Machine Learning to Uncover Correlations Between Software Code Changes and Test Results

Master's thesis in Software Engineering and Management

NEGAR FAZELI

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2017

Master's thesis 2017

Machine Learning to Uncover Correlations Between Software Code Changes and Test Results

NEGAR FAZELI

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2017

Machine Learning to Uncover
Correlations Between Software Code Changes and Test Results
NEGAR FAZELI

Supervisor: Robert Feldt, Department of Computer Science and Engineering
Examiner: Richard Berntsson Svensson, Department of Computer Science and Engineering

Master's Thesis 2017
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Machine Learning to Uncover Correlations Between Software Code Changes and Test Results.

Gothenburg, Sweden 2017

Machine Learning to Uncover
Correlations Between Software Code Changes and Test Results
NEGAR FAZELI
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

STATISTICS show that many large software companies, particularly those dealing with large-scale legacy systems, ultimately face an ever-growing code base. As the product grows, it becomes increasingly difficult to adequately test new changes in the code and maintain quality at a low cost without running a large number of test cases [1, 2, 3]. So a common problem with such products is that, thoroughly testing changes to the source code can become prohibitively time consuming and generally adhoc testing of the product by the designers and testers can potentially miss bugs and errors that can be detrimental to the quality of the end product.

In this thesis we setout to address this problem and investigate the possibility of using machine learning to conduct more economical testing procedures. To this end, the goal of this thesis is to create a test execution model which uses supervised machine learning techniques to predict potential points of failure in a set of tests. This will help to reduce the number of test cases needed to be executed in order to test changes in code.

We have to state that this approach for automatic testing and test selection has been thoroughly investigated before. The proposed state-of-the-art algorithms for this purpose, however, rely on detailed data that includes e.g., the amount of changes made in each and all code modules, their importance and structure of the tests. In contrast, in this thesis we do not have access to such data. So in turn, in this thesis we investigate the possibility of using well-established machine learning techniques for intelligent test selection using the available data, and check whether it is possible to achieve satisfactory results using the information provided by this data. In case the results are not satisfactory this can potentially provide guidelines on how to modify the logging procedure of changes made to the modules and the test results report so as to better facilitate the use of available machine learning techniques.

This work is a case study conducted at a large telecom company - more specifically, in the Session Border Gateway (SBG) product, which is a node within an IP Multimedia Subsystem (IMS) solution. The model is trained on the extracted data concerning the SBG code base and data from nightly builds from October 1st 2014 to August 31st 2015. Having collected the necessary data, we design relevant features based on available information in the data and interviews with the experts working with the product and the testing of the product. We then use use logistic regression and random forest algorithms for training the models or predictors for the test cases.

One of the benefits of this work is to increase the quality and maintainability of the SBG software by creating faster feedback loops, hence resulting in cost savings and higher customer satisfaction [4]. We believe that this research can be of interest to anyone in the design organization of a large software company.

Acknowledgements

This project would have never been possible without the contribution of several people. First and foremost, I would like to thank my supervisor Robert Feldt. Thank you for the great support, and for guiding me through the project. Also many thanks to Anette Carlson, my very understanding and caring manager - you are the best.

Last but not least, I have to thank my family and friends from the bottom of my heart. Thanks Sina for always being there, and for being a true friend. My special thanks go to Gaia, Armin, Claudia and Ramin for giving me very valuable comments and bringing joy and fun through my tough times.

Negar Fazeli, Stockholm 29/05/2017

Contents

1 Introduction

DUring the last decade, testing of software products has become a separate process and has received the attention of project stakeholders and business sponsors alike [5]. In general, the testing of a piece of software corresponds to evaluating the said software with a series of test cases [6]. The main reason for testing software is to get an understanding about the software quality or acceptability and to discover problems [6]. Possible objectives include giving feedback, finding or preventing failures, providing confidence and measuring quality [7]. Due to the rapid increase in the sophistication of software products, many academics claim that testing has become the most important and time consuming part of the software development lifecycle [8]. Moreover, it has been observed that testing techniques have struggled to keep up with the fast trends in software development paradigms, and therefore need more attention [9].

Software companies invest a considerable amount of time and effort into developing and designing new software products. Ensuring consistent quality of the product requires extensive and persistent testing. In fact, commonly more than 25 percent [10, 11] of the time is spent on fixing failures and bugs in the software as well as making sure that the software product meets its requirements specification [12]. In fact, in large companies, especially ones developing complex or legacy systems, it becomes increasingly difficult to make changes to the system as the product grows [13], since every change needs to be thoroughly tested. Particularly, generally service level agreements for telecommunication companies are in the order of 99.999 percent. This leaves a very small margin for errors and hence requires extensive testing of the product. In the case of complex software products that come with many millions lines of code and several interdependent modules, testing the product requires running many test cases, which can take many hours to complete. As software products grow more complex, problems associated with testing are exacerbated [14], and developers and testers often face many challenges in testing. Many testing techniques have traditionally focused on homogeneous, non-distributed software of smaller size, whereas modern software systems include diverse, highly distributed, and dynamic components [15]. Another big hurdle is implementing test oracles. Test oracles are often designed to check whether the output produced by a test case matches the expected behaviour of the program [16]. Implementing and executing such oracles has become a challenge as test case generation continue to improve and massive amounts of test cases are generated automatically [15]. One also has to bear in mind that once software is deployed and accepted by the customers, it needs to be maintained until it becomes obsolete, and the maintenance activities need to be adaptive, corrective, and predictive. Oftentimes maintenance is not included in the core part of software engineering, while as Sharon and Madhusudhan claim in [17], almost 70 percent of time and resources are allocated to software maintenance. Moreover, the authors in [17] and [18] claim that maintenance is the most expensive part of the software life cycle.

There are several other problems related to testing. For instance, in many cases the testing of the software becomes so costly and time consuming that it stagnates the speed and dynamicity of the development process. In order to address this issue, many software companies overlook tests in order to meet tight deadlines, thus testing is often sacrificed to hasten feature deliveries. Since software product complexity is going to grow as we progress, such issues will persist and tend to become worse. Despite this, one must notice that many of the test cases are not relevant or are not needed to be executed. As a result, in order to address the aforementioned problems, there have been studies focusing on the area of test prioritization to alleviate the costly nature of testing [19]. Due to high costs and other challenges mentioned above there has been a growth in prioritizing test cases in a greedy way in order to ensure the largest possible coverage [20].

Our research is focusing on issues related to test prioritization and selection of relevant tests. Oftentimes, the designers and testers of the system will try to identify the system areas affected by the code change and will only run the corresponding test suites, because running a complete regression suite or all the test suites in their entirety can take several days. Sometimes such attempts to shorten testing times can result in problems, specially due to high service level agreements, because most systems have complicated dependencies and interdependencies, and it is not always obvious how a change in one part of the system can affect other parts of it. However, manually isolating the relevant test cases based on a set of changes to the code is very difficult and needs high level of experties and hence, generally companies tend to run if not all but the majority of the tests. This is to avoid missing bugs or errors created by the introduced changes to the code. It would thus be very valuable to have a prediction system tailored to the entire code base which could give an accurate suggestion of which test cases should be executed to test a particular change made in the code.

There are mainly three test categories, namely: black-box, white-box, and gray-box testing [5]. Black-box tests examine what a system is supposed to do by monitoring its output while providing specific test inputs. That is to say, they examine what the output is supposed to be based on specific inputs. These types of tests mainly focus on the functionality of the system [5, 11]. Function tests fall under the black-box category, and their main purpose is to examine what the system does. They are mainly important for ensuring that the software behaves according to its requirements specification. Function tests are the main test category which we are going to focus on in this thesis work. That is mainly due to more streamline possibility of automizing such testing frameworks. In contrast in white-box testing the tester examines the internal structure of the software product [5]. One example of tests in this category is unit test cases. Gray-box tests are a combination of the black-box and white-box tests. These testing schemes are generally more difficult to automate due to the complexity and specificity of the testing procedures.

The main objective of automating relevant test selection is to find almost all the possible faults or failures with the help of a few relevant test cases. There is a lot of research done and being conducted in the area of automated test case generation and selection using machine learning techniques. However, there is not enough evidence yet concerning the efficiency of such techniques [21, 22] and the use of predictive models for finding the best test coverage without needing to run test cases in their entirely based on changes in the code [16, 23, 24, 25, 26]. Furthermore, many of the proposed solutions require highly granular data in order to achieve satisfactory results [27, 28, 29], and not much has been done for situations where highly granular data is not available. Furthermore, it is also difficult to find tailored guidelines on how to gather proper data sets for this purpose. The latter is particularly of importance as many legacy software companies, that are the most in need of such automation schemes, lack highly granular data.

Through the work conducted in this thesis we have attempted to build a smart, adaptive system using machine learning techniques to automize the test selection. The work is conducted as a case study using the Session Border Gateway (SBG) software of an IP Multimedia Subsystem (IMS) solution of a large telecommunications company. During this process, we discuss the issues associated with the available data in this case study and propose guidelines on how to alleviate them and how to improve the quality of data.

1.1 Purpose

The main purpose of this research, in the context of the considered case study, is to investigate the possibility of building a prediction model which, given a current version of the modules of a software, accurately predicts affected test cases, returning these as output. This will be achieved by indicating which subset of test cases to execute for getting faster feedback, instead of running nightly builds and executing the entire selection of test suites, since in the worst case, analyzing the results of the entire nightly build execution and finding the relevant results can take up to several days. One of the approaches to generate a set of relevant test cases based on a set of changes to the code includes using machine learning. This approach has been considered extensively within the past two decades. We will provide a concise review of these approaches in the next chapter.

Notice that considering the high service level agreements, it is of utmost importance to that the set of recommended test cases does not miss any of the relevant test cases. This is because, in case that happened and the missed test case would have failed, this can have dire consequences both for the customer and the reputation of the company. So we intend to investigate the possibility of using machine learning algorithms for producing such recommender systems, or produce accurate predictors for predicting outcome of different test cases based on the available data. Providing such levels of accuracy has not been the main focus of the studies on this topic. Furthermore, our design come with very specific limitations concerning the structure and the amount of information that is available in the data. In this thesis we cover many of the details (as much as permitted by the policy of the company) and difficulties of the data extraction and its pre-processing which also includes feature design. This is one of the most important parts of any design approach using machine learning, however, it is commonly not discussed in detail in many of the available literature on this topic. We hope this can not only shed light on the obtained results of this thesis but also provide a case study for data extraction and pre-processing in a practical setting.

1.1.1 Problem statement

The problem statement is as follows: Based on the historical data of local changes in code $(\Delta_1, \ldots, \Delta_n)$ taken on a granularity of modules or blocks, and test results $(t_1, \ldots, t_m)$ construct a model which, given a code change, accurately predicts which tests are the most likely to fail as a result of the code change.

1.1.2 Research questions

Assume that we are given a set of changes made to certain modules of a code base. Is it possible to design and implement a prediction model which, given a set of changes in the module's version, can identify the most relevant test cases and reduce the testing time of the modified code?

To answer this question, some of the important points which need to be addressed are:

1. Is the provided data suitable for designing our automatic test selection algorithm?
   - What are the issues/limitations associated to the data?
   - How the quality of the data can be improved for our purpose?
2. Based on the available data, what features should be selected from the test results and code repository, or from the data in general?
3. Based on the available data, what are the suitable classifiers or predictors to use for extracting these test cases?
4. What are suitable ways to search for and to evaluate the performance of the proposed algortithm?

1.2 Research Motivation

As mentioned in the beginning of this chapter, with the advances in modern software release engineering such as continuous delivery there is a higher need to keep the quality in such a fast paced environment, particularly for telecommunication companies where often service level agreements are in the order of 99.999 percent. To try to maintain such levels we will be implementing a model which can predict failing test cases based on historical data gathered from source code revision history as well as nightly test executions. There are many benefits in having such a model in place, including but not limited to:

- Faster feedback loops: Since changes in the code will be easier to test and verify, it will be easier for the designers of the system to make improvements by refactoring and redesigning. This will result in higher quality and lessened complexity of the system.
- Decreased maintenance costs: Knowing which parts of a system will be affected by every change will lower maintenance costs, since it may no longer be necessary to run a full regression test suite after every change.
- Faster handling of trouble reports: The model will also make it possible to isolate and fix faults already existing in the system more quickly and efficiently. Naturally, in this scenario developers need to find the bug first, but after introducing a solution they can get quicker feedback by using our model.
- Faster learning and higher productivity: Being able to directly connect code changes with potential points of failure will also help to identify dependencies in the system as well as accelerate the learning of the system, possibly resulting in a shorter introduction time for new designers in the company and increasing productivity.

1.3 Research Steps and Methodology

This section provides an outline of the tasks and steps of the research conducted during our study. The steps and the methodology are based on common machine learning pipelines. To this end, we first perform a literature study by going through related work. This is to familiarize ourselves with some of the relevant research that has been conducted in software quality verification using machine learning. This covered the first two stages/steps of our study, see Figure 1.1. Based on our study, and as shown in Figure 1.1, the remaining steps are: (3) Data collection, (4) Feature extraction, (5) Design of the prediction model. As the last stage we study the results.

In this thesis we have taken a mixed qualitative and quantitative research approach. That is to say, for the data collection, feature extraction and predictive model design, we not only have taken on the experience of the quantitative approaches presented in the literature but also we have conducted interviews with the experts in the domain at the company which cover the qualitative portion of our study. We present our data collection

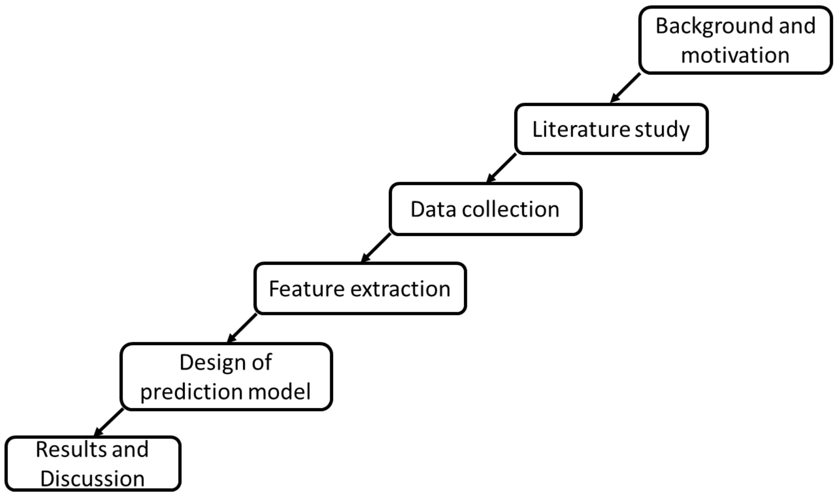

Figure 1.1: Thesis research flow

process by first identifying and extracting relevant data, and parsing it into an acceptable format. This part is explained further in Chapter 2. The chosen features are selected based on both literature review as well as input from discussions with supervisors. This part is heavily influenced by prior work, most notably the research conducted by Feldt's research at Ericsson [30] as well as Gotlieb [31]. Afterwards we design and train a predictor that provides us with an estimate of the probability of failure of test cases based on the newly built software. Having conducted the study in the aforementioned stages, in the last chapter we conclude the thesis and discuss the future work.

1.4 Thesis disposition

In this chapter we provided an introduction to this thesis. In Chapter 2 we provide a background for this thesis and present the related work. Then in Chapter 3 we provide a detailed description of the steps conducted in this research for finding, collecting and parsing data, as well as the motivation behind the type of collected data. Chapter 4 presents the feature extraction procedure, where we discuss how the chosen features are calculated and implemented from the collected data, and we give a motivation for grouping the features in two different categories: test and source code features. We also suggest a weight function which will be described in detail in Section 3.3. Having described the features and how they can be computed, in Chapter 5 we describe the models selected for this study. We also present the machine learning algorithms used in this research and the motivation behind selecting these algorithms. We then present the results and analysis of the conducted research regarding model selection in Chapter 6. Finally we finish by discussing the conclusions of the thesis work and suggestions for future research in Chapter 7.

2

Background

Nowadays in many industries large amounts of data is being generated on a daily basis. The good news is that processing and analysing such data is becoming more and more feasible by the advent of new computing technologies. One of the best sciences for automating data analysis is machine learning. Machine learning techniques are not new, but in the recent years they are gaining fresh momentum.

In the context of this thesis work, the motivation for using machine learning methods is that we are dealing with very large amounts of test and code data, with complex relations. Despite having access to such amounts of data, not all of the data is usable and in fact much of it proven to be faulty and unusable. To make more efficient use of development time, instead of spending time analysing data and drawing conclusions manually, we want to leverage the power of machines to find existing relations. However, there are tradeoffs since the results might be opaque, i.e. it might not always be clear why machine learning models predict that a certain test should be executed. This thesis will also address and discuss these tradeoffs.

2.1 Testing and different testing methods

In software products where there are many developers (several hundreds) working on the same code base, it becomes crucial to integrate each change in such a way that the base line as well as the legacy code stay in a working condition. Booch introduced a very good concept to resolve such situations [32], called Continuous Integration (CI). This is very popular in so-called Agile ways of working [33]. The product in our case study follows Agile ways of working as well as CI for quality assurance purposes.

The goal of CI is to keep the software in a working state at all times. To meet this goal, every time a developer commits a change, the entire application is rebuilt and a set of automated test cases - such as block, function and network tests - are then executed.

CI will then send feedback to the development teams if their change has broken the main branch and it is the responsibility of the team to fix the main track as fast as possible [34].

From different test results we have decided to pick Function Test cases (FT) due to the scope of this thesis as well as the importance of such tests. In software test development, FTs are black box test cases. These test cases only examine the functionality of the test subject and do not concern themselves with the inner workings and internal structure of the subject.

2.2 Related Work

The process of delivering software from the workspace of software engineers to the end users is called software release engineering. Continuous integration and regression/function tests are a part of this process and are gaining new research momentum due to a new way of releasing software called continuous delivery. Continuous delivery makes it possible to deliver contents to end users in a matter of days instead of the previously popular methods which took months or even years. Despite this, one must be wary that quality is still a very important factor and therefore needs to be thoroughly researched given the shortened periods of deliveries [35].

Moreover, in any growing software product with a large number of test cases and legacy tests, the problem of identifying and executing relevant test cases to adequately cover new changes in code is a difficult, costly and challenging task. For instance, after studying 61 different systems, A. Labuschagne and L. Inozemtseva found out that it is expensive to diagnose software defects. There are many flaky, incorrect, or obsolete test cases in a studied CI machinery and nearly 3 million lines of test code, and yet over 99 percent of test case executions could have been eliminated with a perfect oracle [36]. Having said that, and despite the fact that in recent years due to faster pace deliveries there is an even bigger demand for test automation, it has been shown that automated test cases can themselves be buggy and can potentially produce extra maintenance costs. Therefore, it is not always cost effective to have more automation [37].

There are different approaches to testing in the industry. Some companies claim that the best way is test automation, others prefer to do testing manually and some others claim that testing is too expensive and should therefore be eliminated. A. Keus and A. Dyck compare these approaches and suggest that testing is really important but also acknowledge that this largely depends on the product under test [38]. In our case study and the company under study, due to very high service level agreements, testing is of great importance for the company and the products considered in the thesis.

To address the time consuming and expensive testing procedures within test automation, there have been numerous attempts to detect defective code and predict software failure based on historical data of code and test execution, as well as code churn - the lines of code added, modified or deleted from one version of a file to another [39]. Some of the proposed predictive methods utilize statistical analysis of process and product metrics, such as frequency of code changes, delta lines (the number of lines of code changed between two versions) or block and module coverage to estimate failures [40]. Also, recent advances in both software and hardware technology have made it possible to use machine learning techniques for this purpose. There have been publications in this field by Noorian et al. [41], Briand [42, 43, 44], Giger [27], Wikstrand et al. [45] and many more.

The goal of our research is to design a model which, based on the changes made to the code, will suggest test cases that are most likely to isolate possible faults in the code. Based on the conducted literature review, we can classify existing relevant approaches into three main categories. We will expand and elaborate more on the first two categories:

- Defect prediction methods that aim to localize defective code based on historical data of code and test executions, e.g., Giger et al. [27], Brun and Ernst [46], Briand [42] and Wikstrand et al. [45]. Some of the methods in this category aim to isolate failing or defective modules based on trouble reports. Such methods rely on supervised and unsupervised classifiers and use the resulting classification to assess the frequency and severity of failures caused by particular defects and to help diagnose these defects. Nagappan [40] and Podgurski [47] have conducted extensive research in this area.

- Regression test selection, optimisation and reduction [28, 29, 48, 49, 50, 51, 52]

- Methods that not only automate the testing but go deeper by automating test generation using machine learning [41], [43] and [53].

In the first category, Briand et al. [42] and Brun and Ernst [46] have the same approach but target different parts of software. Briand [42] studies test suites and proposes an automated methodology based on machine learning, making the analysis of test weaknesses easier for developers to continuously refactor and improve. Giger [27] catches parts which are likely to contain errors from the source code. This is cost efficient since the authors claim that generating new test cases is expensive - even if test cases could be generated, it would be expensive to compute and verify a model to represent the desired behaviour of the program. Briand [44] identifies suspicious statements during software debugging. That is information is gathered from failing test cases and then test cases executed under similar conditions are assumed to fail due to the same fault. Mockus and Weiss [54] predict the risk of new changes based on historical data collected from the changed files, modules and subsystems (changes include fault fixes or new code). Predictive models are used to build a framework to predict possible faults. Khoshgoftaar [39] detects fault-prone modules in a very complex telecom software system based on the code churn - the number of lines changed or added due to a bug fix - by using historical data between different software releases.

One has to bare in mind that defect prediction models may be unreliable if the trained data is noisy [55]. Recall that prediction models are designed by mining historical data saved by version control systems (VCS). VCS save software changes on a file level. In contrast with VCS, many existing software metrics are designed using class or method-level granularity, and therefore make defect prediction models harder to design. Although there are ways to overcome this problem, such as aggregating software metrics on a file level, this is still not so effective.[56]. Although, aggregation of metrics may help, but one must be vary of redundant metrics generated due to the aggregation. Authors in [57] suggest that researchers must be aware of such threats before designing their prediction models.

Other methods predict failing or defective modules based on trouble reports, which basically consist of plain text. Nowadays there are different ways to report software failures or bugs, which can originate from testers, customer units, or to be reported online directly by the end users, e.g., the bugs reported from the end users in the Windows operating system [40]. Such bugs could be very helpful for designers, but due to their vast numbers it could take days or even months to go through them just to categorize, prioritize, remove duplicates, and filter out useful information in order to fix the bugs. Machine learning provides tools to handle such situations, where the amount of data is too large to process manually. Nagappan [40] states that failure predictions are very useful and possible even in very large and complex software products, such as the Windows operating system. This study creates statistical models to predict post-release failures/failure-proneness in Windows OS. Podgurski in [47] classifies software failure reports by the severity of failures caused by particular defects, then uses this classification to diagnose and prioritise these defects.

Second category focuses on optimization of regression/function test execution, selection and prioritization. In recent years there has been many number of studies done in this area such as the work done by K. Ricken and A. Dyck in which they claim that research efforts has been mainly focusing on reducing the time it takes to execute all the test cases, and only in recent years more focus has been placed on other aspects such as relevant test case selection. The latter types of optimization can be grouped into two categories, namely value- and cost-based objectives. Usually, we would like to minimize the cost, for instance, reduce the cost of a setup needed for a running a test case, and maximize the values or benefits, such as maximizing code-based coverage [58]. D. Card and D. S. Lee, Z suggest that running all regression tests each time a change occurs in the software product is an expensive and time consuming process, thus the authors suggest not running all the test cases but only the parts which have been affected by software changes. Authors claim that there are no techniques which are fully adopted in practice, and discuss one lightweight RTS library called Ekstazi which is getting more attraction amongst practitioners. Ekstazi tracks dynamic dependencies of tests on files, and unlike most prior RTS techniques, Ekstazi requires no integration with version-control systems. In addition to Ekstazi, the authors claim that using source code revision history data such from sources such as git they can increase the precision of Ekstazi and reduce the number of selected test cases [29]. J. R. Anderson claims test selection, reduction and optimization lowers the cost of quality assurance. This research showed that attributes such as complexity and historical failures were the most effective metrics due to a high occurrence of random test failures in the product under study [48].

A. Shi and T. Yung in [49] claim that running regression test cases in general can become expensive, but since regression tests ensure that existing functionality does not break while making changes to the code base, they are still important and it is important to come up with a good model to reduce such costs. To this end, they suggest test-suite reduction and test selection as two main approaches. They also note that previous research did not compare such approaches empirically. They claim test-suite reduction can have a high loss on fault-detection when there are new changes to software. N. Dini and A. Sullivan in [28] found out that such techniques reduce the cost of regression testing by only executing test cases relevant to the modifications in the code. This research investigates regression test selection based on the granularity of code changes, namely file and method/class level, as well as the way test cases were generated, i.e. automatically or manually. The study suggests that regression test selection based on changes in the files is better when tests are generated manually, while, on the other hand, selection based on method level changes is more efficient when test are generated automatically.

C. Plewnia in his research claims that regression test automation improves efficiency and shortens software release cycles. But as software grows, the number of test cases grows as well, therefore executing all tests in time is impossible. Hence, further optimization of the regression tests' efficiency is needed [50]. R. Saha and M. Gligoric in their study use binary search for test selection. They commit the selection and save the number of compiler invocations and the number of executed test cases, therefore saving overall debugging time. According to Linux kernel developers, 80 percent of the release cycle time is dedicated to fixing regression bugs [51]. Among other work in this category, the work in [52] helps reduce the number of wasteful test executions by concentrating on test selection on module level. Authors claim that there are wasteful test executions due to many module interdependencies. Therefore, additional dependencies in the module will cause the execution of test cases that are not actually related to the main purpose of the test case. To reduce test executions, the authors claim there should be better placement of test cases, and they implemented a greedy algorithm which suggests test movements while considering historical build information and actual dependencies of tests. The same idea and research can be extended to system test selection, reduction and optimization. Here the main concern is that one does not have access to source code and it's history. This is because while executing system tests one does not have access to source code, and as a result historical data from test logs is used for test case selection using genetic algorithms [59]. Similarly in [60] it is acknowledged that system testing is an important task in software quality assurance where testers have no access to the source code. The authors use supervised machine learning techniques to prioritize system tests by using test case history as well as natural language test case descriptions. Their result significantly improves failure rate detection.

Research by Harder [53] falls into the third category - by automatically generating test suites one can have complete test coverage using operational specifications. Noorian [41] and Briand [43] both suggest that machine learning has the capacity and potential for automating the software testing process as well as the ability to solve many of the long-standing problems in the area. Noorian [41] proposes a general classification framework for test automation by using machine learning, and Briand [43] describes several existing applications that are currently using such techniques while also emphasizing the need for further research.

During the conducted literature study and at the time of writing this thesis, we found that it is difficult to isolate a universal state-of-the-art algorithm for automatically economizing the testing procedure. This is because the proposed algorithms' performances are heavily reliant on the assumptions made concerning how the changes to the code modules are logged and how the designers set out to localize pieces of problematic code. The discussed literature above, to our knowledge, represent good works conducted on the categories discussed above.

Our work is mainly related to second categories. During the conducted literature review, we have found similarities in other work such as Mockus and Weiss [54], Brun and Ernst [46]as well as [28, 49, 50, 51, 52, 59, 60], where predictions are made based on historical data collected from changed files, modules and subsystems. We will use similar data as well as changes made in test cases. However, instead of trying to build a framework to predict faults in the source code, we aim to shorten the quality assurance feedback loop by relating code changes to potentially relevant test cases. Since the test cases and code base under our consideration span several years of data, it is also interesting to consider the general behaviour of the test cases over time.

The case study conducted by Feldt et al. [30] analyses the efficiency of test cases based on their age. Since the ways of working of the system in Feldt's research [30], as well as the data under consideration, are similar to the ones in our research, we can use the results to influence our selection of features and analysis of final results. This gives ground for using machine learning methods in our research to implement a model for predicting the relations between code churn and relevant test cases.

2.3 Brief product description

IMS is a core network solution built on 3rd Generation Partnership Project (3GPP) standards, enabling real-time consumer and enterprise communication services over any access technology. The SBG source repository history and test history will be used as training data for the model whose planned functionality will be explained in the rest of the thesis.

SBG is a large system with many components with several millions lines of code and many test suites. We have access to large amounts of historic test result data in which test cases have been excluded or included based on platform-related problems and test strategy. The structure of the system is in a constant change - some parts are more rigid, and other parts are evolving with spurs of mutations. There are over 220 developers changing the code on a daily basis.

Moreover, the code is frequently redesigned and refactored, thus further complicating matters by moving around the internal correlations of the system. A piece of code which triggers a failure in a test with a certain probability will be moved to another part of the system or even dispersed over a set of modules when deemed necessary. In our case it is possible to observe code modifications with a very high granularity.

3 Research methodology and data collection

An ideal research methodology to achieve the main goals of this thesis would be an iterative one, where having achieved a preliminary understanding of the problem and the available data, we would in turn iteratively improve our awareness of the problems and improve the design. However, such research methodologies, namely action research and design science research methodologies, could not be effectively implemented in our setting. The reason behind this is as follows. Notice that the design of an effective solution for the considered in this thesis mainly concerns (*i*) a methodology for gathering relevant, useful and high quality data, and (*ii*) design of machine learning algorithms using this data. So ideally one would want to employ a design science research methodology for improving both of these steps. This would constitute two nested iterative loops, where the outer loop would concern the design approaches for acquiring relevant data and the inner loop would concern the design and refinement of the machine learning algorithm.

Unfortunately due to the fact that the data was provided to us in advance and we did not have the possibility nor the time to propose changes and refinements to the data gathering process, we instead adopted a case study methodology towards this work. That is to say, we conducted a thorough study of the provided data and its collection process. Then having identified the issues and limitations of the available data, we set out to design a machine learning algorithm for test case selection automation. Through this process our hope has been to firstly, provide guidelines on how to improve the quality of the data for our goal and secondly, to refine the design of the machine learning algorithm so as to reduce the effects of the issues present in the data. To this end we here present some information regarding the data, its collection and preprocessing approaches.

The data available for this thesis work consists of historical data including test results as well as source code changes collected as log files. Different portions of this data have different formats, and before we can start extracting the necessary features from it, we first need to unify its format. Furthermore, despite having access to large amounts of data, not all of the data is usable and much of it proven to be faulty and unusable. To this end, we first take an inventory in order to understand the distribution over the different types or formats. Then we parse and normalize the data into a format which can be used by the learning algorithm. This chapter provides details about this process.

In Section 3.1 we will provide an overview of different test environments and will briefly describe what kind of test cases are covered by these environments. By test cases we refer to functional tests (FTs). Section 3.2 contains a description of how the test execution data was collected and parsed. In the last section we will mention why and how we collected the data related to the source code.

3.1 Test environment and test cases

In this section we will explain the type of test cases covered in our study, as well as the differences between the two test environments used in the SBG product.

SBG has two main test environments - target and simulated. More than 3600 test cases are executed in the simulated environment during the nightly builds. The target environment is similar to the one sold to the customers. Fewer test cases are executed in this environment due to its high cost. The main aim of the testing procedure is to execute as many test cases as possible in the simulated environment, but it is not possible to cover all the tests due to environmental dependencies. For example, some test cases need to access kernel functionalities on the Linux operating system, therefore they are executed in the target environment. It is the responsibility of the software development teams to implement new test cases as well as execute existing tests in order to ensure both the new and legacy functionality of the system.

This study will mainly focus on black box tests which are collected from Continuous Integration (CI) nightly runs. Almost all the test cases included in our study are collected from the CI function tests, which are mainly implemented by development teams. In this product the main purpose of function tests is to ensure the quality of newly implemented features and to ensure that the newly introduced code behaves according to its requirements specification. Another small set of tests which is also included in CI consists of test cases implemented by the release team to verify the quality of the software which is ready to be released to the customer.

3.2 Data collection and preprocessing

In this section we will explain and show an overview of the data collection process. A preview of this process is visualised in Figure 3.1.

In the SBG product CI test cases are executed on nightly automated builds. During the test runs the latest results are constantly shown on the CI web page, and information

Table 3.1: Data on Daily Test Execusions and Results

Test Data			
Test Case Name	Execution Date	Test Case Execution Time	Test Result
(c_1)	1/1/2015	5.935509	OK
(c_2)	1/1/2015	19.278976	OK
(c_3)	1/1/2015	26.948410	OK
(c_4)	1/1/2015	108.781373	SKIPPED
(c_5)	1/1/2015	898.371059	FAILED
(c_i)	1/1/2015	898.432609	FAILED
...	...	...	...
(c_N)	1/1/2015	6.351767	FAILED

regarding the parallel test executions is saved in the log databases. As the first step of the data preprocessing, we analyse the logs and find out in which files and in what order the test cases have been stored.

After isolating the relevant logs we start the extraction and parsing the desired data out of these logs by implementing a parser in the form of bash scripts. Unfortunately, it is not possible to present the parser in this thesis due to confidentiality reasons.

An excerpt of the collected test results is shown in Table 3.1. In this Table c_i is the name of a test case executed on January 1, 2015. Its execution time took 898.432609 seconds and it failed. In order to validate our data collection and parsing approach, we select some days at random and manually compare the collected data with the results presented on the CI home page. The generated results, albeit quite similar, were not exactly the same, and they mainly differed in the total number of test cases. In order to find the reason behind these differences we conducted a set of interviews with testers in the CI team. The conclusions of these interviews indicated that the reason we had a smaller number of test cases is due to differences in test environments. The CI web page was showing all the test results executed both on simulated and target environments, while our results were collected only from the test cases executed on the target environment. After realising this we identified the logs which stored the simulated environment results and parsed those as well. In the following chapters we will discuss how to fit the results of these two environments to our selected model.

The test cases can have one of four possible outcomes - *failed, OK, skipped, or auto-skipped.* After some data analysis we realised that a high number of test cases were skipped. This was because many test cases are not executed on target environment but only in the simulated environment. These test cases are intentionally configured to be skipped on the target environment, and we have decided to disregard them. This is

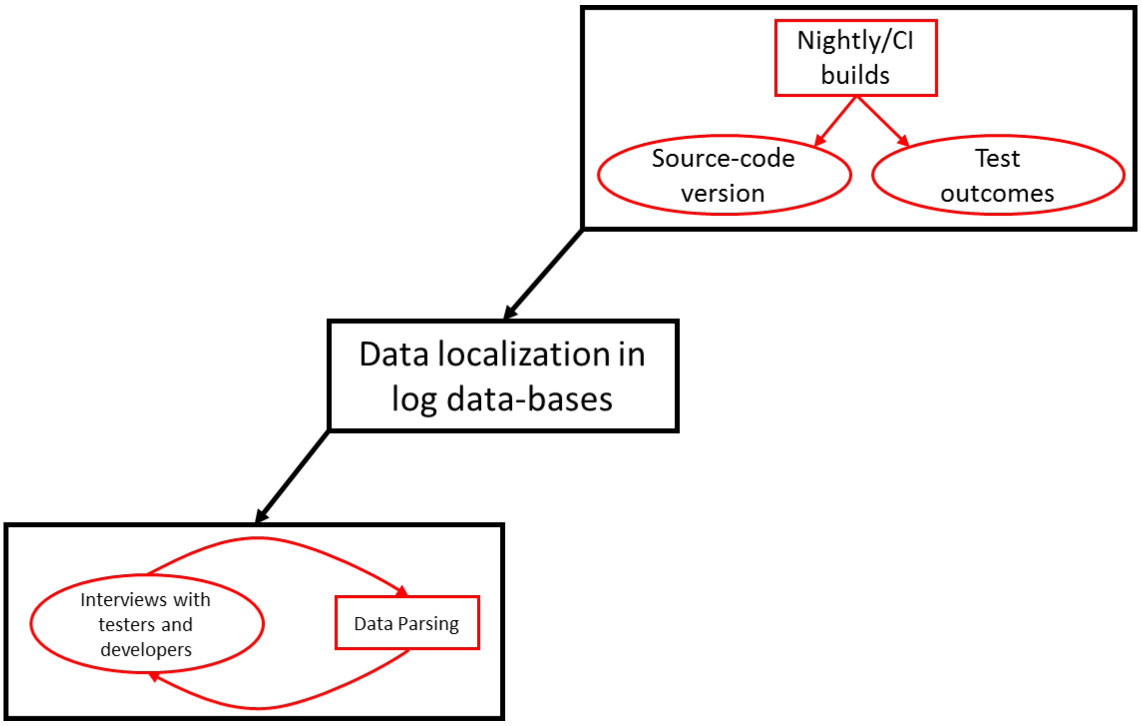
Nightly/CI builds
Source-code version
Test outcomes
Data localization in log data-bases
Interviews with testers and developers
Data Parsing

Figure 3.1: Data collection process

presented in more detail in the chapter on feature extraction.

There are some overlaps between the test cases in the two different environments, which makes it difficult to merge their results. For instance, if a test case fails on the target environment it will fail on the simulated environment as well, but if it passes on the simulated environment it might still fail on the target environment. It is very important to consider this kind of behaviour while comparing the predicted results. While verifying the collected data we faced another interesting characteristic. As mentioned before, data is collected from CI nightly builds which are exeuted every day, including weekends. We have observed instability in some of the test outcomes; for instance, there were occasions where the version of the modules had not been changed between some builds but test outcomes were different. It is not very clear how one should address this behaviour and normalise data so it can be trustworthy and accurate. In the next chapter we will explain how we used this to implement a weight function based on the unstable nature of some of the test cases. Before moving on, however, in the coming section we will explain how we inspected and parsed the appropriate information about the different versions of the software under consideration.

3.3 Source Code

Before explaining how the source data was extracted and parsed, we will explain the basic structure of the software. We will use two main concepts - modules and blocks. In the product the modules are Erlang modules. These are sets of functions grouped in a file, which essentially serve as a container for functions. Modules provide the contained functions with a common namespace and are used to organise functions in Erlang. Blocks are containers of different modules; this concept is not specific to Erlang but is a design choice in the product.

We will now explain how the data was explored, collected, and parsed. A limitation in our work was caused by our source code revision control system - ClearCase. The primary issue was that we could not know which versions of which files corresponded to a specific change. This was due to the way in which ClearCase works - one version is stored per file, and not per repository. Thus, for example, given 10 different file changes during a day, and 2-3 different versions for these, we did not know which versions belonged together.

Due to all this, our decisions are made based on the module version changes included in each automatic build. This means that we know which files/modules are changed in the current system but we are not aware of the size and importance of the change. Having access to module content would have given us the benefit of having more precise information [61]. With our current data we might face some difficulties. For example, given the change itself, it will be impossible to tell whether a module has been changed because a comment has been added or because a major change has been introduced. We are currently blind to the amount and severity of the change. In Table 3.2 an example of module version information is presented. The first column in this table represents the module names M_n, the second column shows the version of the module for a build on a specific date, which is represented in column three. The bash scripts for the parser for extracting the revision information of the relevant software for each build are not included in the thesis due to confidentiality reasons.

Table 3.2: Module revision Data on Daily build

Source Data		
Module Name	Module Version	Date
(M_1)	R7A/1	1/1/2015
(M_2)	R6A/R10A/1	1/1/2015
(M_3)	R4A/R10A/3	1/1/2015
(M_4)	R1A/R2A/R4A/1	1/1/2015
(M_5)	R1A/R6A/1	1/1/2015
...	...	1/1/2015
(M_N)	R6A/R9B/3	1/1/2015

4

Feature Extraction

The features selected for the learning procedure can be divided into two main categories, one concerning the test cases and one pertaining to the modules. The reason behind this selection is the intuitive relevance of the features when it comes to deciding on which test cases are more likely to fail, and hence need to be executed. Our feature selection is also mainly inspired by the previous work of Robert Feldt [30]. Furthermore, the proposed features are chosen so that they can be extracted from the available data.

The rest of this chapter will focus on the description of each of the features. That is, in this chapter we provide a detailed description of the features and the reasoning that motivates their selection. More specifically, we discuss the features related to test cases and those related to modules in Sections 3.1 and 3.2, respectively.

4.1 Features Related to Test Cases

Features related to test cases and test suites are produced from data extracted from CI test runs. As was mentioned earlier, test suites are sets of test cases which verify the quality of a certain behaviour of the software. Test suites categorise test cases for verifying even more specific aspects of the software. This commonly helps speed up the test process by setting up the environment for each group and performing cleanup after the execution of said group. In general, all test suites share a set of sub-functions which are not related to the actual test cases but are only used for setting up the test environment and performing cleanup procedures after the tests are completed. Based on this information, we have extracted features for describing test cases, which will be explained in detail below.

Let us assume that we have N test suites $S = \{S_1, S_2, \dots, S_N\}$ and that in each test suite S_j we have n_j test cases $C^j = \{C_1^j, C_2^j, \dots, C_{n_j}^j\}$. For a test case C_i^j we denote its outcome at time t by $O_i^j(t)$. Using this notation, we define two main features related to a particular test case, namely the failure rate of the test case and its execution rate.

1. **Failure rate feature**. The failure rate of a test case corresponds to the number of times the test has failed in a specific period of time with length of T_w. Specifically the failure rate of each test case C_i^j at a date t_c, that is $F_i^j(t_c)$, can be written as

$$F_i^j(t_c) = \frac{\sum_{t=1}^{T_w} \mathbf{1}(O_i^j(t_c - t) == \text{fail})}{\sum_{t=1}^{T_w} \mathbf{1}(O_i^j(t_c - t) \sim= \text{skipped})} \tag{4.1}$$

   Here $\mathbf{1}$ defines a function that returns 1 if the statement in its argument is correct. For instance, $\mathbf{1}(O_i^j(t_c) == \text{fail})$ returns a one if the test case C_i^j has failed at time t_c and zero otherwise. The collection of these quantities results in the feature vector $F = (F^1, \dots, F^N)$ that summarises the failure rate of all the test cases in all the suites.

2. **Execution rate feature**. This feature corresponds to the number of times a particular test case C_i^j has been executed over a time window T_w,

$$E_i^j(t_c) = \frac{\sum_{t=1}^{T_w} \mathbf{1}(O_i^j(t_c - t) \sim= \text{skipped})}{T_w} \tag{4.2}$$

   The collection of these quantities results in the feature vector $E = (E^1, \dots, E^N)$ that summarises execution rate for the test cases.

Along with these two features, we also consider a feature pertaining to each test suite.

1. **Suite failure rate feature**. For each test suite j we define the suite failure rate feature at time t_c, denoted by SF_j, as the summation of the number of test cases which failed in a particular test suite in a specified time window T_w. The description for this feature can be written as

$$SF_j(t_c) = \frac{\sum_{t=1}^{T_w} \sum_{i=1}^{n_j} \mathbf{1}(O_i^j(t_c - t) == \text{fail})}{\sum_{t=1}^{T_w} \sum_{i=1}^{n_j} \mathbf{1}(O_i^j(t_c - t) \sim= \text{skipped})} \tag{4.3}$$

   The collection of these quantities results in the feature vector $SF = (SF_1, \dots, SF_N)$ which summarises failure rate for the test suites.

4.2 Features Related to Modules

As mentioned earlier, the product under consideration is built up of different blocks, and each block is responsible for delivering certain functionality of the system. The blocks are composed of a number of modules, which are Erlang files grouping different functions. Before describing the features related to modules, we would like to point out that because of the versioning system used in the company, the product is implemented in different development branches. Each branch has a unique name and delivers a certain type of functionality. There is also a main branch which all the other branches merge to. The code used in our study is taken from the main branch, which means that there are changes on secondary branches which have not been included in our study. The reason for this is because the code on the main branch is more stable, and considering more complicated scenarios is outside the scope of our study.

Now let us assume that we have M blocks and that each block k contains m_k modules. For each module k in the block b, let us denote its version at time t with $V_k^b(t)$. Given a time window with T_w days, we have the following set of features related to modules.

1. **Module change indicator feature**. This feature marks the modules which have been modified between the current day t_c and the previous day $t_c - 1$. This is done using a binary function that returns 1 in case the module has been changed between these days and returns 0 otherwise, as in:

$$Z_k^b(t_c) = \mathbf{1}(V_k^b(t_c) \sim= V_k^b(t_c - 1)). \tag{4.4}$$

   The collection of these quantities results in a binary feature vector $Z = (Z^1, \ldots, Z^M)$ that summarizes which modules have been changed.

2. **Module rate of change feature**. The module rate of change corresponds to the number of times a module has been changed within a given period of time. For each module, this quantity can be computed by adding the number of times the module has been changed between two consecutive days and dividing the result by the $T_w - 1$, which is a normalising constant. Hence, this feature can be written as

$$R_k^b(t_c) = \frac{\sum_{t=1}^{T_w - 1} Z_k^b(t_c - t)}{T_w - 1}. \tag{4.5}$$

   The collection of these quantities results in a feature vector $R = (R^1, \ldots, R^M)$ that summarises the rate of change in modules.

Similar to modules, we can define features concerning block of modules. We next present these features.

1. **Block change indicator feature**. This feature concerns the relative number of modules that has been modified between two consecutive days. In order to compute this feature, we compute the number of modified modules in each block and normalise the result by the number of modules in the block. This can be represented mathematically as

$$B^l(t_c) = \frac{\sum_{k=1}^{m_l} Z_k^l(t_c)}{m_l}. \tag{4.6}$$

   The collection of these quantities results in the feature vector $B = (B^1, \ldots, B^M)$ that summarises how many modules have been changed within different blocks.

2. **Block rate of change feature**. The block rate of change concerns the collective amount of module changes for a block within a given period of time. This feature can be computed as

$$P^l(t_f) = \frac{\sum_{t=1}^{T_w-1} B^l(t_f - t)}{T_w - 1}. \tag{4.7}$$

   The collection of these quantities results in a feature vector $P = (P^1, \ldots, P^M)$ that summarises rate of change in modules in blocks.

This concludes the description of features to be used in our predictor. It goes without saying that the right choice of features greatly improves the performance of the designed predictor. However, there are other possibilities for enhancing the devised predictor. Next we touch upon two suggestions for this purpose.

4.3 Incorporating the Stability of Test Cases

The performance of the predictor depends heavily on the quality of the provided data. The collected data in this thesis, basically provides us with the history of the test outcomes, which enables us to devise predictors for each of the tests. In this context in case the test outcomes are uncertain or in other words if a test is unstable, the resulting predictor will also suffer from the same shortcomings. The adverse effects of such test cases has also been considered in [62]. In this paper in order to alleviate this issue Kim Herzig develops a model which detects the false test alarms by focusing mainly on integration test cases. According to his arguments false test alarms are caused by test cases which fail due to test cases being faulty themselves, investigating on such test cases are unnecessary since they wont give any valuable information but they slow down the whole development process. Herzog claims that such test cases therefore should be detected and eliminated. In our work we also suffer from similar issues. In order to isolate problematic test cases, and based on the available data, we first need to find a measure that quantifies our trust in the collected data. This will then allow us to reduce the effect of unstable tests or uncertain data within the learning procedure.

One of the ways for quantifying the tests instability is through monitoring its outcome within given periods of time and detecting the number of times an outcome of a test has changed when none of the modules have been changed. To this end, we can define the following indicator

$$Q_i^j(t_c) = \frac{\sum_{t=1}^{T_w-1} \mathbf{1}(O_i^j(t_c - t) \sim= O_i^j(t_c - t - 1) \wedge \mathrm{sum}(Z(t_c - t)) == 0)}{\sum_{t=1}^{T_w-1} \mathbf{1}(\mathrm{sum}(Z(t_c - t)) == 0)}, \tag{4.8}$$

where recall that the vector Z summarizes the module change indicator for all modules. This indicator quantifies the level of instability of a test case C_i^j. The closer this value is to 1 the more unstable the corresponding test case. So if this quantity is equal to 1 the corresponding test case has been completely unstable within this time window T_w, and if it is equal to 0 the test case has been completely stable. We can use this quantity as a weight in the cost function within the training of our prediction model. That is, if for a particular test case this quantity is equal to 1 we put a zero weight for the term pertaining to this test case for that particular time point. By doing so we avoid colluding the training procedure with unreliable data. Similarly, for a test case where this quantity is equal to 0 we put 1 as the weight for the term pertaining to this test case for that particular time point. In general, we consider the weight $1 - Q_i^j(t_c)$ for a term in the cost function that relates to test case C_i^j at time t_c. This will reflect our trust in different data points, and will potentially result in better trained predictors.

4.4 Incorporation of Ignored Tests

Within the course of testing through several days, it can occur that a test case is skipped several days in a row. It seems logical that if a test case is ignored for a certain period of time, it should be executed even if it is not picked for execution, or even if the prediction model predicts a low probability for failure of this test. We hence suggest to add such a watchdog to our recommender system to avoid neglecting test cases for a long time [63]. This means that we will not use this feature within the training procedure of our predictor and instead we will have it in the recommender system that decides on whether a test should be run or not.

So far we have put forth a description of considered features. We next discuss some practical issues with the feature extraction procedure.

4.5 Practical Issues with Feature Extraction

The features proposed in sections 4.1 and 4.2 are extracted from the provided data. Certain issues with the available data can hinder our ability for extracting the necessary features. In this section we describe two such issues with the data and propose simple fixes that enables us to conduct the extraction procedure.

4.5.1 Varying tests and modules lists

The product considered in this thesis is a dynamic product which undergoes constant development. This means that during the period when the data is being collected, the number of tests and modules in the product is undergoing a continuous change. Particularly, modules can be added or removed and tests can be added. This entails using predictors that can handle varying feature length during both their training and prediction phases. The existence of such predictors has not been investigated for this thesis. As we will see in the next chapter, the considered predictor in this thesis requires the feature length to be constant and cannot handle the mentioned variations. In order to fix this, we instead consider the following remedy: given a data set we first extract a comprehensive list of test cases and a comprehensive list of modules. These lists include all the tests and modules that have at some point been present in the data, and they were generated by going through the data day-by-day. Thus if on a given day there are tests or modules in the data that are not present in the comprehensive lists, they are added to the lists.

Recall that tests can only be added. Let us assume that at a point in time a test case in the comprehensive test list is not present in the data. We then modify the data by adding the test with *skipped* status. So for every test case at every time point we append the data with missing test cases, as described above. Notice that by doing so we merely assure that the number of features corresponding to the test cases remains constant, and the skipped status of the added tests assures minimal effect of the modifications on the resulting predictor. For the modules we face a slightly more complicated scenario, as modules can be both added and removed. Let us assume that at a given point in time a module in the comprehensive list does not appear in the data. We then modify the data by adding the module with a dummy version of the data. In case a module in the data is removed between two consecutive data points, this module is again added to the data with its latest available version. With the outlined strategy, the features extracted from the modified data will always have a constant length and can be used for training our predictor.

4.5.2 Missing data points

While studying the data we realised that the data for certain days was missing. This creates discontinuity in the data and hence can jeopardise the integrity of the extracted features. In order to address this issue, we again consider the cases of missing data for test cases and modules separately. Particularly, for missing test outcomes we simply add a data point with all the tests present but with skipped status. This modification seems logical as in the majority of the cases the absence of test data for a specific day is due to the fact that no tests have been executed during the day. Similarly, if at some point in time module data is missing, we simply add a new data point with all the modules with their most recent versions. This also seems logical as the modification implies that no modules have been changed during the added day. Also, in the majority of cases (if not all) the proposed modification matches reality, since usually the missing data points correspond to weekends or holidays when no changes are introduced to the modules.

4.6 Motivation behind implementing the features

As was mentioned, our choice of features has been inspired by the work conducted in [30], but it also follows the intuitive process behind the design of tests for evaluating the functionality of the modules. In order to shed more light on the motivation of our features choice, we next present this process, which was obtained by interviewing experts within the field.

4.6.1 How the test selection is done currently

The product under study is huge and complex with more than a million lines of code. It has been in development for almost 10 years, therefore there exist much legacy code as well as tests. It is not very simple to introduce a new change and write a couple of unit tests with the hope that if they pass the changed product is ready to be delivered. The developers should always consider backwards compatibility and make sure the legacy code works. For these reasons, the developers need to be aware of the impact of their new changes on the system and verify its quality.

Now let us explain how the testing is performed by developers and see how this flow can be related to our features. Firstly, developers introduce a new change within some of the modules in the product. Then they implement relevant unit tests to check the changed code for bugs and possible incompatibilities. After the unit tests have passed, they need to test the functionality. Developers select or implement function test cases based on the following certain criteria. We should have in mind that the developers know which block is changed and which modules are affected within that block. So they commonly start by finding the relevant test suites and isolating which test cases might be directly affected by the change. Sometimes it is hard to guess the particular test cases therefore developers need to execute the test suite in its entirety and analyse the failing tests based on experience. If they do not have sufficient experience to do so, they consult with those who are more experienced in that area. This is already a strong indication that features which isolate the modules that have been changed and quantify the amount of change they have undergone have considerable role in deciding which tests need to be run. Sometimes, in order to isolate the issue within the code, the testing experts study the history of relevant test outcomes in order to decide the order in which new or existing tests need to be executed in. This in turn shows the relevance of features that quantify and summarise the outcome and execution rate of the tests.

5

Learning and Classification

The aim of this chapter is to lay out our approach for devising a predictor, $h_\theta(x)$, for relevant test selection. Here x denotes the features. The designed and trained predictor $h_{\theta^*}(x)$ is then used for selecting nightly tests. In this chapter we will discuss the considered prediction models, their implementation, and the motivation behind choosing such predictors. Particularly, we will explain learning using logistic regression and random forests, and the reasons behind selecting these supervised learning algorithms. We will end this chapter by explaining how the predictor is evaluated using loss matrix and cross validation techniques.

5.1 Machine Learning Algorithms

There are several different approaches in machine learning; these are classified into three different major categories depending on the nature of data: namely, supervised, unsupervised, and reinforcement learning [64]. In supervised learning, we have a set of inputs and desired outputs which is called the training data. A supervised learning algorithm analyses the training data and produces an inferred function, which can then be used for mapping new examples [65]. Unsupervised learning is best defined by referring to its name - in this class there is no training label, the algorithm is left unsupervised, and it should try to find structure based on the given data [66]. In reinforcement learning, the computer program interacts with the dynamic environment and tries to learn from it. An example of this could be Google news or different types of game theory where there is no teacher and the program is left on its own. In this model, the agent learns from trial-and-error [67, 68].

In this thesis we intend to investigate the possibility of devising predictors that can provide us with a meaningful measure to decide on the relevancy of a test. (Relevant tests are the ones which are most likely to fail, given the identified features.) The data available for this purpose is labeled - that is, we know the outcome of each test or we

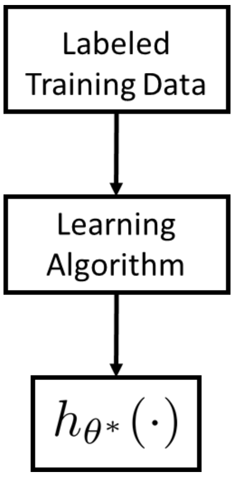

Figure 5.1: Learning

know the target values. Consequently, it becomes convenient to investigate supervised learning algorithms. These types of learning algorithms, given labeled training data, enable us to devise predictors for our underlying task, as shown in Figure 5.1. Having trained a predictor, the setup for using the predictor consists of a *feature extraction unit* which is connected to the database that contains historical data regarding module modifications and test results. Given the module version of the current day and a time window length, this unit generates the necessary features used by the predictor, as in Figure 5.2, for details on feature extraction, see Chapter 4. In a sense, the combination of the predictor, $h_{\theta^*}(\cdot)$ and the feature extraction unit, with the time window length as an internal parameter, can be seen as an oracle that provides us with the most relevant tests for the nightly test run. The inputs to this oracle are the current version of the modules, historical data regarding modules, and the test outcomes.

Notice that the output takes on a discrete number of values. This means that we are facing a classification problem. In fact, since the output values are limited to 0 and 1 - indicating whether a test has passed or failed, respectively - we are dealing with a binary classification problem. For this reason we limit our investigation to learning algorithms tailored for such problems; namely, we focus on logistic regression and random forest algorithms. The reason behind the selection is twofold. Firstly, these algorithms result in predictors which allow us to do binary classification. Secondly, the resulting predictors also provide a measure proportional to the probability of belonging to each of the two classes.

Some of the most common algorithms in the area of supervised learning are: Support Vector Machines (SVMs), neural networks, logistic regression, naive Bayes, random forests, and decision trees. We will briefly introduce decision trees, random forest and logistic regression and compare these algorithms based on the empirical study carried out by Rich Caura et al. [69, 70, 71], [72], [73].

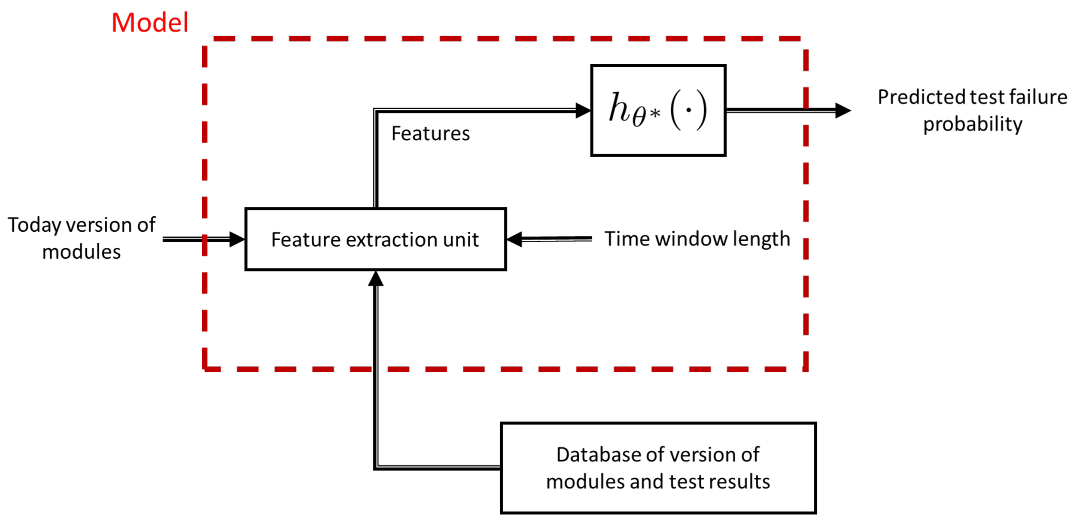

Figure 5.2: Prediction setup architecture

Decision trees (DTs) are extremely robust - they have been used for different types of problems for decades. It is very straightforward to analyse and understand the results of using DTs. They are graphical, and therefore are also appropriate for visualising and understanding the data. These characteristics make DTs very appealing to use in our work. However, it is important to keep in mind that decision trees are prone to overfitting and are somewhat complicated when it comes to data sets with many features. In such cases tuning the parameters of DTs and pruning [74] them becomes very important.

DTs support the ensemble method, that is, learning algorithms that construct a set of classifiers and then classify new data points by taking a vote of their prediction [75]. This ability makes them interesting to use particularly in cases where the data is sparse. Dietterich claims that ensembles can offer better predictions than any single classifier. Leo Breiman and Adele Cutler proposed random forest in the 2000s for building a predictor ensemble with a set of decision trees that grows many classification trees for classifying a new object from an input vector. The input vector will be introduced to each of the trees, and it will then give a classification or a vote. The forest will then choose the classification having the highest vote [76, 77].

The other modelling technique that is interesting for this research is Logistic Regression. This model has been used in many studies, since it is a powerful statistical tool for modelling binary outcomes with one or more explanatory variables. (A binary outcome takes the value 0 or 1, for example, either having or not having a certain disease.) This model is also easy to understand and apply to data [78] and it is based on modeling the log ratios of likelihoods of different outcomes using a linear function. The advantage of this model is that, logistic regression provides a quantified value for the strength of the association, which can be interpreted as the probability of the predicted outcome given a set of features [79].

Next, we briefly review logistic regression and random forest learning algorithms and discuss how these algorithms can be used within our application.

5.2 Prediction Model

5.2.1 Logistic regression

As has been discussed before, we intend to design predictors which will suggest the most relevant test cases to be executed at the end of the day. Particularly, these predictors are to assign an importance measure to each test, preferably proportional to its probability of failure, which will enable us to decide whether to run or skip it. This means that we prefer the measure to be between 0 and 1 [80]. To this end, within a logistic regression framework, the hypothesis or predictor function $h_\theta(\cdot)$ is chosen to be the logistic or sigmoid function described as

$$h_\theta(x) = \frac{1}{1 + e^{-\theta^T x}} \tag{5.1}$$

This function returns a value between 0 and 1. The task is then to compute the parameters in the logistic function, i.e. θ, so that the predictor, given features at each data point, will output a value closest to the available target value. Let us assume that for each test case j we are given N_t training data points. Then the optimal parameter value $\theta^{j,*}$ is computed by solving the following optimisation problem

$$\theta^{j,*} = \underset{\theta^j}{\operatorname{argmin}} \left\{ \sum_{i=1}^{N_t} y^j(i) \log h_{\theta^j}\left(x(i)\right) + \left(1 - y^j(i)\right) \log\left(1 - h_{\theta^j}\left(x(i)\right)\right) \right\} \tag{5.2}$$

There are different methods for solving this problem, however, we abstain from discussing any further details, as this is not the focus of this thesis. The package we have used for the algorithms is called scikit-learn, which is a library for machine Learning in Python. The reason behind choosing this package is its simplicity and reusability, as well as its open source nature (it is built on NumPy, SciPy (Scientific Python) and matplotlib [81]). We will discuss the tuned parameters and reusability of this package in the next chapter, where we present the results of the predictions. Notice that the resulting predictor outputs values between 0 and 1. By default, if this value is larger than 0.5, the prediction result will be 1 (the test is going to fail). Otherwise, the prediction result will be 0, meaning that the test is going to pass. It is rarely the case that simple thresholding would produce good classification results. We discuss the prediction process later when we describe predictor evaluation.

5.2.2 Random forests

In order for us to provide a description of random forest, we first need to discuss decision trees. Decision trees or (in our case) classification trees are models which rely on partitioning the feature space and storing a distribution over class labels for each region. This can be repeated using a tree, which in turn implies that the hypothesis function or predictor for this learning algorithm takes the form of a tree. Every leaf of this tree will then correspond to each region and is assigned with its corresponding probability distribution over the classes. Based on this description, we can use decision trees to devise predictors for each test case. Even though decision trees are easy to interpret and are intuitive, their predictions are not as accurate as for other types of predictors, such as the ones based on logistic regression. Furthermore, the tree structure is very sensitive to the provided data. This means that small changes to the data can have drastic effects on the resulting tree structure. In order to alleviate this issue, random forests are used. Such models can be viewed as a combination of multiple decision trees. To be more specific, they are devised by training different trees on different subsets of data that are randomly selected with replacement. The resulting prediction from a random forest is then produced by combining these decision trees.

We will not discuss the learning process for such methods, as it goes beyond the scope of this thesis. For training predictors based on random forests, we have used the scikit-learn Random Forest Classifier. This algorithm has many different parameters for tuning and refining the data which will be explained in the next chapter.

Similar to logistic regression, having placed a feature in a region in the feature space or in a leaf of the tree, we need to decide the predicted class by thresholding the computed probability. In the next section we discuss how this can be done, particularly when we have two classes.

5.3 Predictor Evaluation: Loss Matrix and Cross-validation

Let us assume that we are faced with a binary classification problem, i.e. we have two classes. For any predictor or classifier and given a data set, it is possible to create a table known as a confusion matrix [68, 82]:

	Predicted Negative	Predicted Positive
Negative	TN	FP
Positive	FN	TP

The diagonal entries of this matrix or table are referred to as true negative (TN) and true positive (TP), respectively, which corresponds to the number of cases that are correctly predicted as negative or positive. The off-diagonal entries - the elements in positions (0,1) and (1,0) - are referred to as false positive (FP) and false negative (FN), respectively, which correspond to the number of cases that are falsely predicted as negative or falsely predicted as positive. It is possible to evaluate the quality of devised predictors by quantifying their performance using the confusion matrix. Particularly, we consider four quantities which very well describe the performance of a predictor - the accuracy - given as

$$ACC = \frac{TP + TN}{TP + TN + FN + FP}, \tag{5.3}$$

misclassification rate given as

$$MC = \frac{FP + FN}{TP + TN + FN + FP}, \tag{5.4}$$

and false positive and false negative rates given as

$$FPR = \frac{FP}{TN + FP}, \tag{5.5}$$

$$FNR = \frac{FN}{TP + FN}, \tag{5.6}$$

respectively. It goes without saying, we prefer classifiers with high accuracy and low misclassification, false positive and false negative rates.

Given a feature vector x, the predictors presented above all return values between zero and one, which corresponds to the probability of x being assigned to a specific class. It is based on this quantity that we need to make a decision concerning the class x belongs to. One of the ways to make this decision is by computing the so-called expected loss or risk. This quantity is defined based on the loss matrix, which is defined as follows: a loss matrix quantifies the level of loss or penalty for misclassifying instances. With this definition a loss matrix can be written as

	Predicted Negative	Predicted Positive
Negative	0	FPP
Positive	FNP	0

where FPP and FNP are the false positive and false negative penalties. Let us denote this matrix by L. Notice that the diagonal entries, i.e. $L(0{,}0)$ and $L(1{,}1)$, are set to zero, as there is no loss or penalty associated with correctly classifying the instances. In such a matrix, if for example $FNP > FPP$, it means that we consider misclassifying a positive case as negative to be worse than misclassifying a negative case as positive. The loss function is defined based on this matrix. Particularly, given a feature vector x, the loss incurred by assigning x to class C_1 is given as

$$l_{C_1}(x) = L(0{,}0)p(C_1|x) + L(1{,}0)p(C_2|x) = L(1{,}0)p(C_2|x), \tag{5.7}$$

and similarly the loss incurred by assigning x to C_2 is given by

$$l_{C_2}(x) = L(0{,}1)p(C_1|x) + L(1{,}1)p(C_2|x) = L(0{,}1)p(C_1|x), \tag{5.8}$$

where C_1 and C_2 correspond to the negative and positive classes, respectively, and $P(C_1|x)$ and $P(C_2|x)$ are the probabilities of being assigned to C_1 or C_2 given x, respectively. Notice that a predictor returns a value proportional to these probabilities,

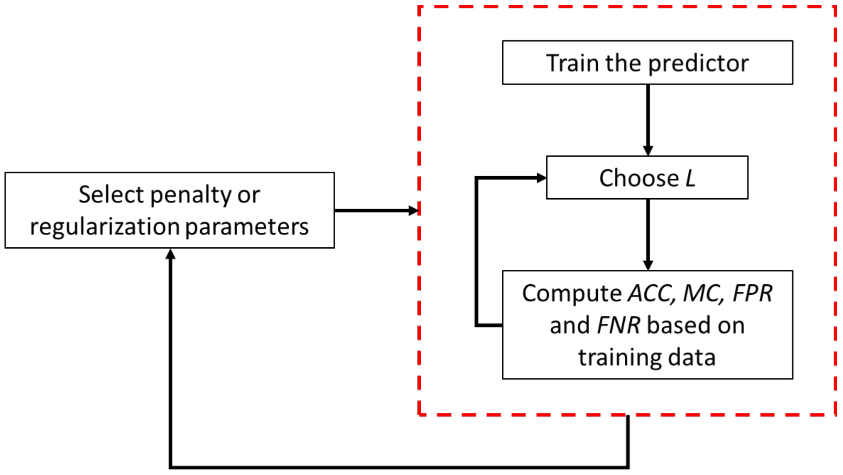
Train the predictor
Choose L
Compute ACC, MC, FPR and FNR based on training data
Select penalty or regularization parameters

Figure 5.3: Tuning the parameters using cross-validation

and hence, having computed the predictions, we can compute these loss functions for each feature vector from the data. We then classify an instant to a class that yields the minimum loss. For instance, assume that the loss matrix is given as

$$L = \begin{bmatrix} 0 & 1 \\ 100 & 0 \end{bmatrix}. \tag{5.9}$$

Then a predictor, given a feature vector x, returns a value between zero and one - say $f_p(x)$ which corresponds to a value proportional to $p(c_1|x)$. Naturally, the probability of x being assigned to C_2, i.e. $p(C_2|x)$, is proportional to $1 - f_p(x)$. Let us assume that this value is $f_p(x) = 0.55$. Then the losses incurred by assigning x to C_1 and C_2 are $l_{C_1}(x) = 45$ and $l_{C_2}(x) = 0.55$. In this case, since assignment to C_2 results in a lower loss, x is assigned to C_2. Notice that if $L(1{,}0) = 1$, then the assignment choice with the smallest loss would have been C_1, which would also correspond to simple thresholding of probabilities with the threshold value of 0.5. It goes without saying that tuning the loss matrix correctly for each of the discussed predictor types has a direct effect on the performance measures discussed above. Consequently, the tuning of this parameter is of paramount importance. The tuning of the loss matrix and other parameters within the predictor (such as regularization or penalty parameters) can be done while evaluating the generalisation performance of the predictor, that is, to see if the trained predictors behave consistently when exposed to data that has not been seen before or has not been part of their training data. One of simplest ways to do so is to use the so-called hold-out validation method. In this method, firstly the features are extracted from the provided sequential data. Then the available features and label pairs are shuffled and the predictor is commonly trained using 70% of these pairs and the remaining 30% is used for evaluating or validating the predictor. In this framework, we first tune the off-diagonal elements of L during the training phase of the learning. This means that, having trained a predictor, we tune L so as to achieve an overall good performance over the training data based on the quantities discussed above. We then expose the predictor to the validation data set, where we tune the other parameters so that the performance measures return acceptable values. This is illustrated in Figure 5.3. As we will discuss in the next chapter, we use this validation technique within our experiments and present how the right tuning of the loss matrix and regularization parameters affect the performance of the trained predictor.

6

Results and Discussion

In this chapter, we present and discuss the results of our study and implemented predictors, as well as provide a somewhat detailed presentation of our approach for training suitable predictors. Recall that we need to compute a predictor for each of the test cases. Here, we investigate and motivate general tuning guidelines and choice of important parameters in the training process of the predictors.

6.1 Important parameters and tuning guidelines

There are two main tuning parameters that have a significant effect on the performance of the predictors, namely the loss matrix and the regularization parameter for predictors designed using logistic regression and the parameter affecting the number of leaves for predictors devised using random forests. Notice that the loss matrix generally affects our decision making based on the predictor's output and, while the parameters affect the flexibility of the predictor and hence its generalizability performance. By generalizability performance we refer to the performance of the trained predictor on a data set that was not used in any way for training the predictor. Here we briefly discuss our approach for tuning these parameters.

6.1.1 Loss matrix

The importance of the loss matrix was discussed in Section 5.3. By tuning this matrix we effectively define our preference or lack of tolerance for false positive or false negative cases. Particularly, recall that if we choose $FNP > FPP$ it means that we consider a case which is wrongly classified as negative worse than a case that is wrongly classified as positive. For our case, predicting or classifying a test case as a ”0” or ”negative” case corresponds to predicting a test case outcome as ”passed.” Consequently, classifying a test case as a ”1” or ”positive” case corresponds to predicting a test case outcome as ”failed.” Notice that in this setting a false negative case can have dire consequences.

Let us assume that a test case is supposed to fail, but our prediction is that the case will pass. This means that based on our prediction, we will not run the test case, even though in reality this test case could have potentially isolated an important bug or flaw in the product. This can in turn lead to disastrous situations. Notice that the situation is not as dire for false positive cases, as they merely result in the execution of unnecessary test cases. Based on this discussion, we need to choose $FNP > FPP$. Consequently, as a general tuning guideline, in the approach presented in Figure 5.3, we tune the loss matrix with this structure. Particularly, we consider tuning the loss matrix as

$$L = \begin{bmatrix} 0 & 1 \\ FNP & 0 \end{bmatrix},$$

with increasing values of $FNP > 0$. Notice that the larger this value, the lower the number of false negative cases. However, unnecessarily large values of FNP can adversely affect the accuracy of the classifier. Hence, care must be taken to avoid this situation.

6.1.2 Parameters affecting flexibility

Recall that for each test case, test suite, module, and module block we have 2, 1, 2, and 2 features, respectively. This means that for each predictor, corresponding to each test case, we have $2 \times 4328 + 50 + 2 \times 1170 + 2 \times 34 = 11114$ features. Consequently, in the case of logistic regression, we need to train a predictor with 11114 parameters. Also in case of random forest this potentially results in predictors with many leaves. This can have dire consequences. Particularly, the resulting predictors are too flexible and over-parameterized which can lead to over-fitting, [82]. Furthermore, in comparison to the number of features, we have very few data points. In this situation, the generalizability of the resulting trained predictors can be adversely affected. This means that when we are exposed to new data different from the training data, the resulting prediction performance will be inaccurate, despite having provided impressive performance over training data.

Notice that not all the test case outcomes and not all changes to modules affect the future outcome of each test case, so many of the features can be potentially neglected. Hence we can effectively employ measures that can reduce and control the flexibility of the predictors and avoid over-fitting. In this thesis we first perform a manual feature selection. To this end, in order to predict the outcome of a test we only include test-related features that correspond to the tests from the suite that the target test belongs to. This significantly reduces the number of features, as the test-related features constitute the majority of the features in our model. This decision is motivated by the fact that the tests coming from a particular suite by design have a correlated behavior. We further improve the generalizability of the model by tuning the regularization parameter in the logistic regression package and the parameter that affects the number of leaves of the resulting predictor in the used random forest package. This is done using the approach discussed in Section 5.3. For this purpose we mainly monitor two measures: accuracy and false negative rate (for definitions, see (5.3) and (5.5), respectively). We are looking

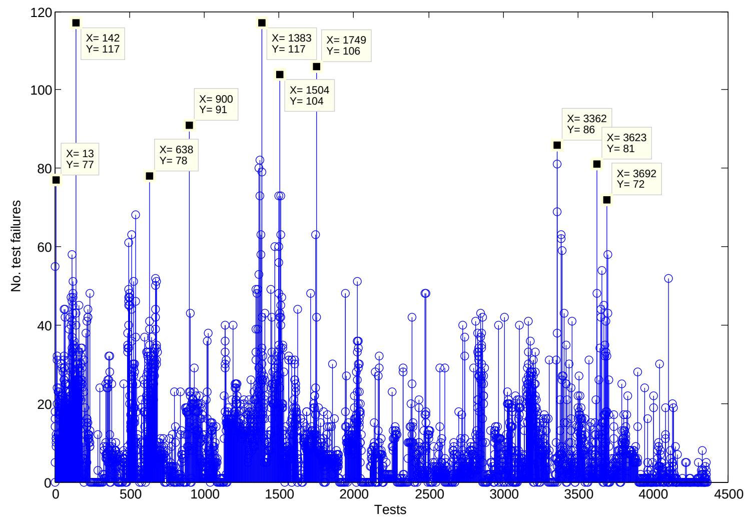

Figure 6.1: Test case selection for prediction

for predictors with as high as possible accuracy and as low as possible false negative rate - preferably zero. To achieve this within the approach discussed in Section 5.3, we tune the necessary parameters so that we obtain the least flexible or over-parameterized predictors which give the best performance over the test data through the use of cross-validation. Let us now discuss and present some of the achieved results.

6.2 Results and Discussion

In this section we use logistic regression and random forests for computing predictors which allow us to predict failures of test cases. For this study we have access to data from October 2014 to August 2015, amounting to 328 data points. In our case study we had 4328 test cases, however, in order to discuss the effectiveness of the produced predictors we decided to select only 10 of those test cases. To this end we first calculate the number of times that a particular test case has failed during the time period, which is shown in Figure 6.1. We then select the test cases that had a relatively high failure rate. The routines used for extracting the features from the available data were implemented in Matlab due to ease of fast prototyping. The time window was chosen to be $T_w = 7$ based on our experience with the performance of the predictors. These routines are reported in Appendix A. However, for the training of the predictors Scikit learn was used which turned out to be more scalable with respect to the number of features. Let us now discuss the results and the training procedure of the predictors.

6.2.1 Logistic regression

We based our first set of predictors on logistic regression. For training these predictors, we used the Logit implementation from the Scikit learn linear model library. We utilized the procedure in Figure 5.3 for training the predictors. During this procedure we tuned the regularization parameter for both choices of L_2 and L_1 regularization within the range of 1 to 0.00001. This refers to parameter C which should be a positive scalar and smaller values specify stronger regularization. We achieved consistently good results with the choice of $C = 0.1$. The following code snippets illustrate the procedure of dividing the data into training and test data sets and shows how the predictor model is trained usingthe Scikit learn toolbox.

```
def apply_logit(features_train, labels_train, features_test, labels_test):
    classifier = LogisticRegression(penalty='l2',C=0.01)
    classifier.fit(features_train, labels_train)
    prob = classifier.predict_proba(features_test)
    prediction = predict_lossmatrix(prob,loss_matrix)
    accuracy, FNR = accuracy_score(labels_test, prediction)
    return accuracy, prediction, probability

def generate_train_and_test_data():
    test_outcomes, all_features = convert_csv_to_list()
    features_train = all_features[0:250, :]
    labels_train = test_outcomes[0:250, 1382]
    features_test = all_features[251:, :]
    labels_test = test_outcomes[251:,1382]
    return features_train, labels_train, features_test, labels_test
```

Here

```
generate_train_and_test_data()
```

is responsible for dividing the available features and test labels into features_train, lables_train, features_test and labels_test, where features_train and lables_train are used for training the predictor and features_test, lables_test which is the remaining 30 percent of the data are used for checking the predictor's generalizability. To train the predictors we use

```
apply_logit(features_train, labels_train, features_test, labels_test)
```

routine. This function is the core part of our approach. It takes the training and test data and returns the predicted results as well as the accuracy and the probability of each test case belonging to a passed or failed class. The way this function is implemented is by creating the logistic regression classifier using the function

```
LogisticRegression()
```

Table 6.1: The result of the predictor based on logistic regression

Logistic regression results					
Test Case Number	Penalty	ACC	MC	FPR	FNR
1383	1	0.42	0.57	0.28	0.77
	10	0.50	0.49	0.71	0.33
	100	0.48	0.51	0.84	0.0

in the first line, and then by calling the *fit* function on the training data in order train the predictor (the arguments are features and labels). The class probabilities are then computed by calling *predict_proba* function on the classifier, which is then used to create the predictions given the loss matrix using the *predict_lossmatrix* function. The predictions are then used for computing the accuracy and false negative rates using the function *accuracy_score*.

The tuning of the loss matrix, particularly the FNP value was done to maximize the accuracy and to minimize false negative rate. The values achieved for a sample test case for different FNP choices are presented in Table 6.1. When the FNP increases, the false negative rate decreases - however, this can potentially adversely affect the accuracy of the predictor. As was mentioned we chose a loss matrix that minimizes FNR - preferably making it zero - which would give the highest accuracy. Due to this we pick $FNP = 100$ for this test case. Notice that we have considered many more values for FNP however in Table 6.1 we only present few of the tested values for FNP that clearly made a difference. Recall that for a normal test run, we need to run each test case everyday. However, using the proposed predictor we do not need to do this. In fact, the number of times we need to run this test case, based on the predictions, allows for a 10 percent reduction. This figure was extracted by studying the obtained predictions for this test case on the hold-out validation set. That is to say for the validation data for this test case, in 10 percent of the cases the test was correctly predicted to have passed and hence was correctly not selected to be run. On average, for the considered test cases, this reduction is 11.3 percent. Next, we discuss the results achieved from predictors based on random forests. It should be noted that in many cases choosing FNP such that it assures a zero false negative rate can result in predictors that require all tests to be executed. That is to say, they predict all tests to fail and hence they all need to be executed. Also, we reiterate that we chose a 70/30 training/validation data split. We also studied 60/40 and 80/20 splits, however the 70/30 split provided the best set of results for the considered test cases.

6.2.2 Random Forest

In this section, we discuss design of predictors based on random forests. The design procedure is very similar to the one discussed for logistic regression. We here also use the Scikit learn library. The routine for this procedure is given as

Table 6.2: The result of the predictor based on random forest

Random forest results					
Test Case Number	Penalty	ACC	MC	FPR	FNR
1383	1	0.41	0.58	0.0	1.0
	10	0.61	0.38	0.81	0.0
	100	0.58	0.41	1.0	0.0

```
def apply_random_forest(features_train, labels_train, features_test, labels_test):
    clf = RandomForestClassifier(n_estimators=10)
    clf = clf.fit(features_train, labels_train)
    scores = cross_val_score(clf, features_train, labels_train)
    prob = classifier.predict_proba(features_test)
    prediction = predict_lossmatrix(prob,loss_matrix)
    accuracy, FNR = accuracy_score(labels_test, prediction)
    return accuracy, prediction, prob, score


def generate_train_and_test_data():
    test_outcomes, all_features = convert_csv_to_list()
    features_train = all_features[0:250, :]
    labels_train = test_outcomes[0:250, 1382]
    features_test = all_features[251:, :]
    labels_test = test_outcomes[251:, 1382]
    return features_train, labels_train, features_test, labels_test
```

The only difference here is that we are creating the random forest classifier by calling

`RandomForestClassifier(n_estimators=10).`

Also, the parameters to this algorithm are different. For instance, here min_sample_leaf is set to 1, we are not using any weight function, and the number of trees in the forest are set to 10. We have tried several different values for these parameters, and these values consistently produced good results. We also used the approach in Figure 5.3 for tuning the loss matrix and, in particular, FNP. The achieved results for some of the trials for a sample test case are given in Table 6.2. Similar to the results concerning logistic regression, we have considered many more values for FNP however in Table 6.2 we only present few of the tested values for FNP that clearly made a difference. Based on the results presented in the table, for this predictor $FNR = 10$ presented the best choice. Using this predictor we are able to reduce the number of unnecessary runs of this test by 12.2 percent, which shows a slight improvement with respect to the predictor based on logistic regression. Also on average, based on considered test cases, we observe a comparative improvement with 13.8 reduction in unnecessary runs of test cases.

7

Conclusions and Future Research

In the previous chapter, we discussed the use of logistic regression and random forests for training predictors that would guide us in choosing test cases that are most likely to fail. Generally, the use of the devised predictors relaxes the need to run all the test cases for each daily CI run. This is because the predictors will allow us to select a number of test cases that are most relevant. Due to this the feedback time is reduced and less time is spent on analysing the test results by developers. Consequently, testing this way is less expensive and this is due to the reduced number of test cases the CI needs to run. Even though the predictors based on random forests showed a slightly better performance, the achieved results are not tremendously impressive. However, they show the potential of the use of machine learning algorithms in relevant test case selection. We next discuss some of disadvantages and discuss some of the possibilities to achieve better prediction results as future works. Before, we dig deeper we first discuss some of the validity threats to our approach and results.

7.1 Validity Threats

Validity threats mainly concern how one designs the experiments for a study and how the way this is done can threaten the conclusions made by the study. During our study, we were limited by the fact that the data was provided to us and we did not have the time to change and design our experiments. In order to alleviate the effects of this, we tried to first understand the issues and limitations associated with the provided data, so as to provide guidelines on how to improve the quality of the data and its collection process. Secondly, we proposed features to extract as much information from the available data as possible, and to prevent its corresponding issues from harming the prediction performance. We did that by monitoring the performance of state-of-the-art algorithms.

Unfortunately, we observed that despite the fact that these remedies tended to slightly improve the performance, they still failed to deliver fully satisfactory results, since it was difficult to improve the performance by utilizing more complicated machine learning techniques. This, coupled with our learnings from literature, led us to believe that the main culprit was the data and its quality. Clearly, given significantly more time, we could have firstly gathered data with better quality and granularity, and, secondly, investigated the use of more advanced machine learning algorithms, preferably with more features.

7.2 Concluding Remarks

One of the possible fundamental reasons that prevent us from achieving better performance is the linear parameter dependence of the predictions on the features. This can be improved by means of:

- introducing nonlinear features,
- using more complicated prediction models which, for example, rely on neural networks, and possibly use deep learning for this purpose.

Another fundamental reason behind the under-performance of the predictor is the lack of access to considerable amount of data. In fact, in order to be able to use more involved features and to be able to use more complicated learning methodologies we need significantly larger amounts of data.

Aside from increasing the number of available data points, it is also possible to improve the performance of the proposed predictors by increasing the amount of information in the data. We believe this for the following reason. While being aware of the limitations of our available data, in this thesis we tried two classification algorithms, namely logistic regression and random forests. The logistic regression is one of the most commonly used algorithms owing to its simplicity and few tuning parameters, mainly the regularization parameter. Random forests on the other hand are more powerful and in many application areas pertaining to classification define the state-of-the-art. After extensive tuning of these algorithms, despite the differences of these classification algorithms, they both gave similar performances. Moreover, such algorithms have been used somewhat successfully in cases with much higher data granularity. These two observations lead us to believe that the main culprit for our unsatisfactory results, is our available data. This in turn means that the specific choice of ML algorithm and tuning is very likely to be secondary compare to getting more and more high resolution data. Consequently, we expect to achieve much better results in case we had data with much higher granularity, and more samples.

The higher granularity also allows us to introduce features which quantify the amount of changes made to modules. Currently this information is not available, since we are not aware of the amount of the changes in the modules or whether a line has been modified or a new functionality has been introduced to the software. We expect that exploiting information in higher granularity data as discussed above can significantly improve the performance of the predictors. Furthermore, the irregularities in the data, such as missing data or instability of the test cases has also adversely affected the achieved performance.

Concerning the data granularity, care must be taken as not all highly granular data are suitable for our purpose, see [83]. As was discussed in this paper, many prediction systems are based on mining the software revision history to analyze which changes are applied by whom, when and where, to be able to predict related changes, future defects and more. In such predictions the accuracy of the mined data is very crucial which is threatened by noise. One of the problems that causes noise is tangled changes, this is investigated in Kim Herzig and Andreas Zeller in this work. They claim that developers often commit unrelated or loosely related code changes in a single transaction. These atomic commits make changes to all modules appear related, but in reality they are not. Let us explain this with an example, imagine a developer has three tasks namely, implementing a new feature, re-factor a module and fix a bug. She implements these tasks and commits her code to the repository adding a commit message only mentioning the bug fix. Such entangled change does not harm the development process but it compromises the accuracy of the prediction model since in reality these changes are not related to one another. Herzig and Zeller claim that the tangled changes occur quiet frequently in software development and up to 15% of bug fixing change sets applied to their projects are tangled changes and hence need to be handled with care.

Delivering high quality software products by host company is of great importance. Particularly, for telecom-grade companies and based on their service level agreements, the provided software needs to enjoy an up time or availability of 99.999 percent. This gives a critical role and extreme importance to the testing procedure that is used to ensure such high quality of the product. Based on this and as a policy, as long as the thorough testing of the product does not prove to be prohibitive and the automated procedures for economical testing does not guarantee such high quality, Ericsson and companies alike would be reluctant to adopt or accommodate machine learning based techniques in their testing frameworks. Having said that, considering the constant advancement in the area of machine learning and ever growing size and improving quality of the available data, these methods show great promise. This was also highlighted by the work presented in this thesis, where despite the problems with the available data (concerning the amount information available in the data and the irregularities in it) the produced predictors resulted in a decrease in the number of executed test cases.

Appendix A: Feature Extraction Code

In this appendix we present the code for extracting the features based on the data concerning the modules and test case outcomes.

Module-based Features

The following excerpt of code produces the features concerning the module changes.

```
%% Initialization parameters

load ModuleDataFixed.mat
load ComprehensiveModuleList.mat
Time_window_length = 7;  % time window
Start_date = '01-Oct-2014';
Final_date = '31-Aug-2015';
Number_days_data = datenum(Final_date)-datenum(Start_date) + 1;
ModuleNamesList = InitialModules;

%===========================================================================
%% Feature regarding modified modules
%===========================================================================

Modified_modules_feature = zeros(length(ModuleNamesList),...
                        Number_days_data - Time_window_length + 1);
k = 0;
for i = Time_window_length - 1:Number_days_data - 1
    k = k + 1
    Curr_day = datestr(datenum(Start_date) + i);
    CurrDayModName = ModuleNames(strcmp(Dates,Curr_day));
    CurrDayModVer = Versions(strcmp(Dates,Curr_day));
    Prev_day = datestr(datenum(Start_date) + i - 1);
    PrevDayModName = ModuleNames(strcmp(Dates,Prev_day));
    PrevDayModVer = Versions(strcmp(Dates,Prev_day));
    [ModChange] = modifiedModule(ModuleNamesList,...
        PrevDayModName,PrevDayModVer,CurrDayModName,CurrDayModVer);
    Modified_modules_feature(:,k) = ModChange;
end

%==========================================================================
%% Feature regarding module rate of change
%==========================================================================

Modules_rate_of_change_feature = zeros(length(ModuleNamesList),...
                                Number_days_data - Time_window_length + 1);
k = 0;
for i = Time_window_length - 1:Number_days_data - 1
    k = k + 1
    temp = datevec(datestr(datenum(Start_date) + i));
    Date_list = listOfDays(temp(1),temp(2),temp(3),7);
    [ModuleChangeRate] = modRateChange(ModuleNamesList,...
                                Date_list,ModuleNames,Dates,Versions);
    Modules_rate_of_change_feature(:,k) = ModuleChangeRate;
end

%==========================================================================
%% Block change indicator feature
%==========================================================================

block_names = {'b2b','dia','sgc','sgn','trc','hiw','oab','sgcc','cha',...
    'mph','reg','sgm','tra','agsa','cpc','gpo','perf','sctp','syf',...
    'evip','hcfa','nih','plc','syfd','ftm','gcp','lpo','om','rcm','smm',...
    'gem','sgc','mpo','sys'};
Block_change_indicator_feature = zeros(length(block_names),...
                                Number_days_data - Time_window_length + 1);
k = 0;
for i = Time_window_length - 1:Number_days_data - 1
    k = k + 1
    Curr_day = datestr(datenum(Start_date) + i);
    CurrDayModName = ModuleNames(strcmp(Dates,Curr_day));
    CurrDayModVer = Versions(strcmp(Dates,Curr_day));
    Prev_day = datestr(datenum(Start_date) + i - 1);
    PrevDayModName = ModuleNames(strcmp(Dates,Prev_day));
    PrevDayModVer = Versions(strcmp(Dates,Prev_day));
    BlockChange = modifiedBlockModule(ModuleNamesList,block_names,...
                PrevDayModName,PrevDayModVer,CurrDayModName,CurrDayModVer);
    Block_change_indicator_feature(:,k) = BlockChange;
end

%===========================================================================
%% Block rate of change feature
%===========================================================================

block_names = {'b2b','dia','sgc','sgn','trc','hiw','oab','sgcc','cha',...
    'mph','reg','sgm','tra','agsa','cpc','gpo','perf','sctp','syf',...
    'evip','hcfa','nih','plc','syfd','ftm','gcp','lpo','om','rcm','smm',...
    'gem','sgc','mpo','sys'};
Block_rate_of_change_feature = zeros(length(block_names),...
                                Number_days_data - Time_window_length + 1);
k = 0;
for i = Time_window_length - 1:Number_days_data - 1
    k = k + 1
    temp = datevec(datestr(datenum(Start_date) + i));
    Date_list = listOfDays(temp(1),temp(2),temp(3),7);
    [blockChangeRate blockNames] = blockRateChange(ModuleNamesList,...
                         block_names,Date_list,ModuleNames,Dates,Versions);
    Block_rate_of_change_feature(:,k) = blockChangeRate;
end
```

In this code snippet, the functions

- `modifiedModule()`
- `modRateChange()`
- `modifiedBlockModule()`
- `blockRateChange()`

play the pivotal roles. Specifically these functions are responsible to extracting

- the status of different modules, that is whether they have been changed or not;
- the rate of change of each module;
- the status of different blocks of modules, that is whether any module in the block has been modified;
- the rate of change of blocks of modules,

respectively. The script for these functions are reported as follows.

```
%===========================================================================
function [ModChange] = modifiedModule(ModuleNamesList,PrevDayModName,...
                                PrevDayModVer,CurrDayModName,CurrDayModVer)
%===========================================================================

[PrevDayModName PrevDayModVer] = moduleFix(PrevDayModName,PrevDayModVer);
[CurrDayModName CurrDayModVer] = moduleFix(CurrDayModName,CurrDayModVer);
ModName = ModuleNamesList;
index = [];
ModChange = zeros(length(ModuleNamesList),1);
if length(PrevDayModName) <= length(CurrDayModName)
    for i = 1:length(PrevDayModName)
        temp = find(strcmp(PrevDayModName{i},CurrDayModName));
        if ~isempty(temp)

            index = [index temp];
            if ~strcmp(PrevDayModVer{i},CurrDayModVer{temp})
                ModChange(find(strcmp(PrevDayModName{i},...
                                                    ModuleNamesList))) = 1;
            end
        end
    end
    if length(index) ~= length(CurrDayModVer)
        temp = ones(size(CurrDayModVer));
        temp(index) = 0;
        for i = find(temp)'
            ModChange(find(strcmp(CurrDayModName{i},ModuleNamesList))) = 1;
        end
    end
else
    for i = 1:length(CurrDayModName)
        temp = find(strcmp(CurrDayModName{i},PrevDayModName));
        if ~isempty(temp)

            index = [index temp];
            if ~strcmp(CurrDayModVer{i},PrevDayModVer{temp})
                ModChange(find(strcmp(CurrDayModName{i},...
                                                    ModuleNamesList))) = 1;
            end
        end
    end
    if length(index) ~= length(PrevDayModName)
        temp = ones(size(PrevDayModName));
        temp(index) = 0;
        for i = find(temp)'
            ModChange(find(strcmp(PrevDayModName{i},ModuleNamesList))) = 1;
        end
    end
end

%===========================================================================
function [ModuleChangeRate ModulesNames]= modRateChange(ModuleNamesList,...
                                           Date_list,Modules,Dates,Version)
%===========================================================================

Nu_mod = length(ModuleNamesList);
ModulesNames = ModuleNamesList;
ModuleChangeRate = zeros(Nu_mod,1);
for i = length(Date_list):-1:2
    Mod_name_curr = Modules(strcmp(Date_list(i),Dates));
    Mod_ver_curr = Version(strcmp(Date_list(i),Dates));
    [CurrDayModName CurrDayModVer] = moduleFix(Mod_name_curr,Mod_ver_curr);
    Mod_name_prev = Modules(strcmp(Date_list(i-1),Dates));
    Mod_ver_prev = Version(strcmp(Date_list(i-1),Dates));
    [PrevDayModName PrevDayModVer] = moduleFix(Mod_name_prev,Mod_ver_prev);
    ModChange = modifiedModule(ModuleNamesList,PrevDayModName,...
                               PrevDayModVer,CurrDayModName,CurrDayModVer);
    ModuleChangeRate = ModuleChangeRate + ModChange;
end
ModuleChangeRate = ModuleChangeRate/length(Date_list);

%===========================================================================
function BlockChange = modifiedBlockModule(ModuleNamesList,block_names,...
                PrevDayModName,PrevDayModVer,CurrDayModName,CurrDayModVer)
%===========================================================================

BlockModChange = zeros(length(block_names),1);
BlockModNumber = zeros(length(block_names),1);
[PrevDayModName PrevDayModVer] = moduleFix(PrevDayModName,PrevDayModVer);
[CurrDayModName CurrDayModVer] = moduleFix(CurrDayModName,CurrDayModVer);
ModChange = modifiedModule(ModuleNamesList,PrevDayModName,PrevDayModVer,...
                                             CurrDayModName,CurrDayModVer);
for j = 1:length(block_names)
    for i = 1:length(ModChange)
        if length(block_names{j})<= length(ModuleNamesList{i})
            if strcmp(block_names{j},ModuleNamesList{i}...
                                                (1:length(block_names{j})))
                BlockModNumber(j) = BlockModNumber(j) + 1;
                if ModChange(i)
                    BlockModChange(j) = BlockModChange(j) + 1;
                end
            end
        end
    end
end
BlockModNumber(find(BlockModNumber==0)) = 1;
BlockChange = BlockModChange./BlockModNumber;

%===========================================================================
function [blockChangeRate blockNames] = blockRateChange(ModuleNamesList,...
                                blockNames,Date_list,Modules,Dates,Version)
%===========================================================================

blockChangeRate = zeros(length(blockNames),1);
for i = length(Date_list):-1:2
    Mod_name_curr = Modules(strcmp(Date_list(i),Dates));
    Mod_ver_curr = Version(strcmp(Date_list(i),Dates));
    Mod_name_prev = Modules(strcmp(Date_list(i-1),Dates));
    Mod_ver_prev = Version(strcmp(Date_list(i-1),Dates));
    blockChangeRate = blockChangeRate + modifiedBlockModule(...
                  ModuleNamesList,blockNames,Mod_name_prev,Mod_ver_prev,...
                                               Mod_name_curr,Mod_ver_curr);
end
blockChangeRate = blockChangeRate/length(Date_list);
```

Features Related to Test Cases

The following excerpt of code produces the features concerning the test cases results.

```
%% Initialization parameters

load TestDataFixed.mat
load ComprehensiveTestList.mat
Time_window_length = 7;  % days
Start_date = '01-Oct-2014';
Final_date = '31-Aug-2015';
Number_days_data = datenum(Final_date)-datenum(Start_date) + 1;
TestNamesList = InitialTest;
NumberOfTests = length(TestNamesList);
Test_names = TestNames;
```

```
Suite_names = suiteExtraction(TestNamesList);


%============================================================================
%% Features regarding test failure and execution rate
%============================================================================

Test_failure_rate_feature = zeros(length(TestNamesList),Number_days_data...
                                                 - Time_window_length + 1);
Test_execution_rate_feature = zeros(length(TestNamesList),...
                                Number_days_data - Time_window_length + 1);
k = 0;
for i = Time_window_length - 1:Number_days_data - 1
    k = k + 1
    temp = datevec(datestr(datenum(Start_date) + i));
    Date_list = listOfDays(temp(1),temp(2),temp(3),7);
    [TestFailRate ExecutionRate] = failureRate(TestNamesList,Date_list,...
                                                     OK,Dates,Test_names);
    Test_failure_rate_feature(:,k) = TestFailRate;
    Test_execution_rate_feature(:,k) = ExecutionRate;
end

%============================================================================
%% Feature regarding test suite failure rate
%============================================================================

Test_suite_failure_rate_feature = zeros(length(Suite_names),...
                                Number_days_data - Time_window_length + 1);
k = 0;
for i = Time_window_length - 1:Number_days_data - 1
    k = k + 1
    temp = datevec(datestr(datenum(Start_date) + i));
    Date_list = listOfDays(temp(1),temp(2),temp(3),7);
    [SuiteFailRate skipped] = suiteFailureRate(Suite_names,Date_list,...
                                                     OK,Dates,Test_names);
    Test_suite_failure_rate_feature(:,k) = SuiteFailRate;
end
```

In this code snippet, the functions

- `failureRate()`
- `suiteFailureRate()`

generate the necessary features. Specifically these functions are responsible to extracting

- each test case's failure rate and execution rate;
- the failure rate of each of the test suites,

respectively. The script for these functions are reported as follows.

```
%==========================================================================
function [TestFailRate ExecutionRate] = failureRate(Test_names,...
                                  Date_list,FailPass,Dates,test_case_names)
%==========================================================================

Number = length(Test_names);
TestFailRate = zeros(Number,1);
NumberOfRuns = zeros(Number,1);
for i = 1:length(Date_list)
    Temp_test = test_case_names(strcmp(Date_list(i),Dates));
    Temp_result = FailPass(strcmp(Date_list(i),Dates));
    for j = 1:length(Test_names)
        if sum(strcmp(Test_names(j),Temp_test))
            if strcmp(Temp_result(strcmp(Test_names(j),Temp_test)),'failed')
                TestFailRate(j) = TestFailRate(j) + 1;
                NumberOfRuns(j) = NumberOfRuns(j) + 1;
            elseif strcmp(Temp_result(strcmp(Test_names(j),Temp_test)),'ok')
                NumberOfRuns(j) = NumberOfRuns(j) + 1;
            end
        end
    end
end
NumberOfRuns(find(NumberOfRuns==0)) = 1;
TestFailRate = TestFailRate./NumberOfRuns;
ExecutionRate = NumberOfRuns/length(Date_list);

%==========================================================================
function [SuiteFailRate skipped] = suiteFailureRate(Suite_names,...
                                  Date_list,FailPass,Dates,test_case_names)
%==========================================================================

SuiteFailRate = zeros(length(Suite_names),1);
NumberOfRuns = zeros(length(Suite_names),1);
skipped = zeros(length(Suite_names),1);
for i = 1:length(Date_list)
    Temp_test = test_case_names(strcmp(Date_list(i),Dates));
    Temp_result = FailPass(strcmp(Date_list(i),Dates));
    for j = 1:length(Suite_names)
        suite = Suite_names(j);
        suite = [char(suite) '_SUITE'];
        temp = find(strncmp(suite,Temp_test,length(suite)));
        if ~isempty(temp)
            temp1 = sum(strcmp(Temp_result(temp),'failed'));
            temp2 = sum(strcmp(Temp_result(temp),'ok'));
            skipped(j) = skipped(j) + sum(strcmp(Temp_result(temp),...
                                                        'auto_skipped'));
            SuiteFailRate(j) = SuiteFailRate(j) + temp1;
            NumberOfRuns(j) = NumberOfRuns(j) + temp1 + temp2;
        end
    end
end
NumberOfRuns(find(NumberOfRuns==0)) = 1;
SuiteFailRate = SuiteFailRate./NumberOfRuns;
```

Bibliography

[1] R. D. Banker, S. M. Datar, C. F. Kemerer, A model to evaluate variables impacting the productivity of software maintenance projects, Management Science 37 (1) (1991) 1–18.

[2] J. Whittaker, et al., What is software testing? and why is it so hard?, Software, IEEE 17 (1) (2000) 70–79.

[3] K. L. Heninger, Specifying software requirements for complex systems: New techniques and their application, Software Engineering, IEEE Transactions on (1) (1980) 2–13.

[4] M. Gligoric, Regression test selection: Theory and practice, Ph.D. thesis, The University of Illinois at Urbana-Champaign (2015).

[5] W. E. Lewis, Software testing and continuous quality improvement, CRC press, 2016.

[6] P. C. Jorgensen, Software Testing: A Craftsman's Approach, 1st Edition, CRC Press, Inc., Boca Raton, FL, USA, 1995.

[7] P. E. Strandberg, W. Afzal, T. J. Ostrand, E. J. Weyuker, D. Sundmark, Automated system-level regression test prioritization in a nutshell, IEEE Software 34 (4) (2017) 30–37.

[8] Y. Zhauniarovich, A. Philippov, O. Gadyatskaya, B. Crispo, F. Massacci, Towards black box testing of android apps, in: Availability, Reliability and Security (ARES), 2015 10th International Conference on, IEEE, 2015, pp. 501–510.

[9] P. A. Laplante, M. Kassab, N. L. Laplante, J. M. Voas, Building caring healthcare systems in the internet of things, IEEE Systems Journal 2017.

[10] P. M. Jacob, M. Prasanna, A comparative analysis on black box testing strategies, in: Information Science (ICIS), International Conference on, IEEE, 2016, pp. 1–6.

[11] M. Beller, G. Gousios, A. Panichella, A. Zaidman, When, how, and why developers (do not) test in their ides, in: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ACM, 2015, pp. 179–190.

[12] F. L. Butt, S. N. Bhatti, S. Sarwar, A. M. Jadi, A. Saboor, Optimized order of software testing techniques in agile process–a systematic approach, International Journal of Advanced Computer Science and Applications (ijacsa), 2017, 8 (1).

[13] M. Godfrey, Q. Tu, Growth, evolution, and structural change in open source software, in: Proceedings of the 4th international workshop on principles of software evolution, ACM, 2001, pp. 103–106.

[14] S. Basri, N. Kama, F. Haneem, S. A. Ismail, Predicting effort for requirement changes during software development, in: Proceedings of the Seventh Symposium on Information and Communication Technology, ACM, 2016, pp. 380–387.

[15] A. Orso, G. Rothermel, Software testing: a research travelogue (2000–2014), in: Proceedings of the on Future of Software Engineering, ACM, 2014, pp. 117–132.

[16] U. Kanewala, J. M. Bieman, A. Ben-Hur, Predicting metamorphic relations for testing scientific software: a machine learning approach using graph kernels, Software testing, verification and reliability 26 (3) (2016) 245–269.

[17] S. Christa, V. Madhusudhan, V. Suma, J. J. Rao, Software maintenance: From the perspective of effort and cost requirement, in: Proceedings of the International Conference on Data Engineering and Communication Technology, Springer, 2017, pp. 759–768.

[18] R. Malhotra, A. Chug, Comparative analysis of agile methods and iterative enhancement model in assessment of software maintenance, in: Computing for Sustainable Global Development (INDIACom), 2016 3rd International Conference on, IEEE, 2016, pp. 1271–1276.

[19] M. Bures, T. Cerny, M. Klima, Prioritized process test: More efficiency in testing of business processes and workflows, in: International Conference on Information Science and Applications, Springer, 2017, pp. 585–593.

[20] D. Hao, L. Zhang, L. Zang, Y. Wang, X. Wu, T. Xie, To be optimal or not in test-case prioritization, IEEE Transactions on Software Engineering 42 (5) (2016) 490–505.

[21] S. Sharma, U. Chandra, P. Jain, A literature survey on automation of test data generation for branch coverage testing using genetic algorithm, International Journal of Computational Intelligence Research 13 (6) (2017) 1521–1531.

[22] G. Fraser, M. Staats, P. McMinn, A. Arcuri, F. Padberg, Does automated unit test generation really help software testers? a controlled empirical study, ACM

Transactions on Software Engineering and Methodology (TOSEM) 24 (4) (2015) 23.

[23] L. Kumar, S. K. Rath, A. Sureka, Using source code metrics to predict change-prone web services: A case-study on ebay services, in: Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE), IEEE Workshop on, IEEE, 2017, pp. 1–7.

[24] B. Caglayan, A. T. Misirli, A. B. Bener, A. Miranskyy, Predicting defective modules in different test phases, Software Quality Journal 23 (2) (2015) 205–227.

[25] E. L. Alves, T. Massoni, P. D. de Lima Machado, Test coverage of impacted code elements for detecting refactoring faults: An exploratory study, Journal of Systems and Software 123 (2017) 223–238.

[26] S. Noekhah, N. B. Salim, N. H. Zakaria, Predicting software reliability with a novel neural network approach, in: International Conference of Reliable Information and Communication Technology, Springer, 2017, pp. 907–916.

[27] E. Giger, M. Pinzger, H. C. Gall, Comparing fine-grained source code changes and code churn for bug prediction, in: Proceedings of the 8th Working Conference on Mining Software Repositories, ACM, 2011, pp. 83–92.

[28] N. Dini, A. Sullivan, M. Gligoric, G. Rothermel, The effect of test suite type on regression test selection, in: Software Reliability Engineering (ISSRE), 2016 IEEE 27th International Symposium on, IEEE, 2016, pp. 47–58.

[29] D. Card, D. S. Lee, Z. Pei, A. Weber, Regression kink design: Theory and practice, in: Regression Discontinuity Designs: Theory and Applications, Emerald Publishing Limited, 2017, pp. 341–382.

[30] R. Feldt, Do system test cases grow old?, in: Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on, IEEE, 2014, pp. 343–352.

[31] A. Gotlieb, D. Marijan, Flower: optimal test suite reduction as a network maximum flow, in: Proceedings of the 2014 International Symposium on Software Testing and Analysis, ACM, 2014, pp. 171–180.

[32] J. Holck, N. Jørgensen, Continuous integration and quality assurance: A case study of two open source projects, Australasian Journal of Information Systems 2003/2004 special edition 11 (1).
URL `http://journal.acs.org.au/index.php/ajis/article/view/145`

[33] S. Stolberg, Enabling agile testing through continuous integration, in: Agile Conference, 2009. AGILE'09., IEEE, 2009, pp. 369–374.

[34] J. Humble, D. Farley, Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation, 1st Edition, Addison-Wesley Professional, 2010.

[35] B. Adams, S. McIntosh, Modern release engineering in a nutshell–why researchers should care, in: Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on, Vol. 5, IEEE, 2016, pp. 78–90.

[36] A. Labuschagne, L. Inozemtseva, R. Holmes, Measuring the cost of regression testing in practice: A study of java projects using continuous integration, in: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, ACM, New York, NY, USA, 2017, pp. 821–830.
URL `http://doi.acm.org/10.1145/3106237.3106288`

[37] Labuschagne, Adriaan, Continuous integration build failures in practice, Master's thesis (2016).
URL `http://hdl.handle.net/10012/10683`

[38] A. Keus, A. Dyck, How much does testing cost?, Full-scale Software Engineering/Current Trends in Release Engineering 2016 p.1.

[39] T. M. Khoshgoftaar, E. B. Allen, N. Goel, A. Nandi, J. McMullan, Detection of software modules with high debug code churn in a very large legacy system, in: Software Reliability Engineering, 1996. Proceedings., Seventh International Symposium on, IEEE, 1996, pp. 364–371.

[40] N. Nagappan, T. Ball, B. Murphy, Using historical in-process and product metrics for early estimation of software failures., in: ISSRE, Vol. 6, Citeseer, 2006, pp. 62–74.

[41] M. Noorian, E. Bagheri, W. Du, Machine learning-based software testing: Towards a classification framework., in: SEKE, 2011, pp. 225–229.

[42] L. Briand, Y. Labiche, Z. Bawar, Using machine learning to refine black-box test specifications and test suites, in: Quality Software, 2008. QSIC '08. The Eighth International Conference on, 2008, pp. 135–144.

[43] L. C. Briand, Novel applications of machine learning in software testing, in: Quality Software, 2008. QSIC'08. The Eighth International Conference on, IEEE, 2008, pp. 3–10.

[44] L. Briand, Y. Labiche, X. Liu, Using machine learning to support debugging with tarantula, in: Software Reliability, 2007. ISSRE '07. The 18th IEEE International Symposium on, 2007, pp. 137–146.

[45] G. Wikstrand, R. Feldt, J. K. Gorantla, W. Zhe, C. White, Dynamic regression test selection based on a file cache an industrial evaluation, in: International Conference on Software Testing Verification and Validation, 2009, pp. 299–302.

[46] Y. Brun, M. D. Ernst, Finding latent code errors via machine learning over program executions, in: Proceedings of the 26th International Conference on Software Engineering, IEEE Computer Society, 2004, pp. 480–490.

[47] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, B. Wang, Automated support for classifying software failure reports, in: Software Engineering, 2003. Proceedings. 25th International Conference on, IEEE, 2003, pp. 465–475.

[48] J. R. Anderson, Understanding contextual factors in regression testing techniques, Ph.D. thesis, North Dakota State University (2016).

[49] A. Shi, T. Yung, A. Gyori, D. Marinov, Comparing and combining test-suite reduction and regression test selection, in: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ACM, 2015, pp. 237–247.

[50] C. Plewnia, MSc Thesis, Achen University. (2015).

[51] R. Saha, M. Gligoric, Selective bisection debugging, in: International Conference on Fundamental Approaches to Software Engineering, Springer, 2017, pp. 60–77.

[52] A. Shi, S. Thummalapenta, S. K. Lahiri, N. Bjorner, J. Czerwonka, Optimizing test placement for module-level regression testing, in: Proceedings of the 39th International Conference on Software Engineering, IEEE Press, 2017, pp. 689–699.

[53] M. Harder, J. Mellen, M. D. Ernst, Improving test suites via operational abstraction, in: Proceedings of the 25th international conference on Software engineering, IEEE Computer Society, 2003, pp. 60–71.

[54] A. Mockus, D. M. Weiss, Predicting risk of software changes, Bell Labs Technical Journal 5 (2) (2000) 169–180.

[55] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, A. Ihara, K. Matsumoto, The impact of mislabelling on the performance and interpretation of defect prediction models, in: Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on, Vol. 1, IEEE, 2015, pp. 812–823.

[56] F. Zhang, A. E. Hassan, S. McIntosh, Y. Zou, The use of summation to aggregate software metrics hinders the performance of defect prediction models, IEEE Transactions on Software Engineering 43 (5) (2017) 476–491.

[57] J. Jiarpakdee, C. Tantithamthavorn, A. Ihara, K. Matsumoto, A study of redundant metrics in defect prediction datasets, in: Software Reliability Engineering Workshops (ISSREW), 2016 IEEE International Symposium on, IEEE, 2016, pp. 51–52.

[58] K. Ricken, A. Dyck, A survey on multi-objective regression test optimization, Full-scale Software Engineering/The Art of Software Testing (2017) 32.

[59] R. Lachmann, M. Felderer, M. Nieke, S. Schulze, C. Seidl, I. Schaefer, Multi-objective black-box test case selection for system testing, in: Proceedings of the Genetic and Evolutionary Computation Conference, ACM, 2017, pp. 1311–1318.

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

[61] T. J. Ostrand, E. J. Weyuker, R. M. Bell, Predicting the location and number of faults in large software systems, Software Engineering, IEEE Transactions on 31 (4) (2005) 340–355.

[62] K. Herzig, N. Nagappan, Empirically detecting false test alarms using association rules, in: 37th IEEE International Conference on Software Engineering (ICSE), Vol. 2, 2015, pp. 39–48.

[63] D. Marijan, A. Gotlieb, S. 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.

[64] S. Russell, P. Norvig, A. Intelligence, A modern approach, Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs p.27. vol.25. 1995.

[65] M. Mohri, A. Rostamizadeh, A. Talwalkar, Foundations of Machine Learning, The MIT Press, 2012.

[66] V. K. Te Ming Huang, I. Kopriva, Kernel based algorithms for mining huge data sets: Supervised, semisupervised, and unsupervised learning, volume 17 of studies in computational intelligence (2006).

[67] L. P. Kaelbling, M. L. Littman, A. W. Moore, Reinforcement learning: A survey, Journal of artificial intelligence research (1996) 237–285.

[68] C. M. Bishop, Pattern recognition and machine learning, springer, 2006.

[69] R. Caruana, A. Niculescu-Mizil, An empirical comparison of supervised learning algorithms, in: Proceedings of the 23rd international conference on Machine learning, ACM, 2006, pp. 161–168.

[70] X. Wang, P. M. Pardalos, A survey of support vector machines with uncertainties, Annals of Data Science 1 (3-4) (2015) 293–309.

[71] B. Yuhas, N. Ansari, Neural Networks in Telecommunications, Springer Publishing Company, Incorporated, 2012.

[72] S. Menard, Six approaches to calculating standardized logistic regression coefficients, The American Statistician 2012, pp. 218-223.

[73] J. He, Y. Zhang, X. Li, P. Shi, Learning naive bayes classifiers from positive and unlabelled examples with uncertainty, International Journal of Systems Science 43 (10) (2012) 1805–1825.

[74] J. P. Bradford, C. Kunz, R. Kohavi, C. Brunk, C. E. Brodley, Pruning decision trees with misclassification costs, in: Machine Learning: ECML-98, Springer, 1998, pp. 131–136.

[75] T. G. Dietterich, Ensemble methods in machine learning, in: Multiple classifier systems, Springer, 2000, pp. 1–15.

[76] G. Biau, Analysis of a random forests model, The Journal of Machine Learning Research 13 (1) (2012) 1063–1095.

[77] D. R. Cutler, T. C. Edwards Jr, K. H. Beard, A. Cutler, K. T. Hess, J. Gibson, J. J. Lawler, Random forests for classification in ecology, Ecology 88 (11) (2007) 2783–2792.

[78] F. C. Pampel, Logistic regression: A primer, Vol. 132, Sage, 2000.

[79] C.-Y. J. Peng, K. L. Lee, G. M. Ingersoll, An introduction to logistic regression analysis and reporting, The Journal of Educational Research 96 (1) (2002) 3–14.

[80] S. Dreiseitl, L. Ohno-Machado, Logistic regression and artificial neural network classification models: a methodology review, Journal of biomedical informatics 35 (5) (2002) 352–359.

[81] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al., Scikit-learn: Machine learning in python, The Journal of Machine Learning Research 12 (2011) 2825–2830.

[82] K. P. Murphy, Machine Learning: A Probabilistic Perspective, The MIT Press, 2012.

[83] K. Herzig, A. Zeller, The impact of tangled code changes, in: 10th IEEE Working Conference on Mining Software Repositories (MSR), 2013, pp. 121–130.