components: 21
- Programming Languages: Java, Python, Javascript
- Description: a web-based analytical application for the sustainable and energy efficient operation of industrial assets. The visualization is written in Javascript. The backend logic is in Java. The data pipelines and the recommendation algorithms are in Python.

The data sets, detailed in Table 7.1, include test verdicts (failures), number of test cases and number of CI cycles. We also have static analysis metrics computed daily using SonaQube[3] for each of the software components of both studied software applications. Table 7.1 gives an overview of the data sets' structure, where all columns show the total amount of data in the data set.

Table 7.1: Data Sets Overview

Data Set	CI Cycles	Test Cases	Failures
QioTec Asset	10829	36173	8593
QioTec Energy	15723	47281	13751

7.3 Results and Analysis

To quantify the accuracy of our approach, we measure recall at various cut-off points, APFD, and NAPFD, respectively. The studied evaluation metrics are described in detail in Chapter 2 in Section 2.3.2.

7.3.1 RQ1: Priority-based test case selection efficiency.

We evaluated our approach on the two industrial data sets. Table 7.2 summarizes the results of the test recall over the studied projects. We consider both the probability of failure of each component and the probability of execution added by the testers. The Test Recall is measured as the percentage of test failures detected if tests ranked before the cut-off point are executed, as defined in Section 5.3.1. The experiments show that our approach can reach a recall of 0.95 after only executing 32% to 43% of the total available test cases. Table 7.2 summarizes how many tests need to be selected by each approach to achieve a given recall average for all studied projects. For the Industrial Asset Management Application (QioTec Asset), our approach requires 3617 (10%) top-ranked tests to detect 50% of the test failures, compared to at least 10321 (28%) tests when using the previous approaches. Our approach requires, for the Industrial Asset Energy Application (QioTec Energy), 17493 (37%) top-ranked tests to detect 75% of the test failures, compared to at least 24113 (51%) tests when using the previous approaches. Our method requires 11578 (32%) top-ranked tests for the QioTec Asset project and requires 20330 (43%) top-ranked tests for the QioTec Energy project to detect 95% of the test failures, whether RETECS needs for both projects at least about 29787 (63%) tests.

7.3.1.1 RQ1.1: Impact of the probability of failure on the efficiency of our approach

We further studied the impact of each component's probability of failure on the efficiency of test case prioritization. Table 7.3 summarizes the number of test cases needed to reach a given recall average. The test recall average for Random, Elbaum's Approach, and RETECS are the same as in Table 7.2, since these studied approaches do not consider the probability of failure of each component. The experiments show that our approach can reach a recall of 0.95 after executing 49% to

55% of the total available test cases. In comparison with the previous results, while considering the probability of failure of each component, we conclude that this information help to improve the average test recall. However, our approach still stands out in all cases and outperform the studied approaches.

7.3.1.2 RQ1.2: Impact of the probability of execution on the efficiency of our approach.

We compare our approach with the previous approaches while not considering the probability of execution defined by testers. Table 7.4 summarizes the results of the test recall over the studied projects with each approach. Since previous approaches do not consider the software testers' domain knowledge, the test recall average for Random, Elbaum's Approach, and RETECS are the same as represented in Table 7.2. Our approach requires, for the Industrial Asset Energy Application (QioTec Energy), 18439 (39%) top-ranked tests to detect 75% of the test failures, compared to at least 24113 (51%) tests when using the previous approaches. Our method requires 18448 (51%) top-ranked tests for the QioTec Asset project and requires 26950 (57%) top-ranked tests for the QioTec Energy project to detect 95% of the test failures, whether RETECS needs for both projects at least about 23928 (63%) tests. In this case, our approach outperforms also the studied approaches.

7.3.2 RQ2: Test case prioritization effectiveness.

We compare three test case prioritization methods with our method. We use the quality indicators: APFD and NAPFD to assess the effectiveness of our approach. The average is computed using results from 30 independent executions found by each approach in each studied projects. We highlighted the best values in bold. Elbaum et al. [ERP14] were among the first that used the sliding window principle in test case prioritization. We were able to re-implement Elbaum's approach since a detailed description of the algorithm is available. Moreover, to compare our approach with RETECS, we choose the best-performing Network-based RL agent (with Test Case Failure reward) [SGMM17]. Table 7.5 shows the APFD and NAPFD results of three approaches compared with our approach on each of the two data sets while considering both probability of execution and probability of failure of each component. Among the studied projects, the Random method has the worst performance. Our approach outperforms all studied approaches.

7.3.2.1 RQ2.1: Impact of the probability of failure on the effectiveness of our approach.

Table 7.6 summarizes the results of APFD and NAPFD over the studied projects with each approach. Since previous approaches do not consider the probability of failure of each component, APFD and NAPFD for Random, Elbaum's Approach, and RETECS are the same as in Table 7.5. In comparison with the previous results, described in Table 7.5, our approach has a lower performance while not considering the probability of failure of each component. For example, for the QioTec Energy application, the APFD decreases from 0.9983 to 0.8371 and NAPFD decreases from 0.9989 to 0.8094. However, our approach still outperforms all studied approaches.

7.3.2.2 RQ2.2: Impact of the probability of execution on the efficiency of our approach.

We studied, moreover, if the testers' domain knowledge and preferences (e.g., probability of execution) do impact the efficiency of our approach, and we compare it to previous approaches. Table 7.7 shows the APFD and NAPFD results of three approaches compared with our approach on each of the two data sets. Our approach has a lower performance compared to the results in Table 7.5, but still stands out in all cases. In addition, we observed a better performance of our approach compared to the previous results in Table 7.6 where the probability of failure is not considered. Thus, we conclude that having the probability of failure of each component without considering the probability of execution (Table 7.7) could outperform our approach while only considering the probability of execution, as described in Table 7.6.

7.3.3 RQ3: Is our approach applicable in the CI development context?

Our prioritization approach required for both studied applications an average computation time of less than a minute. Our scheduling algorithm including the MCTS part was executed with a timeout of one minute. Consequently, our overall approach required less than two minutes to recommend the list of test cases to be executed. This time overhead make our approach applicable in CI cycles.

Summary:

The probability of failure of each software components and the testers' domain knowledge impacts the results of our approach since we observe a lower performance while considering all evaluation metrics (test recall, APFD, and NAPFD). However, our approach still outperforms the other works. Recent works consider the software application as one single component when executing test cases. However, modern applications are composed of different components which have different probabilities of failure. This leads to sub-optimal allocation of test cases (meaning test cases could be executed in some of the component while neglecting other components). Moreover, recent works do not consider the domain knowledge of the software testers. Based on the testers' preferences and domain knowledge of the software application, a probability of execution can be approximated; for example, the tester might know based on profiling information that some functionalities are higher likely to be executed than others. Such a probability helps to better focus the testing activities and hence improve both the efficiency and the effectiveness of the test case prioritization. The empirical evaluation of our approach on 2 industrial data sets from 2 software applications gathered over a range from 1 to 3 years of continuous integration shows that our technique is applicable in real-world industrial settings.

7.3.4 Threats to Validity

The first threat is that our study is dependent on the quality of the static analysis tools, and might not be repeatable with the same degree of confidence with other tools. Moreover, it is possible that static analysis tools do not detect all faults during the development process. In order to mitigate possible skewness, our hypothesis was that combining the static analysis fault density with code complexity metrics and code churn metrics would account for faults not identified by static analysis tools. Furthermore, we experimentally evaluate the performance of our approach without taking into account the impact of the probability of failure of each software component, as the results are dependent on the quality of the tools used. The second threat to validity for our approach is the influence of random decisions on the results. To mitigate the threat, we repeated our experiments 30 times and reported averaged results, and we tested for significant differences with a two-sided student's t-test. A further threat is that our approach can be parameter sensitive, and a set of parameters appropriate for one problem environment may not work as well for another. In our experiments, the parameters initially chosen for different problems were not changed to allow for better comparison. In a real environment, these parameters can be adjusted to adapt the approach to the specific environment. Our validation was executed on two real-world industrial applications.

Table 7.2: Average Recall by Number of Tests for the Studied Projects

	Percentage of Detected Test Failures											
	Random			**Elbaum's Approach**			**RETECS**			**Our Approach**		
	50%	75%	95%	50%	75%	95%	50%	75%	95%	50%	75%	95%
QioTec Asset	15573 (~ 44%)	24961 (~ 69%)	29214 (~ 81%)	11937 (~ 33%)	22065 (~ 61%)	26406 (~ 73%)	10321 (~ 28%)	19763 (~ 55%)	24235 (~ 67%)	**3617 (~ 10%)**	**9405 (~ 26%)**	**11578 (~ 32%)**
QioTec Energy	26477 (~ 56%)	33569 (~ 71%)	43025 (~ 91%)	19385 (~ 41%)	27423 (~ 58%)	35461 (~ 75%)	17021 (~ 36%)	24113 (~ 51%)	29787 (~ 63%)	**6146 (~ 13%)**	**17493 (~ 37%)**	**20330 (~ 43%)**

Table 7.3: Average Recall by Number of Tests: ***without*** *considering the probability of failure of each component*

	Percentage of Detected Test Failures											
	Random			**Elbaum's Approach**			**RETECS**			**Our Approach**		
	50%	75%	95%	50%	75%	95%	50%	75%	95%	50%	75%	95%
QioTec Asset	15573 (~ 44%)	24961 (~ 69%)	29214 (~ 81%)	11937 (~ 33%)	22065 (~ 61%)	26406 (~ 73%)	10321 (~ 28%)	19763 (~ 55%)	24235 (~ 67%)	**7596 (~ 21%)**	**13384 (~ 37%)**	**17724 (~ 49%)**
QioTec Energy	26477 (~ 56%)	33569 (~ 71%)	43025 (~ 91%)	19385 (~ 41%)	27423 (~ 58%)	35461 (~ 75%)	17021 (~ 36%)	24113 (~ 51%)	29787 (~ 63%)	**13238 (~ 28%)**	**19385 (~ 41%)**	**26004 (~ 55%)**

Table 7.4: Average Recall by Number of Tests: ***without*** *considering the probability of execution*

	Percentage of Detected Test Failures											
	Random			**Elbaum's Approach**			**RETECS**			**Our Approach**		
	50%	75%	95%	50%	75%	95%	50%	75%	95%	50%	75%	95%
QioTec Asset	15573 (~ 44%)	24961 (~ 69%)	29214 (~ 81%)	11937 (~ 33%)	22065 (~ 61%)	26406 (~ 73%)	10321 (~ 28%)	19763 (~ 55%)	24235 (~ 67%)	**9043 (~ 25%)**	**13745 (~ 38%)**	**18448 (~ 51%)**
QioTec Energy	26477 (~ 56%)	33569 (~ 71%)	43025 (~ 91%)	19385 (~ 41%)	27423 (~ 58%)	35461 (~ 75%)	17021 (~ 36%)	24113 (~ 51%)	29787 (~ 63%)	**14657 (~ 31%)**	**18439 (~ 39%)**	**26950 (~ 57%)**

Table 7.5: Mean and Standard Deviation APFD and NAPFD

	APFD				NAPFD			
	Elbaum's Approach	RETECS	Random	Our Approach	Elbaum's Approach	RETECS	Random	Our Approach
QioTec Asset	0.5767 ± 0.008	0.6891 ± 0.000	0.4193 ± 0.002	**0.9971 ± 0.000**	0.5567 ± 0.008	0.6861 ± 0.000	0.419 ± 3 0.002	**0.9986 ± 0.000**
QioTec Energy	0.5349 ± 0.008	0.6698 ± 0.000	0.3617 ± 0.002	**0.9983 ± 0.000**	0.5439 ± 0.008	0.6783 ± 0.000	0.3647 ± 0.002	**0.9989 ± 0.000**

Table 7.6: Mean and Standard Deviation APFD and NAPFD: ***without*** *considering the probability of failure of each component*

	APFD				NAPFD			
	Elbaum's Approach	RETECS	Random	Our Approach	Elbaum's Approach	RETECS	Random	Our Approach
QioTec Asset	0.5767 ± 0.008	0.6891 ± 0.000	0.4193 ± 0.002	**0.8795 ± 0.000**	0.5567 ± 0.008	0.6861 ± 0.000	0.419 ± 3 0.002	**0.8142 ± 0.000**
QioTec Energy	0.5349 ± 0.008	0.6698 ± 0.000	0.3617 ± 0.002	**0.8371 ± 0.000**	0.5439 ± 0.008	0.6783 ± 0.000	0.3647 ± 0.002	**0.8094 ± 0.000**

Table 7.7: Mean and Standard Deviation APFD and NAPFD: ***without*** *considering the probability of execution*

	APFD				NAPFD			
	Elbaum's Approach	RETECS	Random	Our Approach	Elbaum's Approach	RETECS	Random	Our Approach
QioTec Asset	0.5767 ± 0.008	0.6891 ± 0.000	0.4193 ± 0.002	**0.9035 ± 0.000**	0.5567 ± 0.008	0.6861 ± 0.000	0.419 ± 3 0.002	**0.8537 ± 0.000**
QioTec Energy	0.5349 ± 0.008	0.6698 ± 0.000	0.3617 ± 0.002	**0.8413 ± 0.000**	0.5439 ± 0.008	0.6783 ± 0.000	0.3647 ± 0.002	**0.8974 ± 0.000**

8. Conclusion

8.1 Summary

This thesis investigates test case scheduling for software testing in the context of Continuous Integration (CI) environments. We propose a regressive model that predicts the fault density of each component based on quality metrics and code churn metrics and a classification model to discriminate faulty from non-faulty components. Both models are used to guide the testing effort to the software components likely to fail. Moreover, we propose methods for test case prioritization and test case scheduling which utilize Reinforcement Learning (RL) principles. We formulate the test case prioritization as an online learn-to-rank model based on foundational results of learning-to-rank from the field of Information Retrieval (IR) using reinforcement learning techniques based on a gradient-based policy search algorithm called Dueling Bandit Gradient (DBGD). DBGD is suitable for online learning of test case ranks because it can generalize over CI cycles as it only requires the relative evaluations of the quality of two test case lists (two test suites) to compute the ranks. The test case scheduling is formulated as a Parallel Machine Scheduling Problem (PMSP), a well known NP-hard combinatorial optimization problem. For solving such a combinatorial problem, we formulated the solving process as an RL problem. RL-based learned policies can be evaluated in real-time and consequently enable fast response times. Furthermore, we used a Graph Neural Network (GNN) to model the sequential solving process. We also represented each separate solution as a graph to enable our approach to generalize over problem instances of different sizes.

The developed approach has been beneficial in dealing with the dynamic changes happening during the software development and CI cycles. It adapts to situations where test cases are added or deleted, and when testing priorities change because of changing failure indications in different code regions as the code matures or as the testers' requirements and preferences change. Furthermore, it has fast response times to be applied within each CI cycle without adding considerable time overhead.

In conclusion, we developed a test case prioritization approach in CI environments that captures testers' requirements and domain knowledge and learns to schedule test cases across the software components. In particular, we formulate the test prioritization problem as a sequential decision-making process using reinforcement learning. Our approach captures the domain knowledge as well as the testers' requirements in the form of a probabilistic graph annotated with the preferences where to focus testing. Our experimental evaluation results show fast learning of our approach on two industrial case studies. Our method is language agnostic and does not require access to source code and hence can be integrated as an extension or plugin for existing CI environments like CircleCI, Gitlab CI/CD, Github actions, etc.

8.2 Future Work

As research is rarely complete, there are also directions for future research in this case. While the work on test case prioritization and test scheduling using reinforcement learning shows promising results, there are further steps that can be taken. Our method uses a lightweight set of historical test case metadata for prioritization. Further work should also consider coverage metrics of test cases during the early stages of development when little failure historical data is available. This could allow improving the effectiveness of our approach. At the same time, this also increases the complexity of the method in each CI cycle since coverage tools require source code access, while our current approach is easier to integrate into any existing CI environment. Still, in a CI environment, the actual tasks to be solved in each cycle are often similar to previous cycles. Thus, future work on re-using the previous CI results should allow for potential speedups in future cycles. Furthermore, adding information from the execution environment to the reinforcement learning environment, can enhance the quality of the test scheduler; for example, by integrating with software engineering frameworks like Palladio [RBB11] to simulate/consider hardware environment, network environment, and Third-Party Framework reliability.

List of Figures

List of Tables

Publications

[1] Safa Omri, Pascal Montag, Carsten Sinz, "Static analysis and code complexity metrics as early indicators of software defects", in *Journal of Software Engineering and Applications 11, 2018*, pp. 153-166.

[2] Safa Omri, C. Sinz, P. Montag, "An Enhanced Fault Prediction Model for Embedded Software based on Code Churn, Complexity Metrics, and Static Analysis Results", in *The Fourteenth International Conference on Software Engineering Advances ICSEA, 2019.*

[3] Safa Omri, Carsten Sinz, "Deep Learning for Software Defect Prediction: A Survey", in *ICSE'20 IEEE/ACM 42nd International Conference on Software Engineering Workshops (ICSEW), 2020.*

[4] Safa Omri, Carsten Sinz, "Learning to Rank for Test Case Prioritization", in *ICSE'22 IEEE/ACM 44nd International Conference on Software Engineering Workshops (ICSEW) in the 15th Intl. Workshop on Search-Based Software Testing (SBST), 2022.*

[5] Safa Omri, Carsten Sinz, "Learning to Schedule Test Cases across Software Components based on Ranked Test Cases and with Human in the Loop", submitted to *ICSE'22 IEEE/ACM 44nd International Conference on Software Engineering Workshops (ICSEW) in the 15th Intl. Workshop on Search-Based Software Testing (SBST), 2022.*

Bibliography

[AAYK20] H. Aman, S. Amasaki, T. Yokogawa, and M. Kawahara, “A comparative study of vectorization-based static test case prioritization methods,” in *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2020, pp. 80–88.

[AD13] M. J. Arafeen and H. Do, “Test case prioritization using requirements-based clustering,” in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, 2013, pp. 312–321.

[AKR18] E. Alégroth, A. Karlsson, and A. Radway, “Continuous integration and visual gui testing: Benefits and drawbacks in industrial practice,” in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, 2018, pp. 172–181.

[Ami18] A. Amidi, *Machine Learning*, 2018. [Online]. Available: https://stanford.edu/~shervine/teaching/cs-229

[AMS09] J.-Y. Audibert, R. Munos, and C. Szepesvari, “Exploration-exploitation tradeoff using variance estimates in multi-armed bandits,” *Theoretical Computer Science*, vol. 410, no. 19, pp. 1876–1902, 2009, algorithmic Learning Theory.

[AMW19] S. Amizadeh, S. Matusevych, and M. Weimer, “Learning to solve circuit-sat: An unsupervised differentiable approach,” in *ICLR*, 2019.

[ARB17] T. Avgerinos, A. Rebert, and D. Brumley, “Methods and systems for automatically testing software,” 2017. [Online]. Available: https://www.google.com/patents/US9619375

[ASD14] J. Anderson, S. Salem, and H. Do, “Improving the effectiveness of test suite through mining historical data,” in *Proceedings of the 11th Working Conference on Mining Software Repositories*. Association for Computing Machinery, 2014, pp. 142–151.

[ASD15] ——, “Striving for failure: An industrial case study about test failure prediction,” in *IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2, 2015, pp. 49–58.

[ATMK03] S. Amasaki, Y. Takagi, O. Mizuno, and T. Kikuno, “A bayesian belief network for assessing the likelihood of fault content,” in *Proceedings of the 14th International Symposium on Software Reliability Engineering*, 2003.

[BBM96] V. R. Basili, L. C. Briand, and W. L. Melo, “A validation of object-oriented design metrics as quality indicators,” *IEEE Trans. Softw. Eng.*, 1996.

[Bel57] R. Bellman, “A markovian decision process,” *Journal of Mathematics and Mechanics*, vol. 6, no. 5, pp. 679–684, 1957. [Online]. Available: http://www.jstor.org/stable/24900506

[BGM+20] A. Bertolino, A. Guerriero, B. Miranda, R. Pietrantuono, and S. Russo, "Learning-to-rank vs ranking-to-learn: Strategies for regression testing in continuous integration," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering.* Association for Computing Machinery, 2020, pp. 1–12.

[BGZ17] M. Beller, G. Gousios, and A. Zaidman, "Oops, my tests broke the build: An explorative analysis of travis ci with github," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, 2017, pp. 356–367.

[BHea18] P. W. Battaglia, J. B. Hamrick, and et al., "Relational inductive biases, deep learning, and graph networks," *arXiv preprint:1806.01261*, 2018.

[BKB21] M. Bagherzadeh, N. Kahani, and L. Briand, "Reinforcement learning for test case prioritization," *IEEE Transactions on Software Engineering*, pp. 1–1, 2021.

[BNJ03] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *J. Mach. Learn. Res.*, 2003.

[BWIL99] L. C. Briand, J. Wüst, S. V. Ikonomovski, and H. Lounis, "Investigating quality factors in object-oriented designs: An industrial case study," in *Proceedings of the 21st International Conference on Software Engineering*, 1999.

[BX16] 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*, ser. FSE 2016, 2016, pp. 975–980.

[CBH+16] J. Chen, Y. Bai, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie, "Test case prioritization for compilers: A text-vector based approach," in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2016, pp. 266–277.

[CCZ+11] S. Chen, Z. Chen, Z. Zhao, B. Xu, and Y. Feng, "Using semi-supervised clustering to improve regression test selection techniques," in *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, 2011, pp. 1–10.

[CDD11] R. Carlson, H. Do, and A. Denton, "A clustering approach to improving test case prioritization: An industrial case study," in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, 2011, pp. 382–391.

[CK94] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Softw. Eng.*, 1994.

[CKL16] Y. Cho, J. Kim, and E. Lee, "History-based test case prioritization for failure information," in *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*, 2016, pp. 385–388.

[CLZ+18] J. Chen, Y. Lou, L. Zhang, J. Zhou, X. Wang, D. Hao, and L. Zhang, "Optimizing test prioritization via test distribution analysis," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* Association for Computing Machinery, 2018, pp. 656–667.

[CT19] X. Chen and Y. Tian, "Learning to perform local rewriting for combinatorial optimization," in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32. Curran Associates, Inc., 2019.

[CTNH12] T.-H. Chen, S. W. Thomas, M. Nagappan, and A. E. Hassan, "Explaining software defects using topic models," in *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, 2012.

[Cum15] J. Cumming, "An investigation into the use of reinforcement learning techniques within the algorithmic trading domain," in *Imperial College London Department of Computing*, 2015.

[CVMG17] A. Celik, M. Vasic, A. Milicevic, and M. Gligoric, "Regression test selection across jvm boundaries," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery, 2017, pp. 809–820.

[DDB$^+$19] V. H. S. Durelli, R. S. Durelli, S. S. Borges, A. T. Endo, M. M. Eler, D. R. C. Dias, and M. P. GuimarÃ£es, "Machine learning applied to software testing: A systematic mapping study," *IEEE Transactions on Reliability*, pp. 1189–1212, 2019.

[DMP02] G. Denaro, S. Morasca, and M. Pezzè, "Deriving models of software fault-proneness," in *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering*, ser. SEKE '02. New York, NY, USA: ACM, 2002.

[DPN$^+$18] K. H. Dam, T. Pham, S. W. Ng, T. Tran, J. Grundy, A. K. Ghose, T. Kim, and C.-J. Kim, "A deep tree-based model for software defect prediction," *ArXiv*, 2018.

[DPN$^+$19] H. K. Dam, T. Pham, S. W. Ng, T. Tran, J. Grundy, A. Ghose, T. Kim, and C.-J. Kim, "Lessons learned from using a deep tree-based model for software defect prediction in practice," in *Proceedings of the 16th International Conference on Mining Software Repositories*, 2019.

[DR06] H. Do and G. Rothermel, "An empirical study of regression testing techniques incorporating context and lifetime factors and improved cost-benefit models," in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Association for Computing Machinery, 2006, pp. 141–151.

[DS17] J. S. I. A. M. L. A. G. D. Silver, T. Hubert, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," *CoRR*, 2017.

[DW05a] S. Dunstall and A. Wirth, "A comparison of branch-and-bound algorithms for a family scheduling problem with identical parallel machines," *European Journal of Operational Research*, vol. 167, no. 2, pp. 283–296, 2005.

[DW05b] ——, "Heuristic methods for the identical parallel machine flowtime problem with set-up times," *Computers & Operations Research*, vol. 32, no. 9, pp. 2479–2491, 2005.

[EE08] K. O. Elish and M. O. Elish, "Predicting defect-prone software modules using support vector machines," *J. Syst. Softw.*, 2008.

[EMG12] Elhampaikari, M. M.richter, and Guentherruhe, "Defect prediction using case-based reasoning: an attribute weighting technique based upon sensitivity analysis in neural network," *International Journal of Software Engineering and Knowledge Engineering*, 2012.

[EMR01] 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. ICSE 2001*, 2001, pp. 329–338.

[ERL11] E. Engstr̉om, P. Runeson, and A. Ljung, "Improving regression testing transparency and efficiency with history-based prioritization – an industrial case study," in *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, 2011, pp. 367–376.

[ERP14] 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*, ser. FSE 2014. Association for Computing Machinery, 2014, pp. 235–245.

[ESAO$^+$17] P. Erik Strandberg, W. Afzal, T. J. Ostrand, E. J. Weyuker, and D. Sundmark, “Automated system-level regression test prioritization in a nutshell,” *IEEE Software*, vol. 34, no. 4, pp. 30–37, 2017.

[FIS$^+$03] Y. Freund, R. Iyer, R. E. Schapire, Y. Singer, and G. Dietterich, “An efficient boosting algorithm for combining preferences,” in *Journal of Machine Learning Research*, 2003, pp. 170–178.

[FMB$^+$16] J. Feist, L. Mounier, S. Bardin, R. David, and M.-L. Potet, “Finding the needle in the heap: Combining static analysis and dynamic symbolic execution to trigger use-after-free,” in *Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering*, ser. SSPREW ’16, 2016, pp. 2:1–2:12.

[GBT18] M. Guillame-Bert and O. Teytaud, “Exact distributed training: Random forest with billions of examples,” 2018.

[GEM15] 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*, ser. ISSTA 2015, 2015, pp. 211–222.

[GKMS00] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, “Predicting fault incidence using software change history,” *IEEE Transactions on Software Engineering*, 2000.

[GKU97] C. D. Geiger, K. G. Kempf, and R. Uzsoy, “A tabu search approach to scheduling an automated wet etch station,” *Journal of Manufacturing Systems*, vol. 16, no. 2, pp. 102–116, 1997.

[GR02] J. v. R. C. H. Gregg Rothermel, Mary Jean Harrold, “Empirical studies of test-suite reduction,” *Proceedings of the International Conference on Software Maintenance*, 2002.

[GSR10] N. Gayatri, N. Savarimuthu, and A. Reddy, “Feature selection using decision tree induction in class level metrics dataset for software defect predictions,” *Lecture Notes in Engineering and Computer Science*, 2010.

[Gup06] S. A. Gupta, A., “Job shop scheduling techniques in semiconductor manufacturing,” vol. 27, pp. 1163–1169, 2006.

[Her19] S. Herbold, “On the costs and profit of software defect prediction,” *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.

[HJL$^+$01] 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 *Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications.* New York, NY, USA: Association for Computing Machinery, 2001, pp. 312–326.

[HMOK18] A. Haghighatkhah, M. M., M. Oivo, and P. Kuvaja, “Test prioritization in continuous integration environments,” *Journal of Systems and Software*, vol. 146, pp. 80–98, 2018.

[HOBK17] A. Haghighatkhah, M. Oivo, A. Banijamali, and P. Kuvaja, “Improving the state of automotive software engineering,” *IEEE Software*, vol. 34, no. 5, pp. 82–86, 2017.

[HP04] N. Hall and M. Posner, “Sensitivity analysis for scheduling problems,” *J. Scheduling*, vol. 7, pp. 49–83, 01 2004.

[HP19] A. Habib and M. Pradel, “Neural bug finding: A study of opportunities and challenges,” *CoRR*, 2019.

[HPMY13] Z. He, F. Peters, T. Menzies, and Y. Yang, “Learning from open-source projects: An empirical study on defect prediction,” in *ACM IEEE International Symposium on Empirical Software Engineering and Measurement*, 2013.

[HS97] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[HTH$^+$16] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, “Usage, costs, and benefits of continuous integration in open-source projects,” in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016, pp. 426–437.

[HTL19] Z. Hu, J. Tu, and B. Li, “Spear: Optimized dependency-aware task scheduling with deep reinforcement learning,” in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, 2019, pp. 2037–2046.

[Jac03] J. E. Jackson, *A User’s Guide to Principal Components*, ser. Wiley Series in Probability and Statistics. Wiley-Interscience, Sep. 2003.

[JC10] B. Jiang and W. Chan, “On the integration of test adequacy, test case prioritization, and statistical fault localization,” in *2010 10th International Conference on Quality Software*, 2010, pp. 377–384.

[JC16a] ——, “Testing and debugging in continuous integration with budget quotas on test executions,” in *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, 2016, pp. 439–447.

[JC16b] I. T. Jolliffe and J. Cadima, “Principal component analysis: a review and recent developments.” *Philosophical transactions. Series A, Mathematical, physical, and engineering sciences*, vol. 374 2065, p. 20150202, 2016.

[JCFdW16] J. P. Joost C. F. de Winter, Samuel D. Gosling, “Comparing the pearson and spearman correlation coefficients across distributions and sample sizes: A tutorial using simulations and empirical data,” *American Psychological Association*, 2016.

[JCT11] B. Jiang, W. Chan, and T. Tse, “On practical adequate test suites for integrated test case prioritization and fault localization,” in *2011 11th International Conference on Quality Software*, 2011, pp. 21–30.

[JG06] D. Jeffrey and N. Gupta, “Test case prioritization using relevant slices,” in *30th Annual International Computer Software and Applications Conference (COMPSAC’06)*, vol. 1, 2006, pp. 411–420.

[JH03] J. Jones and M. Harrold, “Test-suite reduction and prioritization for modified condition/decision coverage,” *IEEE Transactions on Software Engineering*, vol. 29, no. 3, pp. 195–209, 2003.

[Joa02] T. Joachims, “Optimizing search engines using clickthrough data,” in *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. Association for Computing Machinery, 2002, pp. 133–142.

[JTK13] T. Jiang, L. Tan, and S. Kim, “Personalized defect prediction,” in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, 2013.

[JYZ$^+$14] X.-Y. Jing, S. Ying, Z.-W. Zhang, S.-S. Wu, and J. Liu, “Dictionary learning based software defect prediction,” in *Proceedings of the 36th International Conference on Software Engineering*, 2014.

[JZC$^+$12] B. Jiang, Z. Zhang, W. K. Chan, T. H. Tse, and T. Y. Chen, “How well does test case prioritization integrate with statistical fault localization?” *Inf. Softw. Technol.*, pp. 739–758, 2012.

[JZTC09] B. Jiang, Z. Zhang, T. H. Tse, and T. Y. Chen, "How well do test case prioritization techniques support statistical fault localization," in *2009 33rd Annual IEEE International Computer Software and Applications Conference*, vol. 1, 2009, pp. 99–106.

[KAG+96] T. M. Khoshgoftaar, E. B. Allen, N. Goel, A. Nandi, and J. McMullan, "Detection of software modules with high debug code churn in a very large legacy system," in *Proceedings of the The Seventh International Symposium on Software Reliability Engineering*, 1996.

[Kan97] C. Kaner, "Improving the maintainability of automated test suites," *International Software Quality Week*, 1997.

[KB10] S. Kim and J. Baik, "An effective fault aware test case prioritization by incorporating a fault localization technique," in *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement.* Association for Computing Machinery, 2010.

[KDea17] E. B. Khalil, B. Dilkina, and et al., "Learning to run heuristics in tree search," in *IJCAI*, 2017.

[KFM+16] Y. Kamei, T. Fukushima, S. Mcintosh, K. Yamashita, N. Ubayashi, and A. E. Hassan, "Studying just-in-time defect prediction using cross-project models," *Empirical Softw. Engg.*, 2016.

[KGH+93] D. C. Kung, J. Gao, P. Hsia, J. Lin, and Y. Toyoshima, "Class firewall, test order, and regression testing of object-oriented programs," 1993.

[KIL21] A. Kramer, M. Iori, and P. Lacomme, "Mathematical formulations for scheduling jobs on identical parallel machines with family setup times and total weighted completion time minimization," *EJOR*, vol. 289, no. 3, pp. 825–840, 2021.

[KJL15] A. Karpathy, J. Johnson, and F. F. Li, "Visualizing and understanding recurrent networks," *Cornell Univ. Lab.*, 2015.

[KJL17] J. Kim, H. Jeong, and E. Lee, "Failure history data-based test case prioritization for effective regression test," in *Proceedings of the Symposium on Applied Computing.* New York, NY, USA: Association for Computing Machinery, 2017, pp. 1409–1415.

[KK09] B. Korel and G. Koutsogiannakis, "Experimental comparison of code-based and model-based test prioritization," in *2009 International Conference on Software Testing, Verification, and Validation Workshops*, 2009, pp. 77–84.

[KK17] J.-H. Kwon and I.-Y. Ko, "Cost-effective regression testing using bloom filters in continuous integration development environments," in *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, 2017, pp. 160–168.

[KMB17] P. Kandil, S. Moussa, and N. Badr, "Cluster-based test cases prioritization and selection technique for agile regression testing," *Journal of Software: Evolution and Process*, vol. 29, no. 6, p. e1794, 2017.

[KML93] T. M. Khoshgoftaar, J. C. Munson, and D. L. Lanning, "A comparative study of predictive models for program changes during system testing and maintenance," in *Proceedings of the Conference on Software Maintenance*, ser. ICSM '93. Washington, DC, USA: IEEE Computer Society, 1993.

[KMT07] B. A. Kitchenham, E. Mendes, and G. H. Travassos, "Cross versus within-company cost estimation studies: A systematic review," *IEEE Trans. Softw. Eng.*, 2007.

[KP02] 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*, ser. ICSE '02. Association for Computing Machinery, 2002, pp. 119–129.

[KQ19] Z. Khalid and U. Qamar, "Weight and cluster based test case prioritization technique," in *2019 IEEE 10th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*, 2019, pp. 1013–1022.

[KS02] T. M. Khoshgoftaar and N. Seliya, "Tree-based software quality estimation models for fault prediction," in *Proceedings of the 8th International Symposium on Software Metrics*, 2002.

[KS06] L. Kocsis and C. Szepesvári, "Bandit based monte-carlo planning," in *ECML*. Springer, 2006, pp. 282–293.

[KSA+13] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *IEEE Trans. Softw. Eng.*, 2013.

[KSM+15] E. Knauss, M. Staron, W. Meding, O. Śoder, A. Nilsson, and M. Castell, "Supporting continuous integration by code-churn based test selection," in *2015 IEEE/ACM 2nd International Workshop on Rapid Continuous Software Engineering*, 2015, pp. 19–25.

[KTH05] B. Korel, L. Tahat, and M. Harman, "Test prioritization using system models," in *21st IEEE International Conference on Software Maintenance (ICSM'05)*, 2005, pp. 559–568.

[KvHW19] W. Kool, H. van Hoof, and M. Welling, "Attention, learn to solve routing problems!" in *ICLR*, 2019.

[KWZ08] S. Kim, E. J. Whitehead, and Y. Zhang, "Classifying software changes: Clean or buggy?" *IEEE Trans. Softw. Eng.*, 2008.

[KZWJZ07] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller, "Predicting faults from cached history," in *Proceedings of the 29th International Conference on Software Engineering*, 2007.

[Lan08] Z. T. Langford, J., "The epoch-greedy algorithm for multi-armed bandits with side information." NIPS 2008, 2008, pp. 817–824.

[LCK18] Z. Li, Q. Chen, and V. Koltun, "Combinatorial optimization with graph convolutional networks and guided tree search," in *NeurIPS*, 2018.

[LER18] J. Liang, S. Elbaum, and G. Rothermel, "Redefining prioritization: Continuous prioritization for continuous integration," in *Proceedings of the 40th International Conference on Software Engineering*. Association for Computing Machinery, 2018, pp. 688–698.

[LFJ+18] A. Laterre, Y. Fu, M. K. Jabri, A. Cohen, D. Kas, K. Hajjar, T. S. Dahl, A. Kerkeni, and K. Beguir, "Ranked reward: Enabling self-play reinforcement learning for combinatorial optimization," *CoRR*, vol. abs/1807.01672, 2018.

[LHH07] Z. Li, M. Harman, and R. M. Hierons, "Search algorithms for regression test case prioritization," *IEEE Transactions on Software Engineering*, pp. 225–237, 2007.

[LHS+16] O. Legunsen, F. Hariri, A. Shi, Y. Lu, L. Zhang, and D. Marinov, "An extensive study of static regression test selection in modern software evolution," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2016, pp. 583–594.

[LHZL17] J. Li, P. He, J. Zhu, and M. R. Lyu, "Software defect prediction via convolutional neural network," in *IEEE International Conference on Software Quality, Reliability and Security (QRS)*, 2017.

[Li14] H. Li, "Learning to rank for information retrieval and natural language processing." Morgan and Claypool Publishers, 2014.

[LLH$^+$16] M. Laali, H. Liu, M. Hamilton, M. Spichkova, and H. W. Schmidt, "Test case prioritization using online fault detection information," in *Reliable Software Technologies-Ada-Europe 2016*, M. Bertogna, L. M. Pinho, and E. Quiñones, Eds. Cham: Springer International Publishing, 2016, pp. 78–93.

[LMVAa20] J. A. P. Lima, W. D. F. Mendonça, S. R. Vergilio, and W. K. G. Assunção, "Learning-based prioritization of test cases in continuous integration of highly-configurable software," in *Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A - Volume A*. Association for Computing Machinery, 2020.

[Lom65] Z. A. Lomnicki, ""branch-and-bound" algorithm for the exact solution of the three-machine scheduling problem." J Oper Res Soc 16, 1965, pp. 89–100.

[LSN$^+$16] R. Lachmann, S. Schulze, M. Nieke, C. Seidl, and I. Schaefer, "System-level test case prioritization using machine learning," in *2016 15th IEEE International Conference on Machine Learning and Applications (ICMLA)*, 2016, pp. 361–368.

[LV20] J. A. P. Lima and S. R. Vergilio, "Multi-armed bandit test case prioritization in continuous integration environments: A trade-off analysis," in *Proceedings of the 5th Brazilian Symposium on Systematic and Automated Software Testing*. Association for Computing Machinery, 2020, pp. 21–30.

[LZY20] H. Lu, X. Zhang, and S. Yang, "A learning-based iterative method for solving vehicle routing problems," in *ICLR*, 2020.

[MAea16] H. Mao, M. Alizadeh, and et al., "Resource management with deep reinforcement learning," in *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, 2016.

[Mar15] D. Marijan, "Multi-perspective regression test prioritization for time-constrained environments," in *2015 IEEE International Conference on Software Quality, Reliability and Security*, 2015, pp. 157–162.

[MAT12] S. Mirarab, S. Akhlaghi, and L. Tahvildari, "Size-constrained regression test case selection using multicriteria optimization," *IEEE Transactions on Software Engineering*, vol. 38, no. 4, pp. 936–956, 2012.

[MB16] B. Miranda and A. Bertolino, "Does code coverage provide a good stopping rule for operational profile based testing?" in *2016 IEEE/ACM 11th International Workshop in Automation of Software Test (AST)*, 2016, pp. 22–28.

[MB17] ——, "Scope-aided test prioritization, selection and minimization for software reuse," *Journal of Systems and Software*, vol. 131, pp. 528–549, 2017.

[MBdNS17] J. S. Madureira, A. S. Barroso, R. P. C. do Nascimento, and M. S. Soares, "An experiment to evaluate software development teams by using object-oriented metrics," in *Computational Science and Its Applications – ICCSA 2017*, O. Gervasi, B. Murgante, S. Misra, G. Borruso, C. M. Torre, A. M. A. Rocha, D. Taniar, B. O. Apduhan, E. Stankova, and A. Cuzzocrea, Eds. Springer International Publishing, 2017, pp. 128–144.

[MF19] C. Manjula and L. Florence, "Deep neural network based hybrid approach for software defect prediction using software metrics," *Cluster Computing*, 2019.

[MGL19] D. Marijan, A. Gotlieb, and M. Liaaen, "A learning algorithm for optimizing continuous integration development and testing practice," *Software: Practice and Experience*, vol. 49, pp. 192 – 213, 2019.

[MGN+17] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco, "Taming google-scale continuous testing," in *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, 2017, pp. 233–242.

[MGS13] D. Marijan, A. Gotlieb, and S. Sen, "Test case prioritization for continuous regression testing: An industrial case study," in *2013 IEEE International Conference on Software Maintenance*, 2013, pp. 540–543.

[MHZ+12] H. Mei, D. Hao, L. Zhang, L. Zhang, J. Zhou, and G. Rothermel, "A static approach to prioritizing junit test cases," *IEEE Transactions on Software Engineering*, pp. 1258–1275, 2012.

[MK90] J. Munson and T. Khoshgoftaar, "Regression modelling of software quality: empirical investigation," *Information and Software Technology*, vol. 32, no. 2, pp. 106 – 114, 1990.

[MK18] S. McIntosh and Y. Kamei, "Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction," in *Proceedings of the 40th International Conference on Software Engineering*, 2018.

[ML16] D. Marijan and M. Liaaen, "Effect of time window on the performance of continuous regression testing," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2016, pp. 568–571.

[ML17] ——, "Test prioritization with optimally balanced configuration coverage," in *2017 IEEE 18th International Symposium on High Assurance Systems Engineering (HASE)*, 2017, pp. 100–103.

[MLG+17] D. Marijan, M. Liaaen, A. Gotlieb, S. Sen, and C. Ieva, "Titan: Test suite optimization for highly configurable software," in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2017, pp. 524–531.

[MLS18] D. Marijan, M. Liaaen, and S. Sen, "Devops improvements for reduced cycle times with integrated test optimizations for continuous integration," in *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 01, 2018, pp. 22–27.

[MLZ+16] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[MMBT20] N. Medhat, S. M. Moussa, N. L. Badr, and M. F. Tolba, "A framework for continuous regression and integration testing in iot systems based on deep learning and search-based techniques," *IEEE Access*, vol. 8, pp. 215 716–215 726, 2020.

[MMHM21] M. Mahdieh, S.-H. Mirian-Hosseinabadi, and M. Mahdieh, "Test case prioritization using test case diversification and fault-proneness estimations," 2021.

[MMT+10] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener, "Defect prediction from static code features: Current results, limitations, new approaches," *Autom. Softw. Eng.*, 2010.

[MSPC19] M. Machalica, A. Samylkin, M. Porth, and S. Chandra, "Predictive test selection," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2019, pp. 91–100.

[MSV$^+$19] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh, "Learning scheduling algorithms for data processing clusters," in *Proceedings of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 270–288.

[MW00] A. Mockus and D. M. Weiss, "Predicting risk of software changes," *Bell Labs Technical Journal*, 2000.

[NB05a] N. Nagappan and T. Ball, "Static analysis tools as early indicators of pre-release defect density," in *Proceedings of the 27th International Conference on Software Engineering*, ser. ICSE '05. New York, NY, USA: ACM, 2005.

[NB05b] ——, "Use of relative code churn measures to predict system defect density," in *Proceedings of the 27th International Conference on Software Engineering*, 2005.

[NBZ06] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *Proceedings of the 28th International Conference on Software Engineering*, 2006.

[NH15] T. B. Noor and H. Hemmati, "A similarity-based approach for test case prioritization using historical failure data," in *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, 2015, pp. 58–68.

[NNP11] T. T. Nguyen, T. N. Nguyen, and T. M. Phuong, "Topic-based defect prediction (nier track)," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011.

[Nor98] J. R. Norris, "Markov chain." Cambridge Series in Statistical and Probabilistic Mathematics, 1998.

[NOST18] M. Nazari, A. Oroojlooy, L. V. Snyder, and M. Takác, "Reinforcement learning for solving the vehicle routing problem," in *NeurIPS*, 2018.

[NPK13] J. Nam, S. J. Pan, and S. Kim, "Transfer defect learning," in *Proceedings of the International Conference on Software Engineering*, 2013.

[NSR19] A. Najafi, W. Shang, and P. C. Rigby, "Improving test effectiveness using test executions history: An industrial experience report," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2019, pp. 213–222.

[NWH$^+$04] N. Nagappan, L. Williams, J. Hudepohl, W. Snipes, and M. Vouk, "Preliminary results on using static analysis tools for software inspection," in *Proceedings of the 15th International Symposium on Software Reliability Engineering*, ser. ISSRE '04. Washington, DC, USA: IEEE Computer Society, 2004.

[OMS18] S. Omri, P. Montag, and C. Sinz, "Static analysis and code complexity metrics as early indicators of software defects," *Journal of Software Engineering and Applications*, 2018.

[OSM19] S. Omri, C. Sinz, and P. Montag, "An enhanced fault prediction model for embedded software based on code churn, complexity metrics, and static analysis results." ICSEA 2019 : The Fourteenth International Conference on Software Engineering Advances, 2019.

[OvMMW] M. C. Ohlsson, A. von Mayrhauser, B. McGuire, and C. Wohlin, "Code decay analysis of legacy software through successive releases," in *1999 IEEE Aerospace Conference. Proceedings (Cat. No.99TH8403).*

[PBGB21] R. Pan, M. Bagherzadeh, T. A. Ghaleb, and L. C. Briand, "Test case selection and prioritization using machine learning: A systematic literature review," *CoRR*, vol. abs/2106.13891, 2021.

[PCA⁺19] D. Paterson, J. Campos, R. Abreu, G. M. Kapfhammer, G. Fraser, and P. McMinn, "An empirical study on the use of defect prediction for test case prioritization," in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, 2019, pp. 346–357.

[PDS⁺15] H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, and Y. Acar, "Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[PLV20] J. A. Prado Lima and S. R. Vergilio, "A multi-armed bandit approach for test case prioritization in continuous integration environments," *IEEE Transactions on Software Engineering*, pp. 1–1, 2020.

[pms] "Pmsp reference benchmarks and branch and bound results," http://www.or.unimore.it/site/home/online-resources/identical-parallel-machines-scheduling-problem-with-family-setup-times.html.

[PNB18] A. Phan, L. Nguyen, and L. Bui, "Convolutional neural networks over control flow graphs for software defect prediction," 2018.

[PP14] L. Prechelt and A. Pepper, "Why software repositories are not used for defect-insertion circumstance analysis more often: A case study," *Inf. Softw. Technol.*, 2014.

[PP21] C. Pan and M. Pradel, "Continuous test suite failure prediction," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. Association for Computing Machinery, 2021, pp. 553–565.

[PRB08] 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 *2008 Second International Conference on Secure System Integration and Reliability Improvement*, 2008, pp. 39–46.

[Put94] M. L. Puterman, *Markov Decision Processes*. Wiley and Sons, 1994.

[PWA⁺19] D. Pradhan, S. Wang, S. Ali, T. Yue, and M. Liaaen, "Employing rule mining and multi-objective search for dynamic test case prioritization," *Journal of Systems and Software*, vol. 153, pp. 86–104, 2019.

[PXN13] Y. Pang, X. Xue, and A. S. Namin, "Identifying effective test cases through k-means clustering for enhancing regression testing," in *2013 12th International Conference on Machine Learning and Applications*, vol. 2, 2013, pp. 78–83.

[RBB11] R. Reussner, S. Becker, and E. Burger, "The palladio component model," *Theoretical Computer Science*, 2011. [Online]. Available: https://core.ac.uk/download/pdf/197552587.pdf

[RH97] G. Rothermel and M. J. Harrold, "A safe, efficient regression test selection technique," *ACM Trans. Softw. Eng. Methodol.*, pp. 173–210, 1997.

[RH98] G. Rothermel and M. J. Harrold, "Empirical studies of a safe regression test selection technique," *IEEE Transactions on Software Engineering*, pp. 401–419, 1998.

[RJS17] A. Radford, R. Jozefowicz, and I. Sutskever, "Learning to generate reviews and discovering sentiment," 2017.

[ROC] "Receiver operating characteristic (roc)," https://scikit-learn.org/0.15/auto_examples/plot_roc.html.

[RPR13] A. Rafael Lenz, A. Pozo, and S. Regina Vergilio, “Linking software testing results with a machine learning approach,” *Engineering Applications of Artificial Intelligence*, vol. 26, no. 5, pp. 1631–1640, 2013.

[RSHN14] R. Rana, M. Staron, J. Hansson, and M. Nilsson, “Defect prediction over software life cycle in automotive domain state of the art and road map for future,” in *9th International Conference on Software Engineering and Applications (ICSOFT-EA)*, 2014.

[RSM$^+$20] L. Rosenbauer, A. Stein, R. Maier, D. Pätzel, and J. Hähner, “Xcs as a reinforcement learning approach to automatic test case prioritization,” in *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion.* Association for Computing Machinery, 2020, pp. 1798–1806.

[RST$^+$04] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley, “Chianti: A tool for change impact analysis of java programs,” *SIGPLAN Not.*, pp. 432–448, 2004.

[RT01] B. Ryder and F. Tip, “Change impact analysis for object-oriented programs,” in *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering.* Association for Computing Machinery, 2001, pp. 46–53.

[RUCH99] G. Rothermel, R. Untch, C. Chu, and M. Harrold, “Test case prioritization: an empirical study,” in *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM’99). ’Software Maintenance for Business Change’ (Cat. No.99CB36360)*, 1999, pp. 179–188.

[RUCH01] G. Rothermel, R. H. Untch, Chengyun Chu, and M. J. Harrold, “Prioritizing test cases for regression testing,” *IEEE Transactions on Software Engineering*, pp. 929–948, 2001.

[SB18] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction.* MIT press, 2018.

[SBB$^+$02] F. Shull, V. Basili, B. Boehm, A. Brown, P. Costa, M. Lindvall, D. Port, I. Rus, R. Tesoriero, and M. Zelkowitz, “What we have learned about fighting defects,” in *Proceedings Eighth IEEE Symposium on Software Metrics*, 2002, pp. 249–258.

[Sch92] N. F. Schneidewind, “Methodology for validating software metrics,” *IEEE Trans. Softw. Eng.*, vol. 18, 1992.

[SD17] S. K. Silver D., Schrittwieser J., “Mastering the game of go without human knowledge,” *Nature 550*, pp. 354–359, 2017.

[SDG$^+$16] T. Savor, M. Douglas, M. Gentili, L. Williams, K. Beck, and M. Stumm, “Continuous deployment at facebook and oanda,” in *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, 2016, pp. 21–30.

[SGMM17] 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*, ser. ISSTA 2017, 2017, pp. 12–22.

[SGW$^+$20] P. Sun, Z. Guo, J. Wang, J. Li, J. Lan, and Y. Hu, “Deepweave: Accelerating job completion time with deep reinforcement learning-based coflow scheduling,” in *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 20*, C. Bessiere, Ed. International Joint Conferences on Artificial Intelligence Organization, 7 2020, pp. 3314–3320, main track.

[SHM$^+$16] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, “Mastering the game of go with deep neural networks and tree search,” *Nature*, vol. 529, pp. 484–503, 2016.

[SJLS00] S. Singh, T. Jaakkola, M. L. Littman, and C. Szepesvári, "Convergence results for single-step on-policyreinforcement-learning algorithms," *Mach. Learn.*, vol. 38, pp. 287–308, 2000.

[SK03] R. Subramanyam and M. S. Krishnan, "Empirical analysis of CK metrics for object-oriented design complexity: Implications for software defects," *IEEE Trans. Softw. Eng.*, 2003.

[SLB$^+$18] D. Selsam, M. Lamm, B. Bünz, P. Liang, L. de Moura, and D. L. Dill, "Learning a SAT solver from single-bit supervision," *CoRR*, vol. abs/1802.03685, 2018.

[SMPB11] L. S. Souza, P. B. C. Miranda, R. B. C. Prudencio, and F. d. A. Barros, "A multi-objective particle swarm optimization for test case selection based on functional requirements coverage and execution effort," in *2011 IEEE 23rd International Conference on Tools with Artificial Intelligence*, 2011, pp. 245–252.

[SPVV17] P. Singh, N. R. Pal, S. Verma, and O. P. Vyas, "Fuzzy rule-based approach for software fault prediction," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 47, no. 5, pp. 826–837, May 2017.

[SR05] M. Skoglund and P. Runeson, "A case study of the class firewall regression test selection technique on a large scale distributed software system," in *International Symposium on Empirical Software Engineering*, 2005, p. 10.

[SR07] ——, "Improving class firewall regression test selection by removing the class firewall," in *JSEKE*, 2007.

[SSA$^+$16] P. E. Strandberg, D. Sundmark, W. Afzal, T. J. Ostrand, and E. J. Weyuker, "Experience report: Automated system level regression test prioritization using multiple factors," in *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, 2016, pp. 12–23.

[SSSJ17] D. Singh, V. R. Sekar, K. T. Stolee, and B. Johnson, "Evaluating how static analysis tools can reduce code review effort," in *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Oct 2017, pp. 101–105.

[SW05] H. Srikanth and L. Williams, "On the economics of requirements-based test case prioritization," in *Proceedings of the Seventh International Workshop on Economics-Driven Software Engineering Research*. Association for Computing Machinery, 2005, pp. 1–3.

[SWO05] H. Srikanth, L. Williams, and J. Osborne, "System test case prioritization of new and regression test cases," in *2005 International Symposium on Empirical Software Engineering, 2005.*, 2005, p. 10.

[SXW20] T. Shi, L. Xiao, and K. Wu, "Reinforcement learning based test case prioritization for enhancing the security of software," in *2020 IEEE 7th International Conference on Data Science and Advanced Analytics (DSAA)*, 2020, pp. 663–672.

[SZKP15] R. K. Saha, L. Zhang, S. Khurshid, and D. E. Perry, "An information retrieval approach for regression test prioritization based on program changes," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, 2015, pp. 268–279.

[TAS06] P. Tonella, P. Avesani, and A. Susi, "Using the case-based ranking methodology for test case prioritization," in *2006 22nd IEEE International Conference on Software Maintenance*, 2006, pp. 123–133.

[Tho14] H. H. H. A. e. a. Thomas, S.W., "Static test case prioritization using topic models." Empir Software Eng 19, 2014, pp. 182–212.

[TKC99] M.-H. Tang, M.-H. Kao, and M.-H. Chen, "An empirical study on object-oriented metrics," in *Proceedings of the 6th International Symposium on Software Metrics*, 1999.

[TLW17] H. Tong, B. Liu, and S. Wang, "Software defect prediction using stacked denoising autoencoders and two-stage ensemble learning," *Information and Software Technology*, 2017.

[TMBDS09] B. Turhan, T. Menzies, A. B. Bener, and J. Di Stefano, "On the relative value of cross-company and within-company data for defect prediction," *Empirical Softw. Engg.*, 2009.

[TTDM15] M. Tan, L. Tan, S. Dara, and C. Mayeux, "Online defect prediction for imbalanced data," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, 2015, pp. 99–108.

[VFJ15] O. Vinyals, M. Fortunato, and N. Jaitly, "Pointer networks," in *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'15. Cambridge, MA, USA: MIT Press, 2015, pp. 2692–2700.

[WCF+12] Y. Wang, Z. Chen, Y. Feng, B. Luo, and Y. Yang, "Using weighted attributes to improve cluster test selection," in *2012 IEEE Sixth International Conference on Software Security and Reliability*, 2012, pp. 138–146.

[WD92] C. J. C. H. Watkins and P. Dayan, "Q-learning," in *Machine Learning*, 1992, pp. 279–292.

[Wil95] S. W. Wilson, "Classifier fitness based on accuracy," *Evolutionary Computation*, vol. 3, no. 2, pp. 149–175, 1995.

[Wis05] C. Wissler, "The spearman correlation formula," *Science*, vol. 22, no. 558, pp. 309–311, 1905.

[WL] T. Wang and W. Li, "Naive bayes software defect prediction model," in *2010 International Conference on Computational Intelligence and Software Engineering*.

[WLNT18] S. Wang, T. Liu, J. Nam, and L. Tan, "Deep semantic feature learning for software defect prediction," *IEEE Transactions on Software Engineering*, 2018.

[WLT16] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proceedings of the 38th International Conference on Software Engineering*, 2016.

[WNT17] S. Wang, J. Nam, and L. Tan, "Qtep: Quality-aware test case prioritization," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017, 2017, pp. 523–534.

[WRB+18] B. Waschneck, A. Reichstaller, L. Belzner, T. Altenmüller, T. Bauernhansl, A. Knapp, and A. Kyek, "Optimization of global production scheduling with deep reinforcement learning," *Procedia CIRP*, vol. 72, pp. 1264–1269, 2018.

[WS11] D. P. Williamson and D. B. Shmoys, *The Design of Approximation Algorithms*. Cambridge university press, 2011.

[WSC] J. Wang, B. Shen, and Y. Chen, "Compressed c4.5 models for software defect prediction," in *Proceedings of the 2012, 12th International Conference on Quality Software.*

[WSC+19] Y. Wu, W. Song, Z. Cao, J. Zhang, and A. Lim, "Learning improvement heuristics for solving the travelling salesman problem," *ArXiv*, vol. abs/1912.05784, 2019.

[WSKR06] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos, "Timeaware test suite prioritization," in *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, ser. ISSTA '06. Association for Computing Machinery, 2006, pp. 1–12.

[WYY18] W. Wen, Z. Yuan, and Y. Yuan, "Improving retecs method using fp-growth in continuous integration," in *2018 5th IEEE International Conference on Cloud Computing and Intelligence Systems (CCIS)*, 2018, pp. 636–639.

[WZ18] J. Wang and C. Zhang, "Software reliability prediction using a deep learning model based on the RNN encoder-decoder," *Reliab. Eng. Syst. Saf.*, 2018.

[XHLJ19] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" 2019.

[XLWY16] X. Xia, D. Lo, X. Wang, and X. Yang, "Collective personalized change classification with multiobjective search," *IEEE Transactions on Reliability*, 2016.

[XMZ18] L. Xiao, H. Miao, and Y. Zhong, "Test case prioritization and selection technique in continuous integration development environments: a case study," *International journal of engineering and technology*, vol. 7, p. 332, 2018.

[XZYW12] X. Xie, W. Zhang, Y. Yang, and Q. Wang, "Dretom: Developer recommendation based on topic models for bug resolution," in *Proceedings of the 8th International Conference on Predictive Models in Software Engineering*, 2012.

[YFM$^+$19] Z. Yu, F. Fahid, T. Menzies, G. Rothermel, K. Patrick, and S. Cherian, "Terminator: Better automated ui test case prioritization," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. Association for Computing Machinery, 2019, pp. 883–894.

[YFRJ07] Y. Yue, T. Finley, F. Radlinski, and T. Joachims, "A support vector method for optimizing average precision," in *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval.* Association for Computing Machinery, 2007, pp. 271–278.

[YH12] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," in *STVR'12*, 2012, pp. 67–120.

[YHTS09] S. Yoo, M. Harman, P. Tonella, and A. Susi, "Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge," in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis.* Association for Computing Machinery, 2009, pp. 201–212.

[Yim99] L. D. Yim, S.J., "Scheduling cluster tools in wafer fabrication using candidate list and simulated annealing," *Journal of Intelligent Manufacturing*, vol. 10, pp. 531–540, 1999.

[YJ09] Y. Yue and T. Joachims, "Interactively optimizing information retrieval systems as a dueling bandits problem," in *Proceedings of the 26th Annual International Conference on Machine Learning*, ser. ICML '09. Association for Computing Machinery, 2009, pp. 1201–1208.

[YLX$^+$15] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun, "Deep learning for just-in-time defect prediction," in *Proceedings of the IEEE International Conference on Software Quality, Reliability and Security*, 2015.

[YNH11] S. Yoo, R. Nilsson, and M. Harman, "Faster fault finding at google using multi objective regression test optimisation," 2011.

[Yog12] P. S. G. Yogeswaran, Mohan, "Reinforcement learning: exploration-exploitation dilemma in multi-agent foraging task," in *OPSEARCH*, 2012.

[YP19] E. Yolcu and B. Poczos, "Learning local search heuristics for boolean satisfiability," in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32. Curran Associates, Inc., 2019.

[YY17] H. B. Yadav and D. K. Yadav, "Early software reliability analysis using reliability relevant software metrics," *International Journal of System Assurance Engineering and Management*, vol. 8, no. 4, pp. 2097–2108, Dec 2017.

[ZGS19] S. Zheng, C. Gupta, and S. Serita, "Manufacturing dispatching using reinforcement and transfer learning," *CoRR*, vol. abs/1910.02035, 2019.

[Zha19] Z.-Y. S. X. e. a. Zhang, C., "On incremental learning for gradient boosting decision trees." Neural Process Lett 50, 2019, pp. 957–987.

[Zho21] Z.-H. Zhou, *Ensemble Learning*. Singapore: Springer Singapore, 2021, pp. 181–210.

[ZKK12] L. Zhang, M. Kim, and S. Khurshid, "Faulttracer: A change impact and regression fault analysis tool for evolving java programs," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12, 2012.

[ZNG$^+$09] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, "Cross-project defect prediction: A large scale experiment on data vs. domain vs. process," in *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT*, 2009.

[ZPD20] R. Zhang, A. Prokhorchuk, and J. Dauwels, "Deep reinforcement learning for traveling salesman problem with time windows and rejections," *2020 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8, 2020.

[ZPZ07] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," in *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, 2007.

[ZQV18] L. Zhuwen, C. Qifeng, and K. Vladlen, "Combinatorial optimization with graph convolutional networks and guided tree search," in *NeurIPS*, 2018.

[ZSC$^+$20] C. Zhang, W. Song, Z. Cao, J. Zhang, P. S. Tan, and C. Xu, "Learning to dispatch for job shop scheduling via deep reinforcement learning," *CoRR*, vol. abs/2010.12367, 2020.

[ZSR18] Y. Zhu, E. Shihab, and P. C. Rigby, "Test re-prioritization in continuous testing environments," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2018, pp. 69–79.