

Prioritizing Tests with Spotify's Test & Build Data using History- based, Modification-based & Machine Learning Approaches

Testprioritering med Spotifys test- & byggdata genom historik-
baserade, modifikationsbaserade & maskininlärningsbaserade
metoder

Petra Öhlin

Supervisor : Cyrille Berger
Examiner : Ola Leifler

Upphovsrätt

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår. Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns lösningar av teknisk och administrativ art. Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart. För ytterligare information om Linköping University Electronic Press se förlagets hemsida <http://www.ep.liu.se/>.

Copyright

The publishers will keep this document online on the Internet – or its possible replacement – for a period of 25 years starting from the date of publication barring exceptional circumstances. The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/hers own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility. According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement. For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: <http://www.ep.liu.se/>.

Abstract

This thesis intends to determine the extent to which machine learning can be used to solve the regression test prioritization (RTP) problem. RTP is used to order tests with respect to probability of failure. This will optimize for a fast failure, which is desirable if a test suite takes a long time to run or uses a significant amount of computational resources. A common machine learning task is to predict probabilities; this makes RTP an interesting application of machine learning. A supervised learning method is investigated to train a model to predict probabilities of failure, given a test case and a code change. The features investigated are chosen based on previous research of history-based and modification-based RTP. The main motivation for looking at these research areas is that they resemble the data provided by Spotify. The result of the report shows that it is possible to improve how tests run with RTP using machine learning. Nevertheless, a much simpler history-based approach is the best performing approach. It is looking at the history of test results, the more failures recorded for the test case over time, the higher priority it gets. Less is sometimes more.

Acknowledgments

Thanks to all employees at Spotify that supported and helped out during the project. The team I was working in, **Client Build**, that welcomed me as a part of the team, helped me out whenever needed and gave me valuable feedback for the project and the report. Thanks to my manager **Fredrik Stridsman**, for understanding my needs and encouraging me to focus on the right things. Thanks to my project supervisor **Laurent Ploix**, for being passionate about the project. Thanks to all other amazing people I met during the time. Thanks for providing data, sharing ideas and for all the support.

Thanks to my university supervisor **Cyrille Berger**, and examiner **Ola Leifler**, for keeping the project well structured and making sure that I continuously made progress. Thanks to **Mattias Palmgren**, **Johan Henriksson**, **Conor Taylor**, **Oskar Werkelin Ahlin** and **Ola Jigin** for valuable reviews. Thank you for honestly questioning the reasoning, encouraging comments and polishing the language, taking the report to the next level.

Contents

Abstract	iii
Acknowledgments	iv
Contents	v
List of Figures	vii
List of Tables	viii
List of Algorithms	1
1 Introduction	2
1.1 Motivation	2
1.2 Aim	4
1.3 Research questions	4
1.4 Delimitations	4
1.5 Outline	5
2 Theory	6
2.1 Continuous Integration and Regression Testing	6
2.2 Definition of Test Prioritization	7
2.3 Baseline approaches	7
2.4 Coverage-based approaches	8
2.5 History-based approaches	8
2.6 Modification-based approaches	9
2.7 Machine Learning	10
2.8 Test Prioritization in an Industrial Environment	14
2.9 Metrics	14
3 Method	17
3.1 Data at Spotify	17
3.2 Data set	17
3.3 Framework for Evaluating Prioritization Approaches	19
3.4 Baseline approaches	22
3.5 History-based approaches	22
3.6 Modification-based approaches	23
3.7 Machine learning approaches	25
4 Results	29
4.1 APFD	30
4.2 Time & Rank	31
5 Discussion	33

5.1	Results	33
5.2	Method	34
5.3	The work in a wider context	36
6	Conclusion	38
6.1	Research questions	38
6.2	Future work	40
	Bibliography	41

List of Figures

1.1	Spotify applications	3
2.1	An example of a Maximal Margin Classifier in two dimensions	11
2.2	Decision tree example	13
3.1	ER-Diagram of SQLite database	18
3.2	Version history in practice	18
3.3	High level overview of the evaluation system	20
3.4	Distribution of path similarity scores	25
4.1	APFD scores of approaches	30
4.2	APFD scores of recent failure history approaches	30
4.3	APFD scores of failure history approaches	31
4.4	APFD scores of machine learning approaches	31
4.5	Time to first failure	32
4.6	Rank of first failure	32

List of Tables

2.1	Visualization of test execution and corresponding faults	15
2.2	Classification outcomes	15
3.1	List of machine learning approaches, abbreviations and corresponding feature sets	27
4.1	List of approaches and abbreviations	29
6.1	All approaches effect on APFD values	39
6.2	Different configurations of the classification approach's effect on APFD values	39

List of Algorithms

1	History-based algorithm: LastExecutionAndLastFailure	23
2	Modification-based algorithm: NameSimilarity	24
3	Machine Learning algorithm: ClassificationApproach/RegressionApproach	28



1 Introduction

This chapter will introduce the work by describing the motivation, problem and aim behind the thesis. Based on this, research questions are formulated that are answered in Chapter 6.

1.1 Motivation

1.1.1 Context

Continuous Integration (CI) is a software development practice which aims to have a releasable product at all times. In practice, this means developers integrate their work frequently. Each integration is verified by an automated build to detect integration errors as quickly as possible. The test activity performed in this process is called regression testing. For every change in the existing software, the newly introduced changes are tested so that they do not obstruct the behaviors of the existing, unchanged part of the software.

Regression testing can be time-consuming and expensive, [1, 36], therefore early fault detection is highly desirable. Regression test prioritization (RTP) is a widely studied field that aims to achieve this. In general, RTP seeks to find the optimal permutation of the tests with respect to some objective, a common objective is probability of failure. In that case, the tests are sorted in decreasing order to increase the probability of a fast failure. The tests can be prioritized on different granularities, e.g. test case or test suite granularity.

The past decades have seen the rapid development of RTP, and a considerable amount of methods have been developed. Recent trends in machine learning have led to a proliferation of studies that apply machine learning techniques to various domains, and RTP is not an exception to the trend. This is rephrasing the problem from a prioritization problem to different machine learning tasks. As an example, it can be thought of as a classification task. Given a code change and test cases to prioritize, the test cases can be classified in two classes, success and failure, depending on the code change and the test case. Prioritization of the test cases is then performed on a class granularity, hence all tests in the failure class run first, and the tests in the success class run last. A resembling approach has been proven to be powerful by Benjamin Busjaeger and Tao Xie [4]. Using a machine learning approach enables utilization of earlier RTP research by using supervised learning techniques. In the learning phase of the machine learning model, insights from previous research can be incorporated as hypothetically well-performing features. Benjamin Busjaeger and Tao Xie use features leveraging code coverage, result

history of the test case, textual similarity between the test case and the change, and age of the test case [4], based on [30, 16, 32, 12], respectively.

1.1.2 Spotify

This research is carried out for Linköping University in collaboration with Spotify AB.

Spotify AB is a Swedish-founded music streaming company. Spotify aims to bring the right music for every moment to their costumers. The application is therefore available on various platforms including computers, mobiles, tablets, home entertainment systems, cars, gaming consoles, and more. Figure 1.1 shows how the interface of the application looks like on some of the platforms.

Spotify started out as a startup in Stockholm and the first Spotify application was launched in 2008. Since then the company has scaled and the service has currently an active user base of over 100 million users, the music catalog contains over 30 million songs and the service is available in 60 markets ¹.

The size of the company indicates that the code base is also of significant size, and that it requires a considerable amount of tests to be able to provide a reliable product.

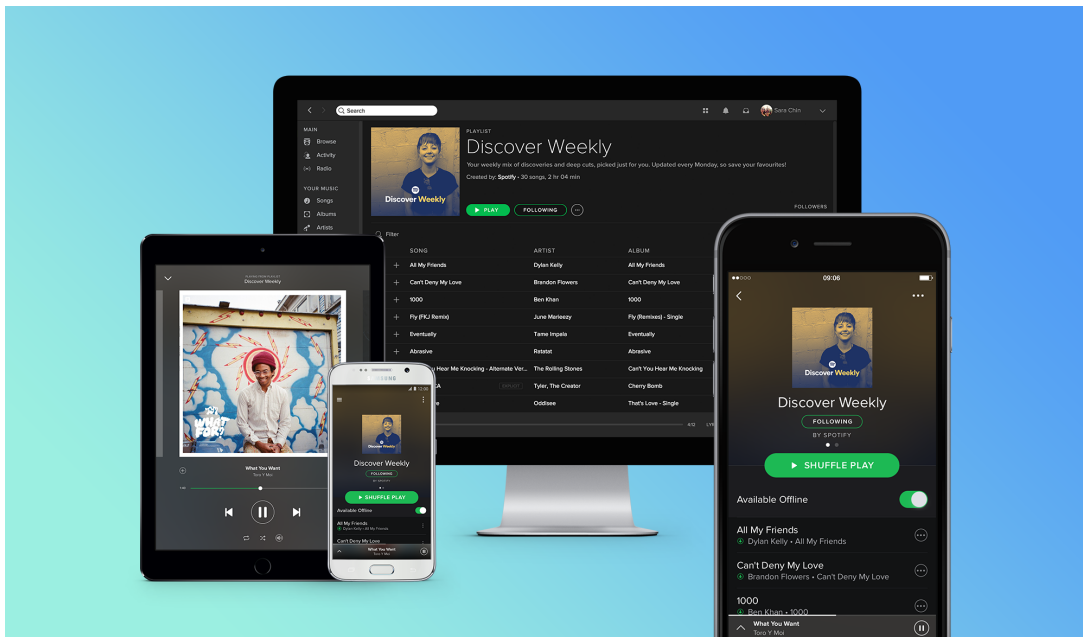


Figure 1.1: Spotify applications

1.1.3 Testing at Spotify

At Spotify, the testing pipelines differ between clients, testing the iOS client is different from the Android client, desktop client and so on. It is a complex CI system that builds and run tests distributed on different pools of build agents. For the iOS client, which is used to gather test data for this thesis, both unit and integration tests run pre- and post-merge.

Recently, Spotify has shown an increased interest in the field of test optimization by various initiatives. One initiative that aims to aid the problems with the current approach of running tests is that one team has developed a way to identify and remove flaky tests. A flaky test is a test that is nondeterministic, hence could fail or pass randomly for the same configuration. This initiative is aiding the testing environment as it increases the certainty that a failure is actually a bug. Currently developers often retrigger failed builds because of the uncertainty of the failure. These retriggers would be unnecessary

¹<https://press.spotify.com/us/about/>

if it was certain that a failure always is correlated with a bug. Another team is working on renewing the testing environment and other hobby projects have been carried out in order to research what can be done in this area.

1.1.4 Problem

Spotify has a CI environment that currently builds around 25.000 builds per day. Currently a *retest-all* approach is applied by the company, where all tests run in no particular order, for all code changes. The code base, the amount of tests and the CI environment itself is expected to grow rapidly in the coming years. With the current approach of running tests, this growth comes at a large cost in terms of increased testing time and resources.

There is an increasing concern that the problems caused by the growth of the CI system impacts the developers at the company negatively. An increase in testing time will result in an increased feedback loop duration for developers. That means, the time the developers need to wait from committing a pull request to know if the regression tests failed due to the new change. That feedback loop is preferably such a short time so that the developer should not need to switch context. This problem is also related to one of the fundamental ideas behind CI, that is, spending less time integrating different developers' code changes.

In addition, the CI environment produces a considerable amount of data, approximately 0.5 TB per day, including artifacts, logs, test results, binaries etc. This data can be utilized to make informed decisions, such as using history-based or modification-based RTP approaches that are looking at test and build data to run tests in a more efficient way.

1.2 Aim

This thesis intends to determine the extent to which the problems that emerge from this growth, described in previous Section 1.1.4, are solvable. The investigation includes prioritization techniques, and more specifically history-based, modification-based and machine learning techniques utilizing Spotify's historical build and test data.

1.3 Research questions

To date there is one study, *Learning for test prioritization: an industrial case study* [4], to the knowledge of the author, that has investigated the association between machine learning and RTP. This research sheds new light on if RTP can be thought of as a classification or regression problem instead of a learning to rank problem, and if it applies on a data set from Spotify. Commonly used metrics and baselines in the field of RTP are used to evaluate the approaches. To carry out the research, two research questions have been formulated:

1. Do history-based, modification-based and machine learning approaches affect the fault detection rate, compared to the retest-all approach used by the company today and random prioritization?
2. Do classification approaches using feature sets based on previous research in RTP affect the fault detection rate, compared to using a feature set with randomly selected sets of metadata?

1.4 Delimitations

There is a distinction in CI systems at Spotify between the back-end services and the client code. They differ in processes and systems used. This investigation focuses on the CI system of the clients, with the reason that the data set used is fetched from that CI system.

In addition, the investigation was executed with a test data set from one specific project and a specific group of tests at Spotify. These delimitations were made to simplify the implementation of the

evaluation system. Knowing what platform and group of tests to incorporate in the test data set enables assumptions to be made about the data set, e.g. which information that exists.

This study is unable to encompass the entire test optimization field, for that reason only prioritization methods are investigated. Other test optimization techniques, such as test selection and minimization, are just briefly mentioned in Chapter 2. In addition to that, only prioritization techniques that utilize some kind of historical or build data are investigated, the reason is to fulfill the aim to use the data Spotify have stored.

This thesis is restricted to only include a one specific machine learning model. Implementing the result of the thesis in Spotify's CI environment is beyond the scope of the project.

1.5 Outline

This thesis is organized as follows:


Chapter 2 first gives a brief overview of the early history of test optimization, and then presents the recent research carried out within test prioritization in combination with machine learning and attempts to implement test prioritization in an industrial environment.

Chapter 3 is concerned with the methodology used for gathering data, implementation strategies and the experiments performed.

Chapter 4 presents the findings of the research, focusing on the commonly used metric Average Percentage of Fault Detected and two additional metrics regarding when the first failure is found.

Chapter 5 discusses the findings, the methods and aims to put the work in a wider context.

Chapter 6 concludes the findings by answering the research questions.



2 Theory

This chapter will lay out the theoretical dimensions of the thesis and look at the background information from related research.

2.1 Continuous Integration and Regression Testing

2.1.1 Continuous Integration

Continuous Integration (CI) is a software development practice with the explicit goal of being ready to deploy to costumers at all times. In practice, developers must therefore integrate their work frequently, at least daily, and keep release artifacts in a potentially releasable state. At a company or a project with multiple developers, this leads to multiple integrations per day, where each integration is verified by an automated build, including testing, to detect integration errors as quickly as possible [33].

Because of the high frequency of integrations, there is significantly less back-tracking to discover integration problems compared to older practices with longer time between integrations. This enables the company to spend more time building features rather than spending time on finding and fixing integration problems.

2.1.2 Regression Testing

The test activity performed in the CI process is called regression testing [14]. For every change in the existing software, the newly introduced changes are tested to provide confidence that the changes do not obstruct the behaviors of the existing, unchanged part of the software. There are several types of testing techniques that can be included in the term regression testing. The different types mentioned in this report are unit tests and integration tests. Unit testing aims to test the smallest units of code in an isolated manner and integration testing aims to test the interfaces between units. Both types of tests can be run pre- or post-merge. Pre-merge refers to testing before the integration of a code change, and post-merge refers to testing after the integration [40].

As software evolves, the test suites tends to grow. This growth introduces problems with respect to the time it takes to run the test suite and the resources needed for running it. As regression testing is done for every change, it may be prohibitively expensive to execute the entire test suite, [1, 36]. This limitation forces consideration of techniques that seek to reduce the effort required for regression testing in various ways.

2.1.3 Test Optimization

A number of different approaches have been studied to aid the regression testing process. A considerable amount of literature has been published on different methods concerning test optimization. In “*Regression testing minimization, selection and prioritization: a survey*” [38], S.Yoo and M.Harman survey the research undertaken on the subject up to 2012, and categorize the research into three major branches; test suite minimization, test suite reduction and test case prioritization.

Test suite minimization is a process that seeks to identify and then eliminate the obsolete or redundant test cases from the test suite. Test case selection deals with the problem of selecting a subset of test cases from the test suite, to only run test cases that are relevant for the code change that is tested. Finally, test case prioritization concerns the identification of the ideal ordering of the test cases that maximize desirable properties, such as early fault detection.

Several recent attempts within the field of test optimization have been made to test programs on large farms of test servers or in the cloud, e.g. [3, 17, 35]. However, this work does not specifically consider CI processes or regression testing. Furthermore, even when executing test suites on larger server farms or in the cloud, the rapid pace at which code is changed and submitted for testing in an CI environment, can lead to bottlenecks in either testing phase. The related work that have addressed the need of adaptation to CI environments is presented in Section 2.8.

2.2 Definition of Test Prioritization

Research into test prioritization has a long history. Not all previous research can be covered in this report, but research related to the aim of the project will be described. The area of research started in 1997 when Wong et al. first introduced the test case prioritization problem [37]. The technique presented in that paper first selects test cases based on modified code coverage, and then prioritizes them. Other methods have been investigated, including modification-based, fault-based, requirement-based, generic-based and history-based [38].

Regardless of which method, or combination of methods being used, a general and formal definition of test prioritization can be formulated as follows:

Given: A test suite, T , the set of permutations of T , PT , and a function f from PT to real numbers 2.1.

Problem: For all permutations of T , find the permutation that yields the highest score by function 2.1. In other words, find 2.2 such that 2.3 holds.

$$f : PT \rightarrow \mathbb{R} \quad (2.1)$$

$$T' \in PT \quad (2.2)$$

$$(\forall T'')(T'' \neq T')[f(T') \geq f(T'')] \quad (2.3)$$

A common evaluation function, 2.1, to use is Average Percentage of Fault Detected (APFD) described in Section 2.9 about metrics.

2.3 Baseline approaches

Research in the domain commonly uses simple prioritization techniques with which to compare novel approaches. Random prioritization, original order and reverse prioritization, are three common techniques, used in a vast set of studies [10, 9, 4, 32, 30], [12, 32, 30] and [9, 10, 30] respectively. If the test results are known, one can use the optimal prioritization. One downside is that there are multiple optimal solutions if the faults are not ranked based on severity.

2.4 Coverage-based approaches

As mentioned earlier, a coverage-based approach was the first attempt to prioritize tests. Code coverage is a widely used quality metric that measures how much of the code (e.g., number of lines, blocks, conditions etc.) from the program that is exercised during the tests execution. The intuition behind coverage-based RTP methods is that early maximization of structural coverage will also increase the probability of early maximization of fault detection. As faulty code needs to be executed to reveal its fault, covering more code increases the probability of covering the faulty code. Nevertheless, covering the faulty code may not always result in detecting its faults [40]. Faults are only revealed when the faulty code is executed with input values which causes the tests to fail.

However, several papers conclude that the hypothesis about a correlation between coverage and fault detection is proven to be correct [30, 41, 11]. A common limitation of coverage-based techniques is that they require coverage information for the old version of the system, which may not be available and can be costly to collect [32]. It is not even feasible in some situations [8]. Furthermore, the use of non-code artifacts, like configurations, is typically not accounted for by coverage based techniques [22].

2.5 History-based approaches

The rationale behind history-based RTP approaches is that if a test detects a fault in the past, it is likely exercising a part of the code that used to be faulty. Defect prediction studies have shown that if a part of the code used to be faulty, it is highly likely to be faulty again, specially if it is being changed [42, 24].

The typical history-based approach to RTP goes through the history of the software and identifies test cases that used to fail in previous test runs. Those previously failed test cases will be ranked higher in the prioritized list of test cases.

Kim and Porter [16] proposed the first history-based technique in 2002, based on this idea. They compute scores for prioritization using a smoothed, weighted moving average of past failures. This value is calculated by Equation 2.4 and Equation 2.5. The status of a test execution is represented with h , it takes on the value 0 if the test case passed and 1 if the test case failed. The smoothing constant, α , determines how much importance the history of a test has, and following how much it will impact the prioritization. A high value of the smoothing constant will assign a high value to test cases that revealed faults recently. In Equation 2.4 there is no execution history of the specific test case.

$$P_0 = h_1 \quad (2.4)$$

$$P_k = \alpha + (1 - \alpha)P_{k-1}, 0 \leq \alpha \leq 1, k \geq 1 \quad (2.5)$$

Newer research has improved this idea in various ways, Elbaum et al. [12] applied a history-based approach tailored to CI environments. Test suites are prioritized by assignment of two priority levels. The high priority includes test suites that recently revealed a failure, test suites that have not been executed in a long time or are completely new. One improvement is the boost of newly added test suites. New test suites do not have any failure history and would be ranked low in approaches that only consider the failure history. On the other hand, new test suites are likely to test new or untested functionality and are therefore likely to reveal faults.

An additional problem with only considering the history of tests is that there are situations, such as when an old test is modified, the test that will detect a fault is not exactly the same as any of the previously failing tests. However, it can be similar to old failing test cases, in terms of the sequence of methods being called. Noor et al. aim to aid this flaw by factor the similarity between tests, in terms of coverage, into the historical weighting [23].

2.6 Modification-based approaches

Several attempts have been made to investigate how code change can be used to perform RTP [34, 30, 37, 10]. Different tools have been investigated to gather data about a code change. Source code differencing data can be gathered from commonly available tools like Unix diff [10]. Another approach is to use data and control flow analysis, to investigate which parts of the system under test might be affected by the change. Lastly the change can also be identified on a binary level [34].

The most recent research within modification-based RTP is utilizing the field of Information Retrieval (IR) to compare changes in source code with the source code of the test cases. The following subsection will lay out the background information in the field to describe how it is utilized to perform RTP.

2.6.1 Information Retrieval

Traditional IR techniques focus on the analysis of natural language in an effort to find the most relevant *documents* in a collection based on a given *query* [19]. An example is retrieving relevant web pages containing text based on a keyword search.

Generally there are three steps performed in IR; pre-processing, indexing and retrieval. Pre-processing usually involves text normalization, removal of stopwords and stemming. A text normalizer is removing punctuation, performing case-folding, tokenizing terms etc. Removal of stopwords refers to removal of frequently used words that do not provide any information, in natural language that is for example prepositions and articles. Finally, stemming conflates variants of the same term, e.g. stemming the words *eat*, *eating*, *eats* all become *eat* to improve the term matching between the query and document. After pre-processing indexing of the documents is done for faster retrieval. Finally, queries are submitted to the search engine, which returns a ranked-list of documents in response. This ranked list can then be evaluated by measuring the quality of ordering with respect to the query.

2.6.2 Test Prioritization using Information Retrieval

In recent years, there has been an increasing interest in not only extracting information from natural language, but using IR techniques for extracting useful information from source code and software artifacts [29, 28, 31]. The effectiveness of these solutions relies on the use of meaningful terms, such as identifier names and comments in source code. Saha et al. used this key insight from previous research that developers that use descriptive identifier names and comments also tend to name their tests with similar terms. This hypothesis of a textual relationship was used in Saha et al.'s research [32], for the first attempt to reduce the RTP problem to a standard IR problem. Basically, it means seeing tests as documents and changes as queries, and given a change, rank the tests in order of relevance. The problem of RTP is formally transliterated as follows:

Given two sets, 2.6 and 2.7, where Δ represents the set of changes made to a system and τ represents the set of tests testing the system. In IR, these sets correspond to queries and documents, respectively. Given a change in the change set, 2.8, and a subset of tests in the test set, 2.9, the task is to rank elements of T using information about δ and T . Ranking is conducted by sorting T according to scores obtained from a scoring function, 2.10.

$$\Delta = \{\delta_1, \delta_2, \delta_3, \dots, \delta_M\} \quad (2.6)$$

$$\tau = \{t_1, t_2, t_3, \dots, t_N\} \quad (2.7)$$

$$\delta \in \Delta \quad (2.8)$$

$$T \subseteq \tau \quad (2.9)$$

$$f(\delta, t) : \Delta \times \tau \rightarrow \mathbb{R} \quad (2.10)$$

There are a wide range of scoring functions to use. Gomaa et al. discuss the existing works through partitioning test similarity approaches into three branches; string-based, corpus-based and knowledge-based [13]. The rationale behind this partition is that words can either be similar lexically or semantically. Words are similar lexically if they have a similar character sequence and words are semantically similar if they have the same meaning, are opposite of each other, used in the same way, used in the same context or a word is a type of another word. Lexical similarity corresponds to the string-based approaches, and semantic similarity is introduced by corpus and knowledge-based approaches. Corpus-based approaches measure the semantic similarity between words based on information gained from a large corpus. Knowledge-based is measuring the semantic similarity based on information gained from semantic networks. The scoring function used by Saha et al. [32] is Okapi BM25, where BM stands for Best Matching. It is a corpus-based ranking function, that uses term frequency, which is counting how many times terms appear in each document, regardless of the inter-relationship between the terms within a document.

2.7 Machine Learning

Machine learning has long been a topic of great interest in a wide range of fields as diverse as business, medicine, astrophysics, and public policy [15]. Test prioritization is not an exception to the trend, a recent approach involving machine learning has been suggested for the problem [4]. This subsection contains background information about machine learning in general, to understand how the problem of RTP can be thought of as a machine learning task. The following section will describe how Busjaeger et al. utilize these techniques in *Learning for test prioritization: an industrial case study* [4].

Classification and Regression

Machine learning refers to a vast set of tools for understanding data. These tools can be divided into two groups; supervised and unsupervised [15]. Generally speaking, supervised learning refers to building a statistical model for predicting or estimating an output based on one or more inputs. With unsupervised statistical learning, there are inputs but no supervising output; nevertheless relationships and structures can be learned from such data.

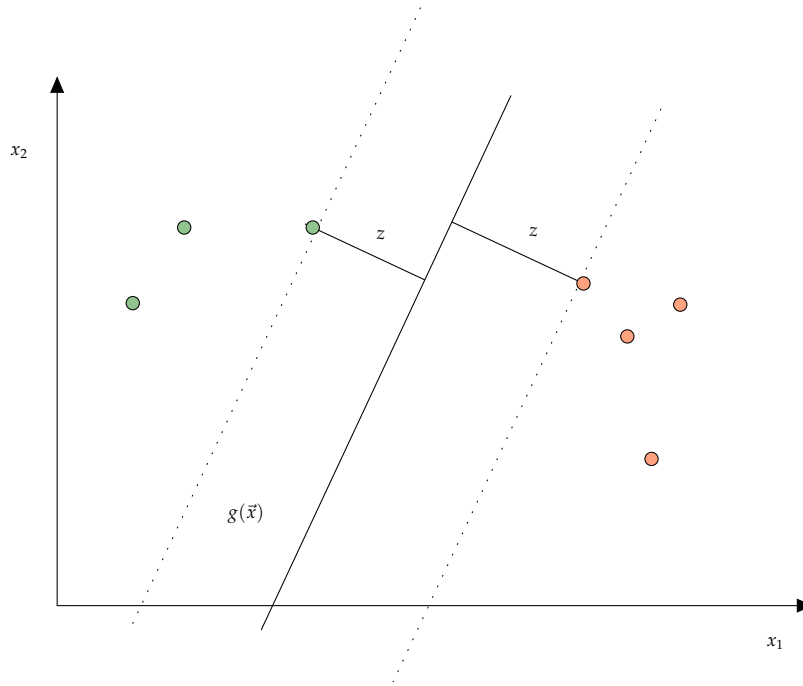
There are two types of variables; the ones that take on numerical values, called quantitative, and the ones that belongs to a category, called qualitative. An example of a quantitative variable is a test case's duration in milliseconds, and an example of a qualitative variable is the test case's status (success or failure). Problems with a quantitative output are often referred to as regression problems, while those involving a qualitative output are often referred to as classification problems. The distinctions between these problems are not always clear, some methods can be used to solve both types of problems. Logistic regression is an example of that [15]. It is often used as a classification method with a qualitative binary response, but it estimates class probabilities, and can therefore be thought of as a regression method as well. Logistic regression is just one type of machine learning model. The following subsections describe a selection of the vast variety of models that are available to be used for test prioritization.

2.7.1 Support Vector Machine

Support Vector Machine (SVM) is a supervised learning technique that can be used for classification and regression. It is an extension of the Support Vector Classifier (SVC) that results from enlarging the feature space in a specific way, using kernels [15]. SVC is in turn an extension of a maximal margin classifier [15].

The main idea of all three approaches is to produce a hyperplane which separates the samples in a data set, according to how the samples are delimited by the hyperplane. The hyperplane constructed in a maximal margin classifier will maximize the margin between the hyperplane and the closest samples.

Figure 2.1: An example of a Maximal Margin Classifier in two dimensions



The closest samples therefore impact the position of the hyperplane and will act as support vectors for the hyperplane. Figure 2.1 shows a two-dimensional feature space, x_1 and x_2 . If one of the closest samples had taken on another value the line would have taken on another value, it is in that way sensitive. SVC is called the soft margin classifier since the margin from the hyperplane allows violation of some training samples, violations in the sense that it can be on the wrong side of the hyperplane or just violating the margin. This property increases the robustness of the classifier and makes it more general since data rarely is optimal for finding a linear hyperplane.

The distance z is calculated by Equation 2.11, where weight \vec{w} is the support vectors that span up the hyperplane, $g(\vec{w}x)$. Samples with values larger than one are classified as *class_green* and samples with values less than negative one are classified as *class_red*.

$$z = \frac{|g(\vec{x})|}{\|\vec{w}\|} = \frac{1}{\|\vec{w}\|}, g(\vec{x}) \geq 1 \forall \vec{x} \in \text{class_green}, g(\vec{x}) \leq -1 \forall \vec{x} \in \text{class_red} \quad (2.11)$$

In some data sets a linear classifier such as SVC is not sufficient. For those situations there are different functions for creating a hyperplane which are called kernel functions that produce hyperplanes of different shapes. The creation of kernel functions is a research area in itself, but some well-known kernel functions are: linear, polynomial, radial basis function and the sigmoid function that will create hyperplanes of different shapes. This extended approach to use kernel functions for producing both linear and non-linear classifiers is called Support Vector Machine (SVM).

There are known advantages and disadvantages of SVMs. They are effective in high dimensional feature spaces, even in the cases where number of dimensions is greater than the number of samples. However, a too significant difference would likely give poor results. It is versatile in the sense that different kernel functions can be specified that enables the method to suit a lot of different data sets. SVMs do not directly provide probability estimates, these are calculated using an expensive k -fold cross-validation. Cross-validation is a way to utilize the data, without the problem of over-fitting. Over-fitting means that the model gets too adopted to the training data, making it perform worse on unseen samples compared with more generalized models. In k -fold cross-validation, the training set is split into k smaller sets. Then the following procedure is followed for each of the k folds; A model is

trained using $k - 1$ of the folds as training data, the resulting model is validated on the remaining part of the data.

2.7.2 Naive Bayes Classifier

Bayesian classifiers are calculating probabilities of belonging to a class using Bayes' rule [2]. Given the class variable y and a dependent feature vector x_1, \dots, x_n Bayes' theorem states the relationship as shown in Equation 2.12. It is called *naive* because it assumes that all attributes is conditionally independent of each other, given the class. Given that assumption, Equation 2.13 is derived.

$$P(y|x_1, \dots, x_n) = \frac{P(y)P(x_1, \dots, x_n|y)}{P(x_1, \dots, x_n)} \quad (2.12)$$

$$P(x_i|y, x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n) = P(x_i|y) \quad (2.13)$$

For all i this relationship is simplifying Equation 2.12 to Equation 2.14.

$$P(y|x_1, \dots, x_n) = \frac{P(y) \prod_{i=1}^n P(x_i|y)}{P(x_1, \dots, x_n)} \quad (2.14)$$

Once the model is trained, it can be used to classify new samples for which the class variable y is unobserved. With observed values on the attributes x_1, \dots, x_n the probability of each class is given by Equation 2.15.

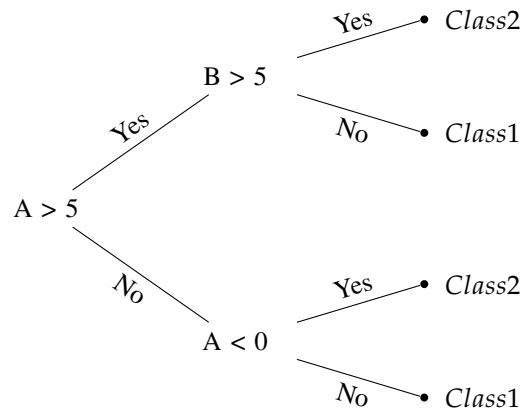
$$P(y|x_1, \dots, x_n) \propto P(y) \prod_{i=1}^n P(x_i|y) \quad (2.15)$$

Despite the over-simplified assumptions, naive Bayes classifiers have worked well in many real-world situations. It is famous for the usage in document classification and spam filtering [21]. One advantage of the naive Bayes classifiers is that it only requires a small amount of training data to estimate the necessary parameters. Another advantage is that they can be extremely fast compared to more sophisticated methods. The decoupling of the class conditional feature distributions means that each distribution can be independently estimated as a one dimensional distribution. This in turn helps to alleviate problems stemming from the curse of dimensionality. On the other hand, although naive Bayes is proven to be good classifier, it is known to be a bad estimator, therefore it is not well suited for regression tasks.

2.7.3 Decision tree

Decision trees can be applied to both regression and classification problems [2]. To build the tree structure, training data is used to recursively split up the data into branches. To perform splits, thresholds are applied to the tree at so called nodes. It can be thought of as conditions on attributes in the training data. Figure 2.2 exemplifies a binary classification decision tree, starting from left to right. Assuming that the data have two numerical attributes A and B . Different algorithms can be used to decide which conditions each node should have, cross entropy is an example. The tree is recursively constructed until the stopping criteria is fulfilled. The class that is assigned to each leaf is decided by the majority of observations from the training data set that ended up in that leaf. When the tree is created it can be used for predicting data by letting new data samples traverse through the tree to get assigned to a class. In case of a regression problem, the leaves correspond to real values. Decision trees as an algorithm in itself often produces bad results with models that over-fits the data, but in other approaches, such as Random Forest which is an improved version of decision trees the resulting model gives much better results [15].

Figure 2.2: Decision tree example



2.7.4 Test Prioritization by Learning to Rank

Learning to rank is a subfield of machine learning concerned with the construction of ranking models of information retrieval. Training data consists of item lists with associated judgment, usually binary or ordinal, which induce an order or partial order of the items. The objective is to train a model to rank unseen item lists to maximize a desired ranking metric. Learning algorithms are categorized into point-wise, pair-wise and list-wise algorithms. Point-wise algorithms reduce ranking to standard classification techniques. Pair-wise algorithms classify pairs of points with the objective of maximizing inversions, and list-wise algorithms consider complete item lists using direct continuous approximations for discrete ranking models. Pair-wise and list-wise algorithms are more natural fits for learning to rank as they optimize an easier problem of predicting relative as opposed to absolute scores. Empirically, they outperform point-wise algorithms [18].

Busjaeger and Xie were the first to leverage the field of machine learning to train a ranking model [4]. Implementing test prioritization by learning to rank uses the problem definition formulated in terms of IR, described in Section 2.6.1 about IR. To cope with heterogeneity, they combine multiple heuristic techniques shown by existing research to perform well. These techniques include test coverage, text similarity, recent test-failure or fault history and test age.

The text similarity score used resembles the method by Saha et al. [32] but another similarity score was used, called cosine similarity. The fault history feature is calculated as presented by Kim and Porter [16], described in Section 2.5. The coverage metric is inspired by the research by Rothmell et al. [30], and lastly test age is based on the research by Elbaum et al. [12].

The rationale behind choosing these features is that all features help to reveal different types of faults. Code coverage can identify interdependencies between seemingly unrelated parts of the system under test, such as impacts on low-level persistence-logic changes on user-interface tests. Text similarity performs well for non-code changes, such as a change in a configuration file, since it is likely that the configuration file contains similar terms as the test, and this is not captured by coverage metrics. Fault history accounts for temporal relationships, which help identify tests impacted during churning or non-deterministic code. Finally, boosting new tests alleviates the cold-start problem of not having any prior coverage, text or faults data.

The text similarity based on paths is their best performing feature. The previous coverage, history and age features yield similar results, and combining all features is outperforming all other individual approaches indicating that the combined multiple heuristics is correct [4].

2.8 Test Prioritization in an Industrial Environment

Recent developments in test prioritization have highlighted the need for integrating the practice in CI environments. This section presents the difficulties of integrating the practice in an industrial environment.

There is a significant difference between the research carried out in academic contexts compared to research that is based on industrial data or implementations of RTP in an industrial environment. Transferring academic research on test prioritization into an industrial setting requires significant adaptation to account for practical realities of heterogeneity, scale and cost. There are a few examples of research addressing the problem of integrating RTP in an industrial environment; Microsoft applied test prioritization for testing Windows [34, 6] and Dynamics Ax [5]. Google evaluated prioritization to optimize pre- and post-submit testing for a large, frequently changing code base [12, 39]. Cisco investigated prioritization for reducing long-running video conferencing tests [20]. Finally, Busjaeger and Xie, presented an approach with data from salesforce.com [4]. Their approach is described in Section 2.7.4.

There is no uniform way of solving the problems with integrating RTP in practice. The implementations presented in the mentioned papers are therefore diverse. On the other hand, the papers [12, 5, 20] have some common aspects, that is the use of historical data. The algorithm presented by Elbaum et al. [12], calculates the time since the test case last revealed a failure, combined with the time since the last execution. This combined value is then used for prioritization. Carlson et. al present a clustering approach [5]. They cluster the test cases depending on history data of faults, as well as code coverage and code complexity. Marijan et al. present an approach where the test cases are ordered based on historical failure data, test execution time and domain-specific heuristics [20]. It uses a weighted function to compute test priority. The weights are higher if tests uncover regression faults in recent iterations of software testing and reduce time to detection of faults.

2.9 Metrics

The success rate of the ordering of the tests can be measured in different ways. The metric depends on the objective of the prioritization. A way to measure the most common objective, that is fast failure, was proposed in 1999 by Rothermel et al. [30] called Average Percentage of Fault Detected (APFD). Since then have multiple refinements of APFD been proposed and new metrics have been developed as the RTP field has evolved. The following subsections will describe the metrics used in the related research that has been the most influential on this thesis.

2.9.1 Average Percentage of Fault Detected

APFD is a metric that measures the effectiveness of prioritization in terms of rate of fault detection, where a higher APFD value denotes a faster fault detection rate. More formally, let the test suite T contain n test cases, and let F be the set of m faults revealed by T . For permutation T' of T , let TF_i be the order of the first test case that reveals the i th fault. The APFD value calculated for T' is calculated by Equation 2.16.

$$APFD(T') = 1 - \frac{\sum_{i=1}^m TF_i}{nm} + \frac{1}{2n} \quad (2.16)$$

The aim is not to maximize APFD but to evaluate how well a prioritization technique is performing. Maximization of APFD is possible only when every fault that can be detected by the given test suite is already known. This would imply that all test cases already have been executed, which would annul the need to prioritize.

To explain the APFD formula with an example, Table 2.1 is used to visualize a simplified test run. The table shows all test cases in the order they were executed and the faults in the order they were found. The first fault was revealed by the second test case, and so on. The total number of faults

found, m , is equal to 3, and total number of tests executed, n , is equal to 5. Equation 2.17 is a concrete example how the value is calculated using the example Table 2.1.

Table 2.1: Visualization of test execution and corresponding faults

	Test1	Test2	Test3	Test4	Test5
Fault1		*			
Fault2			*		
Fault3				*	

$$APFD = 1 - \frac{2 + 3 + 4}{5 \cdot 3} + \frac{1}{2 \cdot 5} = 0.5 \quad (2.17)$$

Even though this metric was first presented in 1999, it is still used in recent research to evaluate the new approaches, [4, 32]. However, R. Pradeepa and K. VimalaDevi [26] evaluate the metric in depth and discuss its limitations. It is concluded that the limitations rely on two assumptions, (1) all faults are of equal severity and (2) all test cases have equal cost. It reveals that under conditions where test cases differ in cost and faults differ in severity, the metric can provide unsatisfactory results. Various studies have addressed the same problems and suggested improved metrics. Elbaum et al. proposed a new metric to measure the effectiveness of test case prioritization considering their influence [11] and Qu et al. presented a normalized APFD to approach the same problems [27].

2.9.2 Precision and Recall

In the machine learning approach to RTP by Busjaeger et al. [4] presented in the Section 2.7.4, the classical way of measuring test case prioritization with APFD is combined with techniques that are common for machine learning applications, namely precision and recall. However, other common machine learning measures including accuracy and F_1 -score were not mentioned in that paper.

In a classification problem, precision is the fraction of classified samples that are classified correctly, while recall is the fraction of relevant samples that are retrieved. Precision and recall are calculated for each class by Equation 2.18 and 2.19, respectively, where Table 2.2 explains the abbreviations used in the equations. The table is an example of a binary classification of the classes failure and success, and the measure in this case is focusing on the failure class.

$$Precision = \frac{tp}{tp + fp} \quad (2.18)$$

$$Recall = \frac{tp}{tp + fn} \quad (2.19)$$

Table 2.2: Classification outcomes

	Failure	Success
Classified correct	True Positive: tp	True Negative: tn
Classified wrong	False Positive: fp	False Negative: fn

Average precision (AP) is a commonly used metric for binary judgments evaluating ranking tasks, it was therefore used in Busjaeger et al.'s research [4]. It averages precision across all recall points. This metric is different from APFD because it does not take the total number of items into account, but provides insight on how many non-relevant items are ranked in front of relevant ones on average. The following equation 2.20 shows how this metric is calculated, where M represents the ranks of relevant items and R_k represents the ranking up to position k .

$$AP(Q) = \frac{1}{M} \sum_{k \subseteq M} Precision(R_k) \quad (2.20)$$

2.9.3 Gained hours

In situations where all tests results are not recorded, for example if the prioritization is integrated in the CI system and it is evaluated in runtime or the execution is aborted when the first failure is found, it is impossible to calculate the APFD value. Therefore, it is more suitable to use metrics that measure time and cost. Elbaum et al. used gained hours over no prioritization as a metric [12].

In Elbaum et al.'s study the prioritization technique is evaluated by measuring the time it takes for the test suites to exhibit a failure. The reason explained for not using APFD is that when integrating the prioritization algorithm in a CI environment, the focus is on obtaining feedback on individual test suites rather than calculating a cumulative value of fault detection over time. It is stated that it is more suitable in situations where batches of test cases are used.



3 Method

This chapter is concerned with the methodology used for gathering data, implementation strategies and the experiments performed. To investigate the research questions a system to compare RTP approaches was developed. The approaches were developed in an agile fashion by starting with simple baseline approaches and then iteratively extending and adding more advanced approaches using history-based, modification-based and machine learning approaches.

3.1 Data at Spotify

Spotify store all data produced by all builds in Google Cloud Storage, that is Google's web based RESTful file storage for storing and accessing data ¹. The data is stored in raw format and it is therefore not easy to query and use. For that reason, some of the data is also stored in indexes in Elasticsearch. Elasticsearch is a search engine with an HTTP web interface and schema-free JSON documents ². There are two indexes at Spotify that are useful for this project, one containing metadata of all builds, and one containing metadata of all test runs. Elasticsearch is easier to query and it also comes with a visualization tool, Kibana, that is useful to visualize and understand the data set.

Examples of metadata stored in the index that contains test result are: who owns the test, who runs the test, what is the duration of the test, what is the name of the test and on what version of the code was the test executed on.

Spotify uses git and Github Enterprise (GHE) ³ as version control tools. GHE stores data about all versions of the code, that is different from the data that is stored in the Elasticsearch index about the builds of the versions. An example of data that can be retrieved from GHE is the diff file between two versions, that summarizes the changes between the versions.

3.2 Data set

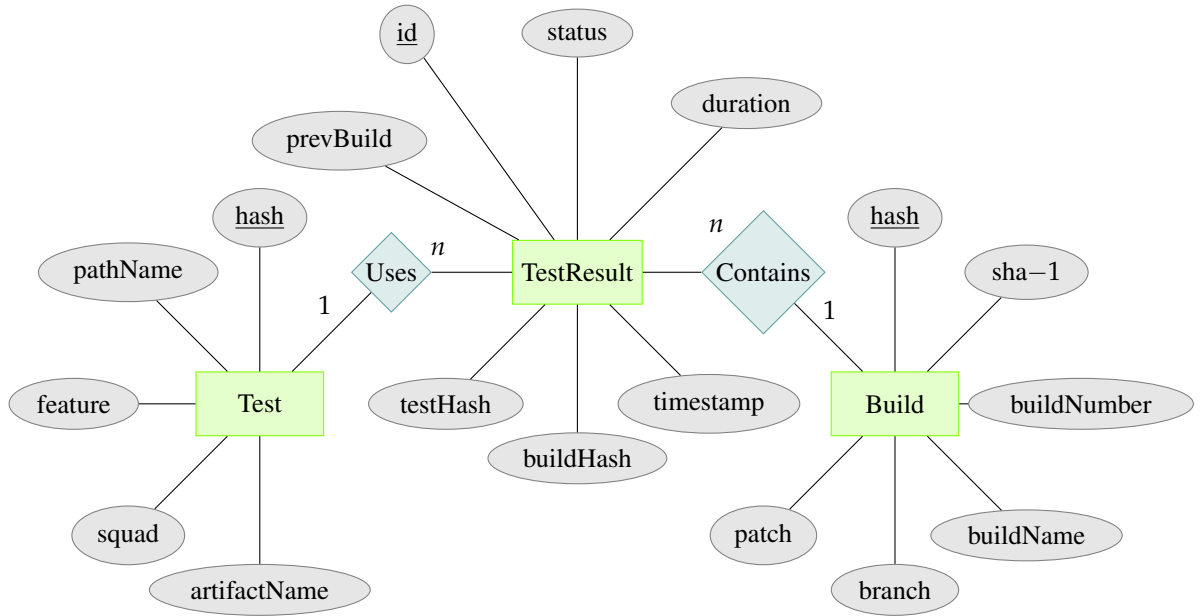
The data set used for evaluating the RTP approaches consists of data from one of Spotify's projects and one specific type of test. The chosen project is the iOS client of the Spotify application, and the tests that are investigated are post-merge integration tests. Post-merge tests are most suitable to be

¹<https://cloud.google.com/storage/>

²<https://www.elastic.co/products/elasticsearch>

³<https://enterprise.github.com/home>

Figure 3.1: ER-Diagram of SQLite database



prioritized because they are most vulnerable for integration. There are two types of tests that run post-merge, integration tests and unit tests, and since the unit tests are tested pre-merge, that dataset contains few failures and is therefore not an interesting data set to evaluate the approaches with.

The data set is composed of data from two different sources. An Elasticsearch index, containing data regarding test runs and GitHub Enterprise (GHE), containing data about the code change. The retrieved data from Elasticsearch is separated with respect to test case, test result and build information. Regardless of the fact that it is a subset of the data that is stored on these resources that is investigated, the whole data set could potentially be requested and would fit the model that is designed.

For every build in the data set, code change data from GHE is fetched. Working with a considerably large data set makes these requests take a significant amount of time. For that reason, an SQLite database was implemented, so that the data set could be saved locally and retrieved fast for testing purposes. The entities, their attributes and relationships in the database that will be referred to throughout the report is visualized in Figure 3.1. The green rectangles represent the entities, the gray ovals represent the entities' attributes and the blue diamonds represent the relationships between the entities.

Figure 3.2: Version history in practice

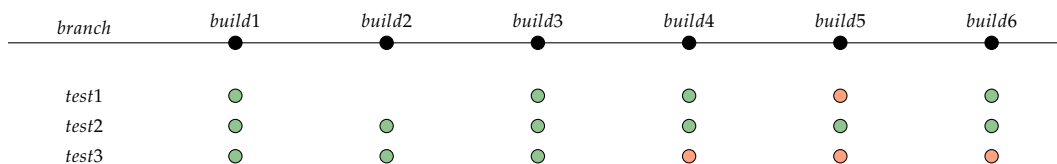


Figure 3.2 illustrates the version history of a branch, with corresponding tests and test results. It can be thought of as the relationship between the entities in the database. A build contains metadata regarding the build environment, such as the branch name, the build id and so on. For every build, a certain number of tests run, and they either fail or pass. The test result is therefore represented with a green or a red circle. A green circle represents a successful test result, and a red circle represents a test result failure. The first test, *test1* does not run for *build2*. That example illustrates that there are two

code changes that are of interest; the code change between builds and the code change between when a specific test is executed.

There is no unique attribute in the data that is referring to a specific build, since the same build might run multiple times with different settings. Therefore the private key of the Build entity is a hash of a combination of attributes that is ensuring that the key is unique. The same approach was used to create the private key for the Test entity.

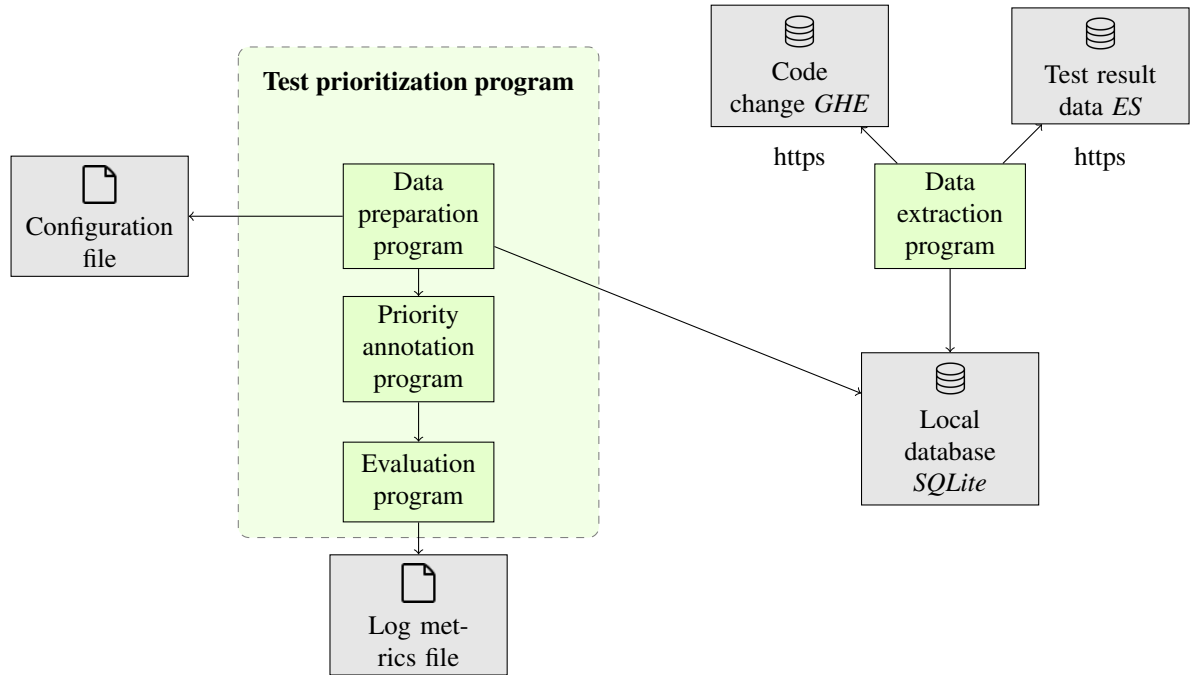
The data set consists of approximately 470 000 test results, 4 200 builds and 350 tests. It was 3 months of data that could be extracted, before that date the data was malformed and unusable. To decide the amount of data to extract to ensure that the result is reliable, the data set size is compared to related research. Busjaeger et al. also extracted 3 months of testing results, consisting of 45 000 tests results [4]. The total number of data points that correspond to a list of tests where, 2 000 builds. 711 data points of those 2 000 were associated with at least one failure and could therefore be used for evaluation. Elbaum et al. gathered 30 days of test data with 3.5 million test results [12].

Since the machine learning approach needs data to train the model on, 80% of the data set is dedicated for training. Before the division, the builds are shuffled with a seed input so that the shuffle is deterministic for every evaluation run. Although the builds are shuffled, the data structure used preserves the internal order of each test cases history. The evaluation of the approaches is performed on the remaining 20% of the data set, it is called the test set. All evaluation on all RTP approaches are performed on that test set, to enable comparison between the approaches. The resulting test set is therefore 20% of 4 200 builds, that is 840 builds. In addition, only builds that had at least one failing test are used in the evaluation, that leaves the remaining 450 builds that were used for calculating the evaluation metrics. This filtering is done when the metrics are calculated. The resulting size of the test set is less than some of the data sets used in related research described earlier, however it is assessed to be large enough to give a good indication of the result. A concrete comparison that indicates this is done between the data set size used by Busjaeger et al. [4]. They had 711 data points in their data set, where 440 data points were used for training and 271 data points were used for evaluation. In that specific case, the test set in this study containing 450 builds is larger. One difference between the data sets is that Busjaeger et al. only considered test sets related to at least one failure in the training set compared to this study where the filtering was performed in a later phase, when the metrics are calculated.

3.3 Framework for Evaluating Prioritization Approaches

This section describes the main structure of the system for evaluating test case prioritization techniques and its high-level architecture. It will also provide a compilation of the different tools, libraries and frameworks used in the development process of the application. The high-level idea of the system is to evaluate RTP approaches by calculating all metrics on the test set.

Figure 3.3: High level overview of the evaluation system



3.3.1 System architecture

A high-level overview of the implementation is provided in Figure 3.3. It describes the overview of the evaluation system’s components. The gray boxes in the diagram refer to different persistent data storages, e.g. log files, databases or servers, and the green boxes are system components.

A prerequisite to run the evaluation system, named *Test prioritization program* in Figure 3.3, is to run a script that downloads the test data set, from Elasticsearch and GHE, pre-processes the data and populates the database. This program is separated from the rest of the evaluation system, and it can be scheduled to run as often as one would like to update the database. In Figure 3.3 it is called *Data extraction program*. This script ran once for the experiments performed in this thesis to test all approaches on the same data set.

The first step in the evaluation system is to query the local database for the test data set. It is using a model that is retrieving the test data from the local database and transferring it into an in-memory object. A test set is created from this object and it is passed to any concrete algorithm classes. The first step is in Figure 3.3 called *Data preparation program*.

The algorithm under investigation and what database file to retrieve data from is stated in what is referred to as configuration file in Figure 3.3. The part of the system that is executing the algorithm under investigation is the *Priority annotation program*. It takes the test set and applies the algorithm on each list of test cases in the set. The last part of the program, the *Evaluation program*, executes the actual evaluation and outputs a log file with all metrics. The metrics used are described in Section 3.3.3.

3.3.2 Tools and Frameworks

This section includes information about which tools and frameworks that are utilized. The tools and frameworks that have been used for the evaluation system have been selected according to the following criteria; will it ease development, prevent reinventing the wheel, and does it have a good community reputation from other users. All code in the evaluation system is written in Python, mainly for the vast set of modules available that fulfills these criteria. The most important modules utilized were;

`Scikit-learn`, which includes a wide range of state-of-the-art machine learning algorithms. More information about the module and how it is used is described in Section 3.7.

`DiffLib`, provides classes and functions for comparing sequences. It was used to compute text-similarity scores for all modification-based approaches. More information is found in Section 3.6.

`Unidiff`, a library with functionality to parse and interact with unified diff data. It was used in the modification-based approaches to easily extract information from the diff files. More information found in Section 3.6.

3.3.3 Metrics

The metrics used for evaluating the prioritization approaches were chosen to be able to compare the approaches to related research.

The evaluation can only be performed on builds that include at least one failure, since all permutations of a test set that only contain successful tests are optimal. For that reason only builds that correspond to test sets that include at least one failure are considered when the metrics are calculated.

The APFD metric described in Section 2.9 is calculated. It was mentioned that there are improved versions of the APFD metric, however, these were not considered due to lack of data regarding severity of faults. Another motivation for using the original APFD metric is to be able to compare this investigation with related research, the first machine learning-approach to RTP by Busjaeger et al. [4] in particular.

Two additional metrics were introduced to be able to reason about how the approaches would perform if they were to be integrated in the actual CI system at the company. In Section 2.9 one metric was presented that aimed to evaluate approaches that were integrated in CI environments, gained hours over no-prioritization [12]. This metric was considered but due to time limitations two simpler metrics based on average time and rank were implemented instead. The two metrics are *average percentage of time to first failure* and *average rank of first failure*. The first metric is concerned with; how big part of the aggregated execution time elapsed before the first failure on average? And the second metric is concerned with; at what rank is the first failure on average? The metric based on time sums all execution times. This is not an accurate time measurement since when running tests additional tasks take time, such as setting up and closing down the test environment. That is the reason for not including a commonly used time unit such as seconds or minutes. Spotify has an internal measurement that is called *agent hours*, which is a measure of how much time in total one build agent would spend if the tests were not parallelized on different build agents. This metric would resemble the gained hours in a more accurate way, however, that information was not extracted to the data set. The implemented metrics, average percentage of time to first failure and average rank of first failure, are new metrics presented for this type of problem. It is therefore not comparable to related research but is an indication of how the approaches perform and are more graspable than the APFD value.

Precision and recall are two common machine learning metrics described in Section 2.9. These metrics were collected during development, but the ranking was not measured with these metrics. When formulating the task as a classification task instead of a ranking task the metric becomes less interesting because of the granularity. Using normal machine learning metrics would be skewed, because the data set is skewed and it is desirable that the predictions do not match that. If all test were classified as successful it would probably yield good metrics, but useless prioritization. The aim is to prioritize so that more samples are classified as failure to get an equilibrium in the prioritization. It is still desirable to prioritize even though the test suite will successfully pass all tests. Therefore, the machine learning metrics were not the focus of this report.

3.3.4 Algorithms

`Algorithm` is an abstract class that has one function, called `execute`. This function takes the test set as input, executes the RTP approach and returns all tests in the test set together with a value that corresponds to the rank of that specific test. The test set is then sorted based on the ranks. All approaches that are compared implement the abstract class `Algorithm`. This enables the framework to assume

that all algorithms can be treated the same way. All implemented approaches are based on theory presented in Chapter 2. The following subsections will describe how the implementations resemble the theory from related research. A list of all implemented approaches, Table 4.1, and their results are presented in Chapter 4.

3.4 Baseline approaches

The baseline approaches are implemented to measure and compare how well novel approaches perform. From the commonly used baseline approaches described in Section 2.3, two were implemented.

`NoPrioritization` is the first baseline approach that is implemented. It refers to the original order the tests ran in during the execution. To achieve this, the tests are sorted depending on timestamps of execution. `NoPrioritization` is used as a baseline to be able to reason about how the approaches would perform if they were integrated in the CI system, and to enable comparison of how the novel approaches would perform compared to the *retest-all* approach applied by the company today.

`RandomBaseline` is the second baseline approach that is implemented. It returns the tests with a random prioritization. It is a commonly used baseline and in some previous research it is shown that it is better than the original order [30].

A motivation for choosing these baselines is that the related research that is the most influential to this thesis are using these baselines [32] [12] [4].

3.5 History-based approaches

Since one of the aims of this project is to utilize Spotify’s build and test data, all methods that are dependent on existing data are of interest for this project. A suitable type of methods to investigate is therefore history-based approaches. In addition, multiple papers addressing the problems of integrating RTP in CI systems [12, 5, 20], all use a history-based approach.

3.5.1 Motivation

Inspired by the history-based RTP approaches implemented by Elbaum et al. [12] three approaches were implemented. The setting and aim of their research is different from this thesis, since their focus was to implement the approach in Google’s CI system, contrary to this thesis which focuses on investigation of prioritization approaches. A difference that follows from this is their approach to prioritize the tests in batches as they arrive in the build environment. They call this *test suite execution window*, where the window either is a time frame or a threshold based on an amount of test suites. As tests arrive in their CI process they are placed in a dispatch queue and when the window is full, prioritization of the tests within the window is executed. If the window is set to a test suite or a time frame that is less than the execution time for a test suite this approach will not do any prioritization. The prioritization is done by assigning two priority levels. The priority level of each test suite depends on if the time since the last failure is within a certain range or if the time since the last execution is within a certain range. In this investigation, there is no need to use a window, since all information exists. Another difference between the study conducted by Elbaum et al. and this study is that their prioritization is executed on test suite granularity, and this investigation is performed on test case granularity. This fact follows from the format of the data.

3.5.2 Implementation

The first history-based RTP approach that was implemented is called `LastExecutionAndLastFailure` and it captures the same idea as the approach by Elbaum et al. [12]. The pseudo-code of the approach is found in Algorithm 3.5.2. It assigns two priority levels to all test cases depending on when the test case was last executed, and when it last failed. To find relevant thresholds for the different priorities, two separate approaches were implemented; `LastBuildExecution` and `LastFailure`.

`LastBuildExecution` prioritizes the test cases depending on when the test case last run, and `LastFailure` prioritizes the test cases depending on how many times it failed during the history. `LastBuildExecution` is calculated by the time difference of the timestamps of when the test last run and the specific execution time. The larger the time difference is, the higher priority a test case gets. `LastFailure` calculates how many times the test case failed for the previous n test runs. The more failures recorded for the test over time, the higher priority it gets. An experiment was performed to decide how many previous test runs should be considered, hence the value of n , and the results are shown in Chapter 4. The constant n is in the Result chapter referred to as the history limit.

As observed and conjectured in previous work [7], new tests often fail since they exercise new and possibly churning code. However, history-based approaches often rank new tests lower because of the lack of data. This cold start problem can be accounted for by assigning high priority to unseen tests. In the pseudo-code of `LastExecutionAndLastFailure`, see Algorithm 3.5.2, this is implemented as the first check in the if-statement that is assigning the high priority. If no history data for that specific test exists, there is no use of calculating *HasRecentlyFailed* or *WasNotRecentlyExecuted*.

Algorithm 1 History-based algorithm: `LastExecutionAndLastFailure`

INPUT: A set of tests T

OUTPUT: A set of tuples R , with tests and corresponding priority (T_i, p)

```

 $R \leftarrow \{\}$ 
for all  $T_i$  in  $T$  do
   $p \leftarrow 0$ 
  if IsNewTest( $T_i$ ) or HasRecentlyFailed( $T_i$ ) or WasNotRecentlyExecuted( $T_i$ ) then
     $p \leftarrow 1$ 
  end if
   $R$  add ( $T_i, p$ )
end for
return  $R$ 

```

3.6 Modification-based approaches

The same motivation for investigating history-based approaches can be used to argue for investigating modification-based approaches, since one of the aims of this thesis is to use approaches that utilize Spotify's build and test data, all methods that include data is of interest for this project. There are mainly two papers that this section relies on, those are the modification-based features in *Learning for test prioritization: an industrial case study* [4] by Busjaeger et al. and *An information retrieval approach for regression test prioritization based on program changes* [32] by Saha et al.

3.6.1 Motivation

Two modification-based approaches were presented as features used in the machine learning approach by Busjaeger et al. described in Chapter 2, namely path similarity and content similarity. The result of the individual features show that content similarity performs worse than path similarity. For that reason only path-similarity is investigated. The same information, the path of the files that had changed and the path of the test, are extracted as Busjaeger et al. but a different similarity score measure is used. A character-based similarity score was calculated between the test case's path and the files that got changed path's and the files that got changed paths. This similarity function was chosen because of the convenience of using an existing Python library, `difflib.SequenceMatcher`, described more closely in the next subsection.

The modification-based approach that inspired Busjaeger et al. to use text similarity score as a feature to their model was Saha et al.'s research [32]. Their research considers text similarity on different granularities of source code level, as an example JUnit has two types of test cases; test-classes

and test-methods. They collect information to the documents on each granularity they investigate. The queries are constructed from the code changes, this was also performed on several granularity levels. As an example, low level corresponds to information in the diff file and high level corresponds to information collected from a change impact analysis tool called FaultTracer.

In Saha et al.'s research it is described that tokenization of source code differs from written English, for instance, names are generally a concatenation of words based on a camel case heuristic. As an example `TestFileName` becomes `[test, file, name]` after tokenizing with this heuristic. This was applied to the test file name in all implemented modification-based approaches.

3.6.2 Implementation

`PathSimilarity` is an approach that is calculating the aggregated result of the similarity between the test case's path name and the paths of the files that got changed. The path is split up into tokens by separating the path, as an example `/directory/another_directory/filename.java` becomes `[directory, another_directory, filename.java]`.

`difflib.PatchSet` is a library that provides an easy way of parsing the information from diff files. The paths can easily be extracted of all changed files, and the filename can easily be extracted from the path, since it is the last token in the path sequence. In the pseudo-code describing the `NameSimilarity` approach, Algorithm 3.6.2, the function *GetChangedFileName* is referring to the functionality in the `unidiff.PatchSet` library.

`NameSimilarity` is an approach that is calculating the aggregated result of the similarity between the test case's name and the files that got changed names. This approach was constructed based on the hypothesis that the filename is the most informative part of the file path. This hypothesis was founded on intuition by looking at the data. An additional assumption was made regarding that the file ending does not provide any additional value to the similarity score, it was therefore thought of as a stop word and was removed. As an example `filename.java` becomes `filename` after preprocessing. In the pseudo code, Algorithm 3.6.2, this is referred to as the preprocessing step, the *PreProcessing* function.

The Python library `difflib.SequenceMatcher` was used to calculate the text similarity in both `PathSimilarity` and `NameSimilarity`. It is a general pattern matching algorithm, that for textual input corresponds to a character-based approach to measure text similarity. The reason for not using a Corpus-based or Knowledge-based approach, described in Section 2.6.1, is based on the naming conventions hypothesis of developers naming tests similarly to the functionality it aims to test and the ease of using Python's standard library. In the pseudo code, Algorithm 3.6.2, this is referred to as the *GetTextSimilarity* function.

Algorithm 2 Modification-based algorithm: `NameSimilarity`

INPUT: A set of tests T

OUTPUT: A set of tuples R , with tests and corresponding priority (T_i, p)

PARAMETERS: A set of filenames F

A set of testnames N

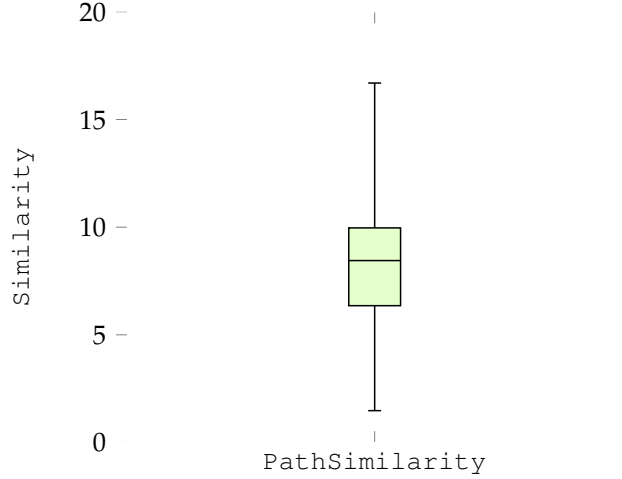
```

 $R \leftarrow \{\}$ 
for all  $T_i$  in  $T$  do
   $F \leftarrow \text{GetChangedFileNames}(T_i)$ 
   $F \leftarrow \text{PreProcessing}(F)$ 
  for all  $F_i$  in  $F$  do
     $N \leftarrow \text{SplitByCamelCase}(T_i.\text{name})$ 
     $p \leftarrow \text{GetTextSimilarity}(F_i, N)$ 
  end for
   $R \text{ add } (T_i, p)$ 
end for
return  $R$ 

```

To acknowledge that there exists a similarity between the files that got changed and the test cases, a minor experiment was carried out. The experiment showed that there is a wide distribution of similarity scores. It confirms that there exists a structure in the similarity measure, that might be correlated with failures. Figure 3.4 shows the result of the experiment plotting the distribution of path similarity scores in a box-plot.

Figure 3.4: Distribution of path similarity scores



3.7 Machine learning approaches

Since the samples in the data set refer to test runs with known results, binary labeled as success or failure, using supervised learning is a natural approach. As presented in Section 2.7, an example of this is the study carried out by Busjaeger et al. [4] in which a supervised learning approach was used to learn a model to rank. The implementation in this thesis resembles the study performed by Busjaeger et al., but the ranking task is reduced to a point-wise algorithm, which is standard classification. Instead of the list-wise algorithm used by Busjaeger et al. that aims to predict the optimal permutation of the list of tests. In this context a point-wise algorithm refers to classifying if a test case will fail or pass given its history and the change that it aims to test. These predictions are then used as priority levels, where the test cases classified as failure refers to high priority and success refers to low priority. Additionally, the problem was reformulated to a regression task. These approaches were chosen to investigate new approaches of combining machine learning with RTP, compared to previous research by Busjaeger et al.. The following formulations were used to rephrase the RTP problem;

- *Formulating RTP as a classification problem:* Given a set of tests and a code change, classify all tests as failure and success depending on features of the code change and the history of the test. Prioritize all tests that are classified as failure first, and secondly all test that are classified as success.
- *Formulating RTP as a regression problem:* Given a set of tests and a code change, calculate the probability of all tests of resulting in a failure depending on features of the code change and the history of the tests. Run all tests sorted on the calculated probability.

To train the model, feature vectors of each samples were created. Experiments were performed with different feature sets, the following section will describe the implemented feature sets.

3.7.1 Feature selection

Combining different RTP methods as features to a machine learning model is proven to be powerful by Busjaeger et al.. Therefore, similar to that study, the best performing history-based approach and

the best modification-based approach are chosen as features for the first experiment. To build the feature vectors, corresponding functions in the `LastFailure` and `PathSimilarity` approaches were called to calculate the features. The best history-based approach was easy to determine because `LastFailure` performed significantly better than the others. The best modification-based approach could be based on either approach because they performed equally well. `PathSimilarity` was chosen over `NameSimilarity`, as the variance of the result were lower for `PathSimilarity`. This machine learning approach corresponds to the first configuration called CA1 in Table 3.1.

The result from the paper by Busjaeger et al. shows that path text similarity is their strongest individual feature. The reason explained is that tests written to verify certain code often contain similar terms in their class or package names. Failure history and boosting new tests are also performing good individually. A feature that does not perform well in their setting is code coverage. Their explanation is the high degree of failures induced by non-code changes, that is not captured by code coverage. Nevertheless, they still point out that it is valuable for predicting that are not textually or temporally related to changes. This is a motivation for not focusing on gathering coverage information to build a coverage-based feature. Another difference in feature selection compared to [4], is that they incorporate the idea from [12] to boost unseen tests. This feature is in this implementation incorporated in the `LastFailure` feature, explained in Section 3.5. Busjaeger et al. perform normalization of all features, this is incorporated in this implementation as well.

Since the feature vector gets very short in the first configuration CA1, and there are more data stored to leverage, an additional experiment was performed, using as much information as possible from the database. This approach corresponds to the configuration called CA2 in Table 3.1. The feature of counting how many files that got changes to add another measure of modification, was inspired by the techniques used by Saha et al. [32].

To distinguish how much the `PathSimilarity` is contributing to the result of CA1 an experiment with only `LastFailure` was performed, CA3. Similarly CA4 corresponds to an experiment with only a `PathSimilarity` feature.

CA5 is using the set of features used in CA2 excluding the best performing history-based and modification-based approaches. This configuration was evaluated to be able to reason about if the additional features are redundant.

It was found that `LastFailure` performed well on its own, therefore an experiment with a set of different history limits was tested, CA6.

Table 3.1: List of machine learning approaches, abbreviations and corresponding feature sets

Name (Abbreviation)	feature set
ClassificationApproach (CA1) Best history and modification-based approaches	normalized failure history with limit 20, normalized path similarity score
ClassificationApproach (CA2) All simple features that exists in the data set including the best performing history-based and the best modification-based approaches	last test execution's result, normalized failure history with limit 1, 5, 10 and 20, normalized path similarity score, normalized last execution's duration, normalized number of files that got changed binary features of the test's ownership
ClassificationApproach (CA3) Only the best performing history approach	normalized last failure with history limit 20
ClassificationApproach (CA4) Only the best performing modification approach	normalized path similarity score
ClassificationApproach (CA5) All simple features that exists in the data set excluding the best performing history-based and modification-based approaches	last test executions' result, normalized last executions' duration, normalized number of files that got changed binary features of the test's ownership
ClassificationApproach (CA6) The best performing history approach with different history limits	normalized failure history with limit 1, 5, 10 and 20
RegressionApproach (RA) Best history and modification-based approaches	normalized failure history with limit 20, normalized path similarity score

3.7.2 Model selection

It is stated by Busjaeger et al., that SVM_{map} is a suitable model for the RTP problem. With the Python package `scikit-learn`, it is possible to reproduce a similar model. SVM was mainly chosen as model for this investigation because it was used by Busjaeger et al. and because it can be configured to perform a regression task easily. Other models were not investigated. The reason why SVM_{map} is suitable for the problem described by Busjaeger et al. is that it is specifically designed to maximize average precision given binary labeled data.

The Python module `Scikit-learn` was used to implement the SVM-model. The package includes a wide range of machine learning algorithms for medium-scale supervised and unsupervised problems [25]. The SVM model is implemented with a class weight on the failure class as an additional parameter. It compensates for a skewed training set. Which is required since if this weight was not added, the classifier would perform well in terms of classification, if it classifies everything as success. In that case no prioritization is done since all samples are classified as one class, hence there is only one prioritization level. Adding the weight to the failure class, is adding more tests to the high prioritization level.

To be able to rank the tests within the classes, the SVM was extended to do regression, predicting probabilities of belonging to the failure class. Adding an argument when creating the model, transforms the SVM classifier to perform a regression task. This experiment is referred to as RA in Table 3.1. It uses the same feature set as CA1.

The model was trained on 80% of the data and evaluated on 20% of the data. The training data can not be used for validating the model since it has already seen those samples and therefore already know the answers to them. 80% of the data was used for training to be able to train the model on as much data as possible and still have a sufficient amount of data to test the approaches on.

3.7.3 Pseudo code

The Algorithm 3.7.3 represents pseudo code for performing both `ClassificationApproach` and `RegressionApproach`. The difference between the approaches is how the pre-trained SVM

model M is configured. The function *BuildFeatures* is building the feature set based on functions in *LastFailure* and *PathSimilarity* or other data from the dataset with respect to the features in Table 3.1. The *Predict* function represents functionality in the Python module *scikit-learn*, it returns the class belonging, 1 or 0, in the classification case and a probability of belonging to the failure class in the regression case.

Algorithm 3 Machine Learning algorithm: ClassificationApproach / RegressionApproach

INPUT: A set of tests T

OUTPUT: A set of tuples R , with tests and corresponding priority (T_i, p)

PARAMETERS: A set of filenames F

A pre-trained SVM model M

```

 $R \leftarrow \{\}$ 
for all  $T_i$  in  $T$  do
   $F \leftarrow \text{BuildFeatures}(T_i)$ 
   $p \leftarrow M.\text{Predict}(T_i)$ 
   $R \text{ add } (T_i, p)$ 
end for
return  $R$ 

```

4 Results

This chapter presents the findings of the research, focusing on the three implemented metrics; APFD score, average percentage of time until the first failure and average rank of the first failure. The implemented approaches are visualized in Table 4.1.

Two additional experiments were performed, the first to determine the history limit of the approach `LastFailure` and secondly to determine which feature set that performs best of the machine learning approaches. The results of those experiments are represented by APFD scores.

All experiments are performed on the same test set containing test result and code change data. The test set consists of 20% of the total data set, that is 840 builds. The metrics only considers builds that are related to at least one failing test, that is approximately 450 builds. The remaining 80% of the data set was only used for training the machine learning model for the machine learning-based approaches.

The baseline approaches are plotted in gray in all graphs, to distinguish them easily from the more novel approaches.

Table 4.1: List of approaches and abbreviations

Name	Abbreviation
RandomBaseline	RB
NoPrioritization	NP
LastExecutionAndLastFailure	LELF
LastBuildExecution	LE
LastFailure with history limit = 20	LF
PathSimilarity	PS
NameSimilarity	NS
ClassificationApproach, with varying feature sets	CA1,..., CA6
ClassificationApproach, same features as CA1 based on best history and modification-based approaches	CA
RegressionApproach, same feature set as CA	RA
LastFailure with a varying history limit	H1, H2, H3, ... , H30

4.1 APFD

Each approach's distribution of APFD scores is plotted as a boxplot. The line inside each box corresponds to the median score of the approach, the end of the lines correspond to the maximum and minimum score of the approach, and the start and end of the box corresponds to the first and third quartile of the scores.

Figure 4.1 shows a boxplot of the APFD-scores of the investigated approaches. The higher the score the faster the failures are found, hence the better the approach is.

Figure 4.1: APFD scores of approaches

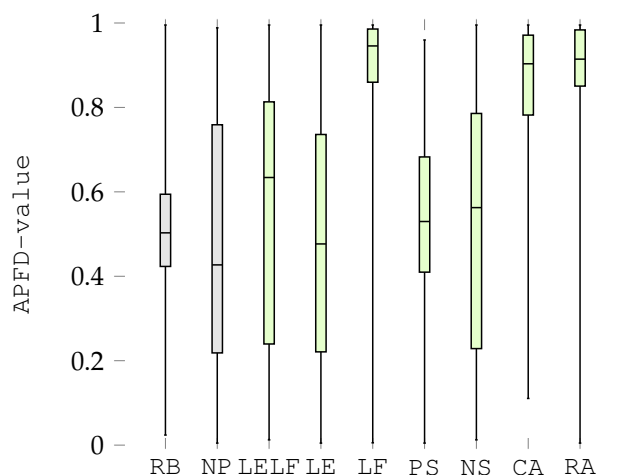


Figure 4.2 and Figure 4.3 shows how the `LastFailure` approach performs with varying history limitations.

Figure 4.2: APFD scores of recent failure history approaches

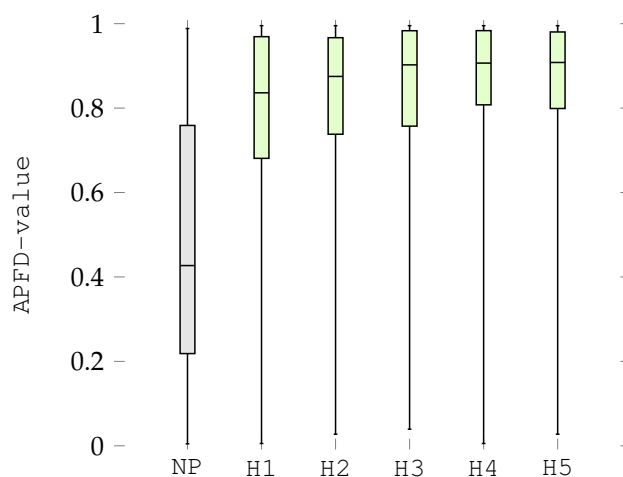


Figure 4.3: APFD scores of failure history approaches

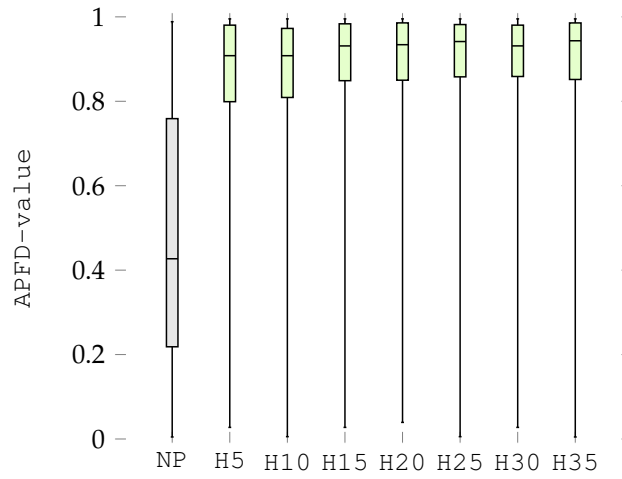
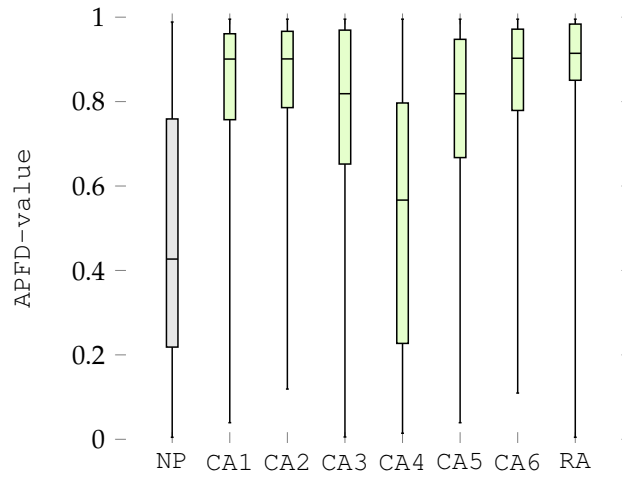


Table 3.1 in Section 3.7 lists the variations of the implemented machine learning approaches, and Figure 4.4 shows the corresponding APFD scores. The variations of ClassificationApproaches is represented by CA1, CA2, CA3, CA4, CA5, CA6 and RA1 corresponds to a RegressionApproach.

Figure 4.4: APFD scores of machine learning approaches



4.2 Time & Rank

Figure 4.5 shows the average of the percentage of time until the first failure for each approach. The lower the score the faster the first failure was found.

Figure 4.5: Time to first failure

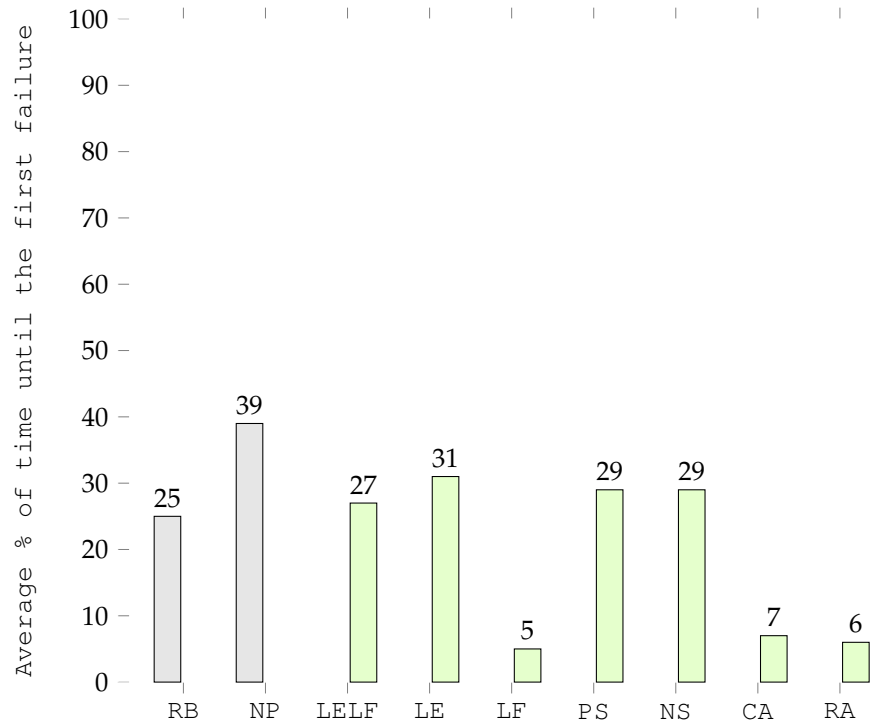
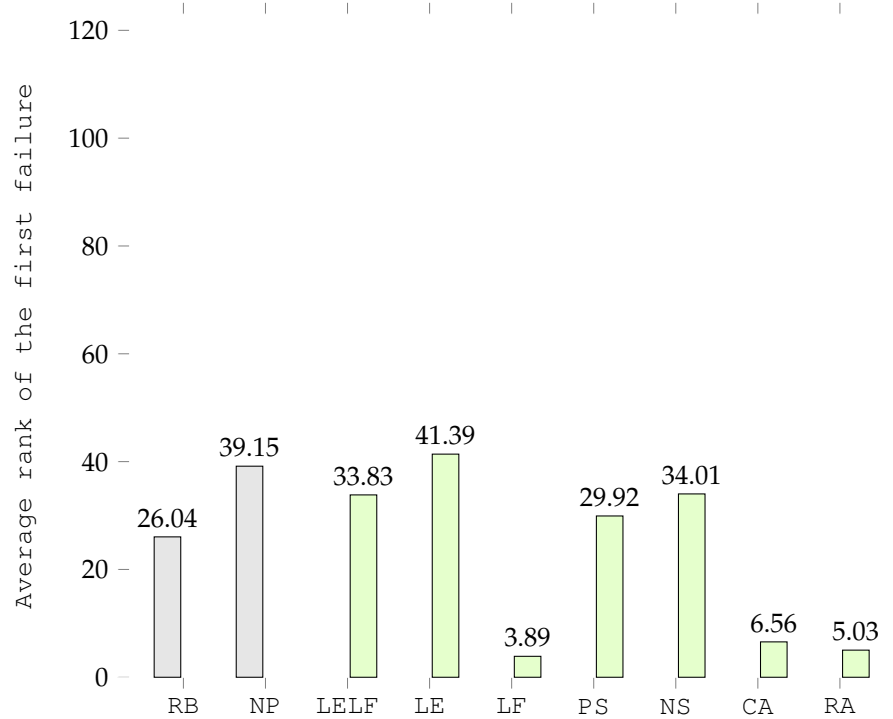


Figure 4.6 shows the average rank of the first failure. The lower the score the earlier the first failure was found.

Figure 4.6: Rank of first failure





5 Discussion

This chapter discusses several aspects of this thesis; the findings based on the results showed in Chapter 4, questioning the chosen method presented in Chapter 3 and reasoning about the work in a wider context with respect to ethical and social impact.

5.1 Results

5.1.1 Findings

One unanticipated finding was that there were only three approaches that performed significantly better than both baselines approaches on all metrics, those are `LastFailure`, `ClassificationApproach` and `RegressionApproach`. Possible explanations of why the remaining approaches performed worse is discussed in the following subsection.

The most obvious finding to emerge from all metrics visualized in Chapter 4 is that `LastFailure` is the only feature that performs well on its own. It even performs better than the machine learning approaches when it comes to the metrics regarding rank and time. An implication of this is possibly that `LastFailure` assigns more priority levels than the `ClassificationApproach`. `LastFailure` assigns 21 different priority levels based on failure history, values from 0 to 20, if the history limit is 20. The `ClassificationApproach` only assigns two priority levels. `RegressionApproach`, on the other hand, is assigning probabilities of belonging to the failure class, which is a continuous value, so that explanation does not apply in that case. It might be the case that the machine learning approaches need more features to be able to perform better. Further investigation is needed to determine if the machine learning approaches can outperform `LastFailure`.

One interesting finding is that running tests in random order is better than the original order based on all metrics. This finding is in agreement with those obtained by [30] that also concluded this.

5.1.2 Results Compared to Related Research

`LastExecutionAndLastFailure` resembles the approach presented by Elbaum et al. [12]. `LastExecutionAndLastFailure` improves on all aspects of the metrics, presented in Chapter 4 compared to the *retest-all* approach applied by the company today, that is represented by `NoPrioritization`. In that sense, the result is similar to the findings in Elbaum et al.'s study. However, the other baseline approach, `RandomBaseline`, performs better on average with respect

to finding the first failure in terms of rank and percentage of time. Since the approach combines `LastBuildExecution` and `LastFailure` and since the result of `LastFailure` outperforms all other approaches there is no use of considering `LastBuildExecution`. A possible explanation for this might be that all tests run almost every time, making the measure last execution similar for all tests, and therefore containing too slight difference to use for prioritization. The reason presented for considering it by Elbaum et al. is that a test suite selection is performed in the pre-merge testing phase. This is making this measure more informative since all tests in the post-merge phase do not get selected to run in the pre-merge phase.

Combining selection techniques with prioritization techniques would also raise the importance of which code change that is used in the modification-based approaches. The alternatives are; using the code change on the current branch or for every test using the code change since that specific test last ran. If the tests do not run for all builds, the difference between the two alternatives will be larger and the impact of the choice will increase. More investigation needs to be done to ensure which code change that is the most correlated with revealing failures.

A strong relationship between learning to rank and RTP has been reported [4]. However, the results of this investigation are unable to demonstrate that same outperforming result for the machine learning approaches compared to the results previously presented. There are, however, possible explanations since both the dataset and the implementation differ from that investigation. These differences are discussed in the following paragraphs. Another possible explanation for these results may be the lack of adequate features. The result of comparing different feature sets also indicates this. The different feature sets investigated are presented in Table 3.1 and the results of running those configurations are visualized in Figure 4.4. The configuration CA1 has only two features, compared to CA2 that uses more attributes of the data, nevertheless, they get very similar results. This indicates that the additional attributes are redundant. The experiment using the best performing history- and modification-based approaches as features, shows that they complement each other, as Busjaeger et al. state, they are good at indicting different faults [4]. The feature set composed by different history limits, CA6, contradicts this by yielding a similar result, showing that `LastFailure` is as good indicator at finding faults by itself. Removing the best performing history- and modification-based approaches as features decrease the result, shown by CA5.

`PathSimilarity` was one of the best performing features in previous research [4], however, the result for `PathSimilarity` and `NameSimilarity` in this investigation is not as outstanding. Both approaches have a slightly better median at the APFD-score compared to the baseline approaches and find faults faster than original order. This result might depend on the similarity function used, or by the fact that it heavily relies on the hypothesis that the naming of the files and tests are correlated. It can be a characteristic that does not correspond to the data set under investigation. More research is required to determine the efficacy of modification-based approaches on this dataset, similar techniques as performed by Noor et al. [23] with content-based approaches is necessary to determine the relationship between which modifications that have been made and which faults it causes. In addition, applying other types of similarity functions such as corpus-based similarity functions as used in related research [4, 32] would be desirable.

The result of the experiment performed to determine the value of the history limit of `LastFailure`, shows that the most recent history is the most informative. Only looking at the last time the test run is a good indicator for prioritization. It is in general the best indicator for prioritization based on the result in Figure 4.1 and configuration CA6 in Figure 4.4.

5.2 Method

5.2.1 Data set

The subset of data that is used impacts the result. This subsection questions the choice of data set.

The APFD scores show that all approaches have a wide distribution of scores. This is probably due to the characteristics of the data in combination with the metric. Similar structure is found in the results in Busjaeger et al.'s study [4].

It is arguable that the code change data should have been fetched in another way. There are two changes that are interesting to this problem; the change that was made on the specific branch or all changes that were made since the test case last executed. This means that fetching the diff from the previous build on the branch might be misleading, because it is not certain that the test ran for the previous build. This difference is explained in the Method chapter by Figure 3.2.

The data set is on a test case granularity, therefore this research was carried out on a test case granularity. In the related research different granularities of test have been executed. Elbaum et al. implemented their research on a test suite granularity [12].

The attributes that were referring to ids were not unique. The fact that it was solved by combining and hashing multiple attributes to create new ids, is difficult to verify. The technique used is explained in Section 3.2. Counting the number of test results for each build yields an expected value. One way to make sure that this is right is to store this attribute from the build system.

Since this is an investigation based on industrial data, but no experiment is performed in the CI environment, none of the challenges presented in Section 2.8 regarding implementation were faced. However, the investigated approaches are well suited for such an implementation based on the similarities with the experiments performed in an industrial environment, presented in Section 2.8. Since the testing process is different, implementing it in the CI system would probably face difficulties with respect to the differences. This is an advantage of investigating one platform at a time, and one type of test because it is clear where it would suite in the CI process.

Overlooking the advantages of investigating a subset of the whole data set instead of the whole data set, there are also advantages of increasing the size of the dataset. Increasing the size of the data set would result in more reliable result. When increasing the size of the data set, consideration of additional tools to handle the data processing is needed. A natural way to increase the data set is to use data from multiple platforms and types of tests.

There is a possibility that the data set will change drastically in the future, if the initiative on fixing flaky tests succeeds. If the tests were stable the data would not contain as many failures as it does today, hence the history of faults might not be as informative for prioritization as it is today. However, fixing flaky tests and using test prioritization techniques for aiding the problems described in Chapter 1 do not need to be mutually exclusive, because they are attacking the problem from different angles.

5.2.2 Approaches

The hypothesis used when choosing features for the machine learning approaches, is that each feature excels for certain type of changes and tests. The use of multiple heuristic was inspired by the implementation by Busjaeger et al. [4]. The difference in result, between this implementation and the Busjaeger et al.'s, may partly be explained by the difference in machine learning task. Learning to rank is as stated in Section 2.7.4 more advanced than the classification approach applied in this investigation. Another difference that most probably contributes to the difference in result is the difference in feature sets. First, this investigation does not have any coverage based feature. This makes it harder to find the faults related to interdependencies between seemingly unrelated parts of the system under test, based on the heuristic. In addition, more research has been carried out within RTP that was shortly mentioned in this report, e.g. requirement-based, generic-based. This research would probably also be useful to incorporate as additional complementary features.

More work can be done to improve the machine learning method. There are a lot of machine learning models, and only a few are presented and described in Section 2.7. In addition, each model can be tweaked in different ways. One advantage of using Naive Bayes classifier that is presented is that it is fast. That would be beneficial if experiments with a larger data sets are conducted in the future. Using a feature set with multiple heuristics as this project might be good with respect to the downside of Naive Bayes classifier, that is an assumption that all features are conditionally independent. The decision tree model is known to suffer from over-fitting the training data. However, it would be interesting to investigate the improved version of decision trees that is Random Forest.

5.2.3 Evaluation

Refinements of the APFD metric have been proposed and described in Section 2.9. With the current implementation of the APFD value, there are two known flaws; all faults are treated as equal severity and all test cases are treated as equal cost.

The test cost are captured in the metric *average percentage of time to first failure*. In this implementation, cost is represented by execution time. Which in itself could be improved by a more accurate measure, since the execution time depends on other variables than the test itself.

There is no other metric that captures the flaw regarding fault severity. One advantage of not considering this flaw is that it forces developers to think of faults as the same severity, which means faults that otherwise would have been considered as low severity would be fixed. Especially if the test run was configured to stop at the first failure.

Knowing the severity of faults would be valuable, not only for prioritization but for the feedback loop to the developers in general. There is currently no obvious information in the data set about the severity of the faults. However, that data could potentially be retrieved from error messages in combination with if it is an error or failure. More investigation is needed to know how to construct such a metric on this particular dataset and how it has been done in similar settings.

The largest threat of validity of the additional metrics used, *average percentage of time to first failure* and *average rank of first failure*, is that they are not used in any related research. However, it is graspable metrics for the company, and data for alternative metrics, such as *gained hours* used by Elbaum et al. [12] was not gathered.

5.2.4 Related research

Mainly publications from conferences and journals published by well-known organizations are used as references for this thesis, supplementary books have been referenced for basic concepts. The method of the project relies heavily on two papers, therefore the validity of those papers is discussed. The papers are;

1. Busjaeger, Benjamin, and Tao, Xie *Learning for test prioritization: an industrial case study* [4] (2016) - cited by 1
2. Elbaum, Sebastian and Rothermel, Gregg and Penix, John *Techniques for improving regression testing in continuous integration development environments* [12] (2014) - cited by 48

The fact that the first paper only been cited once, could be seen as unreliable but on the other hand, the study was conducted last year, that could be an explanation for the low number of citations. The same reasoning goes for the second paper, it has a fair amount of citations for being as recent as it is. In addition, both papers are published by the well-known organization ACM¹, that could be seen as a reliable source of information.

5.3 The work in a wider context

The investigation of this thesis helps to understand the link between RTP research and its implementation in industrial environments. Currently there are few companies that have incorporated the idea of test optimization into their CI system, but a lot of growing companies will face the problems of growth and will be forced to act on it. This is a motivation for carrying out this research, not only for Spotify, but for all fast growing companies. To the knowledge of the author, this is the first larger project at Spotify addressing this solution to aid growth.

The fact that not so many companies have incorporated test optimization yet is natural, the problem of growth does not apply to small code bases. For small code bases the *retest-all* approach is good enough because all tests can run in a reasonable time without impacting developers negatively. With that in mind, this investigation is also helping companies in the long term to be more productive,

¹<http://dl.acm.org/>

and it all boils down to that it enables developers to be more effective. Nevertheless, it is a waste of computational resources to not run tests in a smarter way than the *retest-all* approach, and the waste grows larger as the tests consume more resources. Aiming for minimizing computer resources is therefore desirable.

Solving problems with the mindset of machine learning have been revolutionary in several different fields where it has been applied. Applying it to a new domain is therefore of great interest. This study did not revolutionize the field of RTP, but it confirmed that it works, and that is hopefully encouraging for others to try other ways of combining RTP and machine learning.



6 Conclusion

The present study was designed to determine the effect of RTP techniques using machine learning and utilizes Spotify's historical build and test data. The conclusions are drawn based on the research questions formulated in Chapter 1.

To measure the ability to increase the fault detection rate the commonly used metric APFD was used. The *retest-all* approach is represented by `NoPrioritization`, and `RandomBaseLine` represents random prioritization.

6.1 Research questions

6.1.1 Do history-based, modification-based and machine learning approaches affect the fault detection rate, compared to the retest-all approach used by the company today and random prioritization?

History-based approaches

The most powerful approach in general is a history-based approach, `LastFailure`. It prioritizes tests based on how many times the test failed in the past and it gets the best distribution of APFD-scores. It is also concluded from the experiment determining the history limit, that the most recent history is the most significant to get a good result.

The other history-based approaches that were investigated, `LastBuildExecution` and `LastExecutionAndLastFailure`, perform similar to the baseline approaches. This result shows that the time difference between the last test execution is not correlated with failure. It is discussed that the reason for that is that Spotify run all tests at all times making the time difference small between tests.

Modification-based approaches

There are two modification-based approaches implemented, `PathSimilarity` and `NameSimilarity`. They perform similar to the baseline approaches. It can be concluded that more investigation needs to be performed to know if modification-based approaches are correlated with the fault detection rate.

Machine learning approaches

The machine learning approaches, `ClassificationApproach` and `RegressionApproach`, both improve the fault detection rate compared to both random prioritization and no prioritization. The fact that `RegressionApproach` prioritizes the tests on a finer granularity makes it perform better than the `ClassificationApproach`.

Summary

The conclusions are summarized in Table 6.2. The abbreviations of the approaches are explained in Table 4.1.

Table 6.1: All approaches effect on APFD values

Effect on APFD score	Approach
Improved	LF, CA, RA
Similar to baseline	LELF, LE, PS, NS

6.1.2 Do classification approaches using feature sets based on previous research in RTP affect the fault detection rate, compared to using a feature set with randomly selected sets of metadata?

The classification approaches investigated used different configurations, with respect to different feature sets. Different feature sets contain different information for the classifier to learn from, and it is therefore an important part of the machine learning process. The different feature sets investigated are presented in Table 3.1. The results of running those configurations are visualized in Figure 4.4. The results show that the feature set has a great impact on the result.

From Figure 4.4 it can be concluded that the best performing configurations of the classification approaches investigated are; CA1, CA2 and CA6. They are all improving the fault detection rate significantly. The common feature for those configurations is the best performing history-based approach, where CA6 only contains history-based features. The configurations CA3 and CA5 are also improving the fault detection rate, but not as significant. The configuration CA3 only contains the best performing history-based feature and CA5 contains a randomly selected feature set excluding the best performing approaches based on previous research.

The configuration CA4 is the only configuration that does not improve the fault detection rate. It performs as the baseline approaches. The only feature used in that configuration is the best performing modification-based approach. Even if it is based on the best performing modification-based approach, that approach is not performing well on its own based on the result in Figure 4.1. For that reason, it is not surprising that CA4 does not perform well either.

It can be concluded that the best performing history-based approach is a feature that improves the fault detection rate remarkably. The best performing modification-based approach on the other hand, is not as informative on its own. The configurations that combines the best performing history-based and modification-based approaches, CA1 and CA2, improve the fault detection rate compared to a randomly selected feature set CA5. However, using a random feature set also improve the fault detection rate, compared to the baseline approaches. The conclusions are summarized in Table 6.2.

Table 6.2: Different configurations of the classification approach's effect on APFD values

Effect on APFD score	Configuration of <code>ClassificationApproach</code>
Improved significantly	CA1, CA2, CA6
Improved	CA3, CA5
Similar to baseline	CA4

6.2 Future work

The natural continuation of this investigation is to investigate the best performing approaches on a larger data set containing data from multiple clients and types of tests. If that shows similar positive results, it should be integrated in the CI environment, and measurements similar to Elbaum et al.'s study [12] could be performed.

To be able to do a more accurate comparison between this thesis and the previous work that has been done combining RTP with machine learning [4], a feature related to code coverage needs to be added. Also more work could be done to investigate the features and the model.

One common detail from the most influential related research by Elbaum et al. [12] and Busjaeger et al. [4] is that both papers are investigating test selection in combination with test prioritization. Elbaum et al.'s history-based selection is performed on pre-merge tests and history-based prioritization on post-merge tests. Busjaeger et al.'s prioritization-based selection is performed, meaning selecting the top ranked test in the prioritized permutation of tests. To get as powerful test optimization as possible, combining test selection and prioritization should be considered as future work.



Bibliography

- [1] Boris Beizer. *Software testing techniques*. Dreamtech Press, 2003.
- [2] Christopher M Bishop. “Pattern recognition”. In: *Machine Learning* 128 (2006), pp. 1–58.
- [3] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. “Parallel symbolic execution for automated real-world software testing”. In: *Proceedings of the sixth conference on Computer systems*. ACM. 2011, pp. 183–198.
- [4] Benjamin Busjaeger and Tao Xie. “Learning for test prioritization: an industrial case study”. In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM. 2016, pp. 975–980.
- [5] Ryan Carlson, Hyunsook Do, and Anne Denton. “A clustering approach to improving test case prioritization: An industrial case study”. In: *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*. IEEE. 2011, pp. 382–391.
- [6] Jacek Czerwinka, Rajiv Das, Nachiappan Nagappan, Alex Tarvo, and Alex Teterov. “Crane: Failure prediction, change analysis and test prioritization in practice—experiences from windows”. In: *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*. IEEE. 2011, pp. 357–366.
- [7] Hyunsook Do, Gregg Rothermel, and Alex Kinneer. “Empirical studies of test case prioritization in a JUnit testing environment”. In: *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*. IEEE. 2004, pp. 113–124.
- [8] Sebastian Elbaum, David Gable, and Gregg Rothermel. “The impact of software evolution on code coverage information”. In: *Proceedings of the IEEE International Conference on Software Maintenance (ICSM’01)*. IEEE Computer Society. 2001, p. 170.
- [9] Sebastian Elbaum, Alexey G Malishevsky, and Gregg Rothermel. *Prioritizing test cases for regression testing*. Vol. 25. 5. ACM, 2000.
- [10] Sebastian Elbaum, Alexey G Malishevsky, and Gregg Rothermel. “Test case prioritization: A family of empirical studies”. In: *IEEE transactions on software engineering* 28.2 (2002), pp. 159–182.
- [11] Sebastian Elbaum, Alexey Malishevsky, and Gregg Rothermel. “Incorporating varying test costs and fault severities into test case prioritization”. In: *Proceedings of the 23rd International Conference on Software Engineering*. IEEE Computer Society. 2001, pp. 329–338.

- [12] Sebastian Elbaum, Gregg Rothermel, and John Penix. "Techniques for improving regression testing in continuous integration development environments". In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM. 2014, pp. 235–245.
- [13] Wael H Gomaa and Aly A Fahmy. "A survey of text similarity approaches". In: *International Journal of Computer Applications* 68.13 (2013).
- [14] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation (Adobe Reader)*. Pearson Education, 2010.
- [15] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An introduction to statistical learning*. Vol. 6. Springer, 2013.
- [16] Jung-Min Kim and Adam Porter. "A history-based test prioritization technique for regression testing in resource constrained environments". In: *Proceedings of the 24th international conference on software engineering*. ACM. 2002, pp. 119–129.
- [17] Moonzoo Kim, Yunho Kim, and Gregg Rothermel. "A scalable distributed concolic testing approach: An empirical evaluation". In: *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*. IEEE. 2012, pp. 340–349.
- [18] Hang Li. "Learning to rank for information retrieval and natural language processing". In: *Synthesis Lectures on Human Language Technologies* 7.3 (2014).
- [19] Christopher D Manning, Prabhakar Raghavan, Hinrich Schütze, et al. *Introduction to information retrieval*. Vol. 1. 1. Cambridge university press Cambridge, 2008.
- [20] Dusica Marijan, Arnaud Gotlieb, and Sagar Sen. "Test case prioritization for continuous regression testing: An industrial case study". In: *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*. IEEE. 2013, pp. 540–543.
- [21] Vangelis Metsis, Ion Androutsopoulos, and Georgios Paliouras. "Spam filtering with naive bayes-which naive bayes?" In: *CEAS*. Vol. 17. 2006, pp. 28–69.
- [22] Agastya Nanda, Senthil Mani, Saurabh Sinha, Mary Jean Harrold, and Alessandro Orso. "Regression testing in the presence of non-code changes". In: *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*. IEEE. 2011, pp. 21–30.
- [23] Tanzeem Bin Noor and Hadi Hemmati. "A similarity-based approach for test case prioritization using historical failure data". In: *Software Reliability Engineering (ISSRE), 2015 IEEE 26th International Symposium on*. IEEE. 2015, pp. 58–68.
- [24] Tanzeem Bin Noor and Hadi Hemmati. "Test case analytics: Mining test case traces to improve risk-driven testing". In: *Software Analytics (SWAN), 2015 IEEE 1st International Workshop on*. IEEE. 2015, pp. 13–16.
- [25] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. "Scikit-learn: Machine learning in Python". In: *Journal of Machine Learning Research* 12.Oct (2011), pp. 2825–2830.
- [26] R Pradeepa and K VimalaDevi. "Test Case Prioritization Based On Faults". In: *International Journal of Scientific Engineering and Technology* 3.7 (2014), pp. 856–860.
- [27] Xiao Qu, Myra B Cohen, and Katherine M Woolf. "Combinatorial interaction regression testing: A study of test case generation and prioritization". In: *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*. IEEE. 2007, pp. 255–264.
- [28] Shivani Rao and Avinash Kak. "Retrieval from software libraries for bug localization: a comparative study of generic and composite text models". In: *Proceedings of the 8th Working Conference on Mining Software Repositories*. ACM. 2011, pp. 43–52.

- [29] Meghan Revelle, Bogdan Dit, and Denys Poshyvanyk. "Using data fusion and web mining to support feature location in software". In: *Program Comprehension (ICPC), 2010 IEEE 18th International Conference on*. IEEE. 2010, pp. 14–23.
- [30] Gregg Rothermel, Roland H Untch, Chengyun Chu, and Mary Jean Harrold. "Test case prioritization: An empirical study". In: *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on*. IEEE. 1999, pp. 179–188.
- [31] Ripon K Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E Perry. "Improving bug localization using structured information retrieval". In: *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. IEEE. 2013, pp. 345–355.
- [32] Ripon K Saha, Lingming Zhang, Sarfraz Khurshid, and Dewayne E Perry. "An information retrieval approach for regression test prioritization based on program changes". In: *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*. Vol. 1. IEEE. 2015, pp. 268–279.
- [33] James Shore et al. *The art of agile development*. " O'Reilly Media, Inc.", 2007.
- [34] Amitabh Srivastava and Jay Thiagarajan. "Effectively prioritizing tests in development environment". In: *ACM SIGSOFT Software Engineering Notes*. Vol. 27. 4. ACM. 2002, pp. 97–106.
- [35] Matt Staats, Pablo Loyola, and Gregg Rothermel. "Oracle-centric test case prioritization". In: *Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on*. IEEE. 2012, pp. 311–320.
- [36] L White. "Insights IntoRegressionTesting". In: *Proceedings of the Conference on Software Maintenance-1989*. Vol. 1. IEEE Computer Society Press. 1989, pp. 60–69.
- [37] W Eric Wong, Joseph R Horgan, Saul London, and Hiralal Agrawal. "A study of effective regression testing in practice". In: *Software Reliability Engineering, 1997. Proceedings., The Eighth International Symposium on*. IEEE. 1997, pp. 264–274.
- [38] Shin Yoo and Mark Harman. "Regression testing minimization, selection and prioritization: a survey". In: *Software Testing, Verification and Reliability 22.2 (2012)*, pp. 67–120.
- [39] Shin Yoo, Robert Nilsson, and Mark Harman. "Faster fault finding at Google using multi objective regression test optimisation". In: *8th European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'11), Szeged, Hungary*. 2011.
- [40] Michal Young. *Software testing and analysis: process, principles, and techniques*. John Wiley & Sons, 2008.
- [41] Lingming Zhang, Dan Hao, Lu Zhang, Gregg Rothermel, and Hong Mei. "Bridging the gap between the total and additional test-case prioritization strategies". In: *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press. 2013, pp. 192–201.
- [42] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. "Predicting defects for eclipse". In: *Proceedings of the third international workshop on predictor models in software engineering*. IEEE Computer Society. 2007, p. 9.