CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Machine Intelligence in Automated Performance Test Analysis

Master's thesis in Computer Science and Engineering

Elona Wallengren
Robin S. Sigurdson

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2018

Master's thesis 2018

Machine Intelligence in Automated Performance Test Analysis

Elona Wallengren
Robin S. Sigurdson

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2018

Machine Intelligence in Automated Performance Test Analysis
Elona Wallengren
Robin S. Sigurdson

Academic Supervisors: Patrizio Pelliccione, Eric Knauss, Computer Science and Engineering
Industry Supervisors: Rohit Guliani, Adam Bengtsson, Ericsson AB
Examiner: Regina Hebig, Computer Science and Engineering

Master's Thesis 2018
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2018

Machine Intelligence in Automated Performance Test Analysis
ELONA WALLENGREN AND ROBIN S. SIGURDSON
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Software testing is a large part of development and especially important for projects that practice Continuous Integration. In order to reduce the burden of testing and make the process more efficient, as much as possible is automated. In this thesis, a design science approach is used to investigate how machine intelligence can be used to improve the automation of the analysis of non-functional testing. A prototype is created in order to demonstrate the ability of machine intelligence methods to provide useful information about the relationships between different test cases and their histories. This prototype was found to be fairly accurate in its predictions of test results, could identify most related degradations across test cases, and was positively received by the testers. Based on the results of this thesis, machine intelligence was found to have great potential in dealing with the large amount of data created during testing.

Keywords: machine intelligence, software testing, root cause analysis, test oracles

Contents

1 Introduction **2**

2 Background **4**
2.1 Important Concepts and Related Work 4
2.1.1 Software Testing 4
2.1.2 Machine Intelligence 7
2.2 Case Company 8

3 Methods **10**
3.1 Research Purpose 10
3.2 Research Question 10
3.3 Research Methodology 11
3.3.1 Metrics 11

4 Results **13**
4.1 RQ1: Critical problems that can be solved with machine intelligence . 13
4.1.1 First Iteration 13
4.1.2 Second Iteration 13
4.1.3 Summary 15
4.2 RQ2: Potential promising solutions 15
4.2.1 First Iteration 15
4.2.2 Second Iteration 17
4.2.3 Summary 22
4.3 RQ3: The extent the implemented solutions solve the problems . . . 22
4.3.1 First Iteration 22
4.3.2 Second Iteration 28
4.3.3 Summary 29

5 Discussion **31**
5.1 Challenges 32
5.2 Threats to Validity 33
5.3 Contribution to Academic Research 33
5.4 Future Work 34

6 Conclusion **35**

Bibliography **36**

1
Introduction

Software testing is a necessary step in software development. It is also a large part of the labor, usually requiring at least 40% of the development effort and taking up even more work when reliability is paramount [21]. In addition to functional regression testing that checks the functionality of the software, non-functional regression testing also has to be completed to ensure that the system overall performs correctly [10]. Although research into testing has been done since there was software to test, the application of the research to industry remains a problem [7], and testing is often still a cumbersome and costly task [21].

Testing has a central role for software companies that follow Continuous Integration, which is a practice where the software being developed is continually updated as developers constantly work to improve the system [11]. As integrations are made multiple times per day [11], it is important to also test the software continuously in order to discover any problems as quickly as possible. Otherwise, testers may find it difficult to identify which integration caused the issue, and if too much time has passed, there may even have been further integrations that built on this faulty code, compounding the issues.

To limit the cost of testing, software companies are increasingly trying to automate as much of the process as possible. While test execution is often already automated [11], the analysis of the test results is still a time and labor intensive effort. With automated test executions, large amounts of data about the system can be generated, but analyzing this continually and finding the sources behind problems requires a large amount of manual labor. One way to automate some of this analysis is through an automated test oracle, which given a test execution, decides if the system performed correctly or not [3]. Although regression testing has been studied extensively, the automation of test oracles and verdicts remains a relatively unexplored area [3].

In this study, machine intelligence is used to automate and aid in non-functional performance regression test analysis. These techniques have previously been identified as a useful aid in software testing [5], but the combination of machine learning in regression testing still lacks research [25].

The proposed solutions were discussed and tested with a case company. The case company currently utilizes a test oracle but finds that their test analysis is still resource intensive and could be made more efficient. Previously, investigations at this case company have been made into these limitations and the challenges surrounding the automation of test analysis [8] [9]. The study's goal is to address some of these limitations using machine intelligence.

The thesis is organized in the following way: Chapter 2 explores the background

and previous work related to this research, as well as the case company. In Chapter 3, the methods are explained, and Chapter 4 shows the results. Chapter 5 contains the discussion of the results, and the thesis ends with the conclusion in Chapter 6.

2
Background

This chapter provides the background information to the study and discusses the area's related work. The central concepts are software testing and its automation. The case company and department are also introduced.

2.1 Important Concepts and Related Work

In order to understand the context of this thesis, some important concepts need to be defined, and relevant research related to these concepts needs to be discussed. The thesis studies test analysis at a department that conducts performance regression testing on a system that follows Continuous Integration, so these terms are all defined. When improving the test analysis process, the focus is on automation. Previous automation work has resulted in the creation of a test oracle, but root cause analysis still requires much manual effort. The machine intelligence methods used in this thesis to expand test automation are presented as well.

2.1.1 Software Testing

Software testing is commonly thought of as "bug fixing", but this definition is incomplete. According to Whittaker [26], testing also has to determine whether the System Under Test (SUT) fulfills its specifications in its environment. An important aspect of testing that separates it from other forms of code verification is that a test involves executing the code and examining the results instead of reviewing the code itself. Because of this feature, software testing is vital to finding unexpected errors and ensuring that the system functions correctly.

The two main categories within software testing are functional and non-functional, or structural, testing. Functional testing checks that the specifications for the program are fulfilled without considering the code directly and is therefore also referred to as black-box testing [26]. On the other hand, structural testing is considered white-box testing and instead verifies the system's structure and implementation [23]. This thesis will only consider non-functional testing.

2.1.1.1 Continuous Integration

Continuous Integration is an industry practice which, according to Fowler and Foemmel [11], emphasizes frequent integration of each developer's work into the main software. Often, multiple updates are made every day to the system, ensuring that the software is always a current reflection of everyone's work. Fowler and Foemmel

[11] state that this practice is enacted in order to enable faster development and reduce the separation between the developers and the users. If testing is conducted between each integration, identifying where faults in the system originated is simpler than with a slower update model.

2.1.1.2 Regression Testing

Regression testing is the testing and retesting of software after it has been modified [12]. As regression testing needs to be conducted whenever a change is made, continually updated software also needs continual testing to verify that its performance continues to meet the requirements [12]. In practice, complete regression testing can become infeasible for companies with large and complicated systems that practice Continuous Integration. Unless the thoroughness of the testing process is sacrificed, it can become the bottleneck in development [7].

The problem of rigorous testing can be handled both by limiting the number of tests that need to be executed [12] or by reducing the amount of work required to complete and analyze each test. When deciding what tests to run and how often to run them, there are four main types of approaches: minimization, selection, prioritization and optimization [25]. However, this study focuses on the second part of the solution, considering improvements to test analysis automation.

2.1.1.3 Performance Testing

Performance testing is defined by Foo et al. [10] as a type of non-functional testing conducted in order to identify degradations in the operation of a system. When this type of testing is done between each update of the software, it is referred to as performance regression testing. Performance regression tests can take anywhere from hours to days to complete, and usually generate a large collection of performance metrics and logs from the execution. These results are then compared to the requirements or previous test runs to determine if there has been a degradation of the performance, usually referred to as a regression.

2.1.1.4 Test Automation

When utilizing test automation, the goal is to have software handle parts or all of test execution, test analysis, and other control and reporting functionality [18]. Using automation, the amount of human involvement in the testing process is reduced, and some of the large costs of testing lessened. Although automated test scripts are often used in practice [18], a significant amount of labor is still required in the testing process [23]. In particular, industry adoption of automated test analysis is limited [10].

A 2012 case study by Engström, Feldt, and Torkar [7] explored the adoption of regression test automation and identified challenges faced in industry when applying the techniques. Their automation prototype was found to increase efficiency and transparency, but even though their tool was only used to give suggestions to human testers, risks still arose with its utilization. Some of these indirect effects were that the users may rely on the tool too much and the mental exertion required of the

testers may increase. These results show the importance of considering what effects a new tool will have on its users.

A study of industrial applications of automated regression analysis was done in 2015 [10]. The authors dealt with the problem of testing heterogeneous environments, which are situations where the system has to work on different hardware and software set-ups. The authors created a system for detecting regressions, which was then successfully used in industry to aid analysts in finding anomalies in test runs. The system used previous test results to create rules about what performance values a test run should have, and reported violations of these rules.

2.1.1.5 Test Oracles

The test oracle problem is defined by Barr et al. [3] as the issue of identifying whether the system completed a test execution correctly or not. A test oracle can be a human tester, who looks through logs from the test execution to determine the results, but automated oracles are becoming an increasingly popular area of research. Regardless of whether the oracle is human or machine, their role is to deliver a verdict of pass or fail for the test execution.

According to Barr et al. [3], there are three types of test oracles: specified, where explicit requirements are given; derived, where the oracle uses documentation or system properties to evaluate the test results; and implicit, which are general and use simple facts such as that crashes indicate a failure in the test. They also note that, when improving automated test oracles, there are two aspects where human input can be reduced: the writing of the oracles and the analysis of the tests.

Hierons [16] examined the inadequacies with current test verdicts. Hierons identifies the key features of a verdict machine; it needs to be consequential in its decisions, and each verdict should tell something about the system under test's properties. One conclusion made was that there is a need for verdicts that consider more than a single test run. This thesis seeks to address this issue of considering both multiple test cases and the history of each test case.

2.1.1.6 Root Cause Analysis

According to Rooney and Heuvel [24], root cause analysis (RCA) is a process or tool used to find out what, how, and why an event occurred. The goal is to find a specific underlying reason that can then be solved. The four main steps in the analysis is gathering the data, finding causal factors between events, deciding the specific root cause for the event being investigated, and creating recommendations.

In the context of software testing, root cause analysis is the investigation into the reasons behind a test failure. When a degradation is detected during testing, the cause must naturally be identified in order to restore the performance to its previous values, which is what root cause analysis entails. Finding the root cause can be difficult, especially when multiple updates to the system are made between each set of testing [14]. This problem is further exacerbated if the system continues to be modified while the analysis is conducted, as changes may be made that build on the faulty functionality or that cause further degradations [19]. As a result, reducing the time between integrations and conclusion of analysis is of high importance.

Automating root cause analysis is one approach to improving its efficiency. Lee, Cha, and Lee [19] found that their automated framework for analysis in database management systems was able to reduce the time required to find root causes by about 90%. Heger, Happe, and Farahbod [14] created a more general method that identifies root causes for any software with thorough unit tests. However, they note that there are overall few studies that consider root cause analysis in the context of performance regression testing [14].

One case that was examined in 2014 [22] is the identification of regression-causes in performance testing. In the study, a regression-causes repository was created and used to identify the reason behind regressions. This approach was found to be able to accurately identify causes specified in the repository.

2.1.2 Machine Intelligence

Machine intelligence is a term used to describe both artificial intelligence and machine learning. Machine learning is the main tool used in this thesis and is the process of software discovering properties about a system through data [5]. The field builds on the concept that there are patterns in the data that may be difficult for a human to see but that can be found through machine learning algorithms.

Two of the most common types of machine learning algorithms are classification and regression [13], which are the approaches that are used in this study. Classification is a task where the goal is to assign a category to each data point given a set of input values. In regression, the machine has to predict a numerical value for the data point based on a set of input values.

An important concept when conducting machine learning in particular, but also in general when creating statistical models, is overfitting. A model that is overfitted shows relationships in the data that exist in the sample used to create the model but are not true for the whole population [2]. Instead of showing the true connections between the independent and dependent variables, the specifics of the sample are given. This becomes a problem when, for example, the model is used to predict the result of a data point that was not included in the sample. The accuracy of this prediction will then be lower than the accuracy calculated by looking at the model's ability to predict the sample data. In order to create a model that has applicability beyond the given data set, some accuracy needs to be sacrificed in order to find relationships that are more universally true.

Briand [5] identified the usefulness and issues with applying machine learning to software testing. Briand states that test oracles are difficult to automate but that machine learning can be a helpful part of the process, especially in systems that are constantly changing. Image processing is given as an example, but this is also the case for the system studied in this thesis, where even the requirements on the system change over time. Briand also notes that the accuracy of the predictions made by the algorithm is not the only important metric when choosing a machine learning method but that the simplicity of the model can also be relevant. If understanding why a prediction has been made is of interest in addition to the prediction, a model with easy interpretability is preferable.

Few papers were found about machine intelligence in regression testing, which

Rosero, Gómez, and Rodriguez [25] note may be due to the fact that much of regression testing is done as part of the maintenance of a system, making industry unlikely to explore new methods in this field.

The two machine learning algorithms used in this thesis are neural networks and random forests, which are further discussed in the following two sections.

2.1.2.1 Neural Networks

Neural networks are machine learning models loosely based on the discoveries about neurons in neuroscience [13]. The unit in a neural network is referred to as a neuron, and receives a number of inputs. The neuron then has a set of functions for each neuron it is supposed to output to, and calculates this output based on the function for that output and the inputs it receives. The network is essentially a composition of functions, which are not directly given in the data but instead created through training using a learning algorithm [13]. In this study, we utilize feedforward neural networks, which are networks where information only goes one way, from the input to the output.

2.1.2.2 Random Forests

Another popular machine learning method is the random forest. The random forest is an ensemble method, which means that many simpler models' outputs are used to make one overall prediction [20]. The random forest consists of a set of regression or classification trees, which given an input, moves through a set of binary branches to a leaf. Each split in the tree partitions the input in some way, based on one or more of the input values, and the leaf at the end of all of the splits is the prediction from that tree. The whole forest's result is then the average of the results from each of the trees [20].

2.2 Case Company

Ericsson was the case company used in this study. The company offers Information and Communication Technology to mobile service providers, and their networks carry approximately 40% of all mobile traffic [1]. Over 100,000 employees work at Ericsson, and they hold 45,000 patents [1], further demonstrating their leading position in the industry.

The case department at Ericsson is responsible for performance testing of a system in the mobile network. The goal of the testing is to ensure that the system meets performance requirements, and if it does not, to identify the degradation as fast as possible, along with deciding what update caused the change as well as figuring out why the problem occurred. However, the department's testing ability is limited both by the computing power needed to run tests as well as their ability to analyze the tests they have executed. There are many testers working on the system spread across several locations at the case company. As a result, face to face meetings between all of the testers are not feasible, and communication can be difficult despite being important.

To aid in analysis, the department has a tool called Automatic Verdict, which functions as a test oracle and provides a standardized analysis of a specific test in a matter of minutes. Automatic Verdict currently checks a number of checkpoints defined for each test case and records their results as pass or fail. The checkpoints are created by the testers and can be requirements on values known as key performance indicators. A key performance indicator (KPI) is a numerical value describing a part of the test, such as the CPU or memory usage. The acceptable ranges for each checkpoint is either manually set based on expectations and previous experience or given by a sample run of the test. The testers use the oracle as a summary of the test run, and when there are regressions, are also given information about what part of the test is problematic.

However, there are a number of limitations on Automatic Verdict. Previously, investigations have been made into the shortcomings and the challenges surrounding the automation of test analysis at the case company [8] [9]. Some of the problems identified were unstable test data, unclear expectations and interpretations, usability, and difficulty of root cause analysis [8]. Some of these problems have been addressed since these studies, but Automatic Verdict still has a limited scope. The current test oracle also does not provide a reason why the test failed, and each result is given only for a specific execution of a specific test case with no context to other executions and other tests. The thesis aims to address some of these limitations using machine learning.

3
Methods

This chapter discusses the purpose, question, and methodology of the research. The metrics used to evaluate the prototype's predictions are also described.

3.1 Research Purpose

The purpose of this study was to apply machine intelligence to improve the case company's test oracle's ability to analyze non-functional performance regression test results. The goal was to provide more context to the test results, by considering the circumstances of each test execution, and how it relates to previous runs of the same test case, as well as other related test cases run on the same version of the system. We aimed to create a prototype that used the specific test case's history to show when test results are out of the ordinary, helping to identify significant degradations and limiting the need for re-runs. In addition, the prototype should also be able to identify correlated degradations across tests, aiding root cause analysis. This could reduce the amount of manual labor needed for analysis, leading to a shorter time between test completion and test analysis.

One purpose of the prototype was that it would provide context that would allow testers to more easily see when test cases that are not part of their daily work are affected by the same issues that affect test cases that they are responsible for. The goal here is to prevent having to conduct multiple instances of root cause analysis into problems that share a common root cause. This would improve the efficiency of test analysis by enabling better communication between testers.

3.2 Research Question

The following research questions were investigated:

- RQ1: What are critical problems faced by testers, when conducting non-functional performance regression testing, that can be solved with machine intelligence?
- RQ2: What potential promising solutions exist to the problems faced by testers?
- RQ3: To what extent do the implemented solutions solve the problems faced by testers?

3.3 Research Methodology

We used a design science research method because our goal was to build and analyze a new addition to automated test analysis [15]. Accordingly, the work was conducted in cycles with five phases: investigating a problem or question, creating one or more possible solutions to the problem, applying one of the solutions, analyzing how well the solution solved the problem, and finally identifying a new problem or aspect that was not solved in the previous cycle. The investigation phase seeks to answer RQ1, the creation and application of a solution tackles RQ2, and the analysis phase deals with RQ3.

The first cycle was started by investigating the problem by discussing it with engineers and developers at the case company. Unstructured interviews were conducted on a weekly basis with the industry supervisors as well as various other testers in the case department. Approximately 20 interviews were conducted with more than 10 testers from the case company. The chosen testers were the ones that worked most closely with the test cases we utilized in our prototype. In order to create and apply a solution, we created a prototype that utilized machine intelligence. The analysis phase consisted of examining the results from the prototype with the industry supervisors as well as determining its accuracy.

Once the prototype was created and analyzed, the problem identified for the next phase was to implement and expand the prototype so that it provided applicable information to the test analyzers at the case company. At the end of the first cycle and start of the second, this problem was identified and investigated and possible applications were brainstormed through further unstructured interviews and demonstrations with testers at the case department. We applied the most promising suggestions by building on the prototype created in the first cycle, focusing this time on creating helpful outputs to the user. This cycle was analyzed through more unstructured interviews with practitioners and through practitioners testing out the prototype.

3.3.1 Metrics

The two types of machine learning utilized in this study were classification and regression. For classification, the main measurements used were the Matthews correlation coefficient and the confusion matrix. For regression, the root mean square error was applied to evaluate the models.

For binary classification, the data is categorized into true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN). True positives are events that are predicted to occur that do occur, while false positives are events that are predicted to occur but do not. True negatives are events that are not predicted to occur that do not occur, while false negatives are predicted to not occur but do. A confusion matrix is simply a display of these four values, dividing the data up based on its predicted and actual categories.

The Matthews correlation coefficient (MCC) is a performance metric for classifiers, which is especially suited for data classification containing an imbalance of data [4]. The MCC ranges between -1 and 1, where 1 signifies that the classifier fits

the data perfectly [4]. Classifiers with an MCC value of -1 have complete disagreement with the data, while 0 indicates a model that has a negligible relationship with the data [4]. The closer the classifier's MCC value is to 1, the more accurate it is. The MCC is calculated using the following equation:

$$MCC = \frac{TP \cdot TN - FP \cdot FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}. \tag{3.1}$$

The root mean square error (RMSE) is used to evaluate regression models. The RMSE provides an unambiguous and differentiating measure of a model's performance [6]. The RMSE is calculated using the following equation:

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n} e_i^2}, \tag{3.2}$$

where n is the total number of points and e_i is the error, or the difference between the predicted and actual value, of point i.

4
Results

This chapter discusses the results obtained during the study from following the methodology outlined in the previous chapter. The data gathered from the prototype is presented.

4.1 RQ1: Critical problems that can be solved with machine intelligence

The first research question was answered in the investigation of the problem phase of the design science cycles. An initial problem was identified in the first iteration, and more specific problems were discovered during the second.

4.1.1 First Iteration

Overwhelming data. After being introduced to the case department by the industry supervisors, the problem was not clearly described or specified. Despite having an automated test oracle, there was still plenty of manual work required to analyze test execution data. Through unstructured interviews, it was evident that the amount of data generated through test executions was large enough that testers were unable to utilize it fully. The testers lacked knowledge about if and how different test cases executions were related, although a connection seemed natural since many of the test cases were similar. Since machine learning methods are designed to identify patterns in large data sets, they were particularly suited to addressing the testers' situation.

4.1.2 Second Iteration

In the second iteration, we used the prototype and related knowledge from the first iteration to further investigate the testers' specific problems caused by having to handle such a large amount of data. One result of the large amount of data generated through test executions is that a large amount of testers are required to analyze the results. There are several groups of testers, each with their own set of test results. Because of the scale of the testing, sufficient and efficient communication is difficult; an illustration of this is shown in Figure 4.1.

Error report communication. A specific scenario where communication problems lead to inefficiencies is fault reporting and root cause analysis. When a tester finds a fault in the system, they create a report about it, identifying the test cases

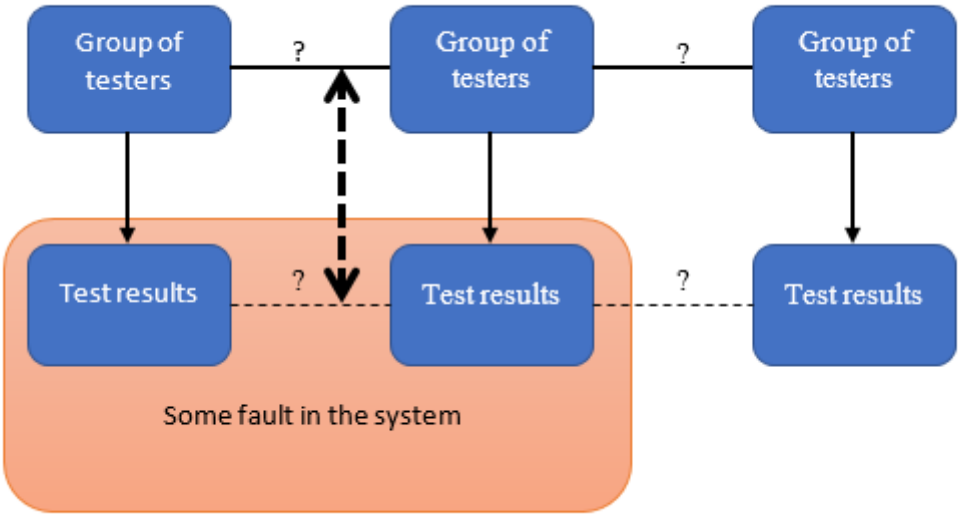

Figure 4.1: An illustration of the difficulties in communication. There are several groups of testers, and they all have their own test results to analyze. If there is a fault that affects test results across multiple groups, there is no automatic or simple way for the testers to know that this is the case or to inform the other groups about it.

and builds affected and providing a description of the problem. This is based on the tester's own awareness of the issue, but the fault may also affect test cases that the tester does not work closely with. Because there are so many different people each independently working on their own test cases, the tester who first finds the fault may not be aware of all cases that should be listed in the report. The tester working on those cases would then have to run a search through all active reports for descriptions matching their error. If the second tester does not find the report created by the first, they will most likely create a new report, complete their own root cause analysis, and create another solution to the problem. Solving the problem twice is obviously quite inefficient, and something we seek to prevent from happening again in the future.

Verdict requirements over degradations. Another issue is that testers are mainly focused on changes in KPI values that cause checkpoints in the test oracle to fail. Changes in KPI values that do not cause the value to go outside of the range defined for the checkpoint are often assumed to be random fluctuations, since the system is not completely deterministic. Consider, for example, a KPI value has an acceptable range of 80 to 120 and is usually around 100. This can cause problems if, between two builds of the system, the value decreases from 100 to 82, and in the next build, the value further decreases to 78. The third build would be the one that the test oracle marks as a failure, and the tester would start looking for issues with the KPI based on that. However, the major performance degradation occurred between the first two builds, and the updates made between these two are the ones that need to be examined to find the root cause.

Re-runs. Another problem faced by the department is the issue of testers having to re-run test cases to figure out if the results in a given execution are caused by a temporary fluctuation in the system or an actual change in the performance caused by an update. Often, if a tester notices that an execution has failed the requirements set in the test oracle by a small amount, the tester will re-run the test case to see if the failure can be replicated. This becomes problematic for the case company because a significant amount of computing power is dedicated to running the same test scenario multiple times.

Because of the problems outlined above, test analysis is very time intensive, and not all test executions are manually analyzed. This means that there can be a few new executions of a test case when a tester analyzes it. If a degradation has occurred since the last analysis of the test case, root cause analysis begins with identifying in which execution the problem started and can be cumbersome. With the importance of quick root cause analysis to Continuous Integration, making test analysis more efficient would provide large benefits.

4.1.3 Summary

The main problem testers faced that could be solved with machine intelligence was the **overwhelming amount of data** generated during test executions. The specific problems that resulted from this issue with the data were **communication issues** and knowledge gaps surrounding fault reporting, the focus on **requirements in the test oracle instead of degradations**, and the issue of testers **re-running** test cases due to the non-deterministic nature of the system.

4.2 RQ2: Potential promising solutions

The second research question was answered during the creation and application of solutions phases in the design science cycles. In the first iteration, an initial prototype that predicts test results was created. Applications of the prototype that directly aid testers in analysis were created in the second iteration.

4.2.1 First Iteration

In order to address the problem of extracting information from the large data set and to determine if connections exist between test cases, we created a prototype. The focus was to make accurate predictions of test execution data using other test cases' executions. If the prototype was able to obtain accurate predictions of the data, it would demonstrate that there are connections between the test cases.

The implementation phase of the first iteration was started by creating classifiers to predict the pass or fail verdicts made by the test oracle. Then, we trained regressors using numerical data used by the test oracle to give a pass or fail verdict. The two methods used for the prototype were the same throughout; the only difference was that regressors were used in the second part instead of classifiers. However, before we could use the test execution data for an implementation of machine in-

telligence algorithms, we first had to pre-process the data, transforming it into an easily readable format.

4.2.1.1 Data Handling

The data consisted of logs from system test executions. The log used to represent each execution was either the log from the case company's test oracle containing a set of checkpoint results or the log containing the test case's numerical key performance indicators. A checkpoint is a requirement on a specific value or set of values describing a part of the test and can be either pass or fail. Most requirements are on the values known as key performance indicators. A key performance indicator (KPI) is a numerical value describing a part of the test, such as the CPU or memory usage. Both types of logs were text files and needed significant pre-processing to be usable when conducting machine learning.

First, the logs that are needed to train the machine learning model are identified. To do this, the type of test, the type of hardware, and the specific test name need to be given in order for the script to be able to find the log. Then, each line containing data from each log is saved in one of two text files, one for the input and one for the output of the model.

After the two text files are created, each distinct build and checkpoint or KPI was identified. These were used to create two NumPy arrays, where in each the first dimension corresponds to the build and the second to the checkpoint or KPI. The data was then read from the two text files and inserted into the correct position in the corresponding array. Rows or columns in the arrays containing very little data were removed, and any remaining missing data was replaced with the median value for that checkpoint or KPI. The data was rescaled to have a range of $[0, 1]$ in order to normalize the size of each KPI.

4.2.1.2 Classifier and regressor implementation

Based on input received from the problem phase, we built a prototype system that expands the functionality of the test oracle at the case company using machine learning. The goal was to create a system that was able to identify patterns in test results and checkpoint failures, aiding system testers in finding the problems that cause the tests to fail. We thought that the best approach was to create a neural network, but other options were also considered and tested. The system was trained using historical data with known results. The case company had already collected a large amount of data from their continuous testing, and more data was continually generated, so the model needed to be able to adapt as new results became available.

The two machine learning methods tested in this study were neural networks and random forests. The two machine learning methods were compared to logistic regression for classification and linear regression for regression. The prototype was implemented in the Python programming language. The libraries used were NumPy for general math and array handling, Keras with TensorFlow as the backend for the neural networks, and scikit-learn for the random forests and some data analysis. MatPlotLib was the library utilized to make the plots in the thesis.

For the neural networks, multiple feedforward networks were trialed. In the end, a simple network without hidden layers was chosen in order to enable easy analysis. The data was split into training and validation sets, where the network was given the training data and evaluated its progress based on the validation set to avoid over-fitting. K-fold cross validation was used to further improve the accuracy and consistency of the model. The number of folds was set based on the number of data points available for training.

Since our network does not have any hidden layers, each weight or bias in the network created during training only affected one output node. The parameters' independence allowed the training to be evaluated individually for each output dimension, minimizing the error in each output separately.

For the random forests, one random forest was created for each output dimension. A grid search was used to determine the correct hyper-parameters for each output.

When utilizing relatively simple models such as these, using machine learning at all may seem unnecessary, and a statistical linear regression approach may be appealing. However, the machine learning methods are advantageous in that they can ignore outliers that, for example, are errors in how the data is recorded and do not reflect the system's properties, and that they are able to handle incomplete data. Unlike statistical models, these techniques iteratively improve their predictions based on the data, and if some data points are misleading, their effect on the result can be removed through this process. This means that the machine learning methods create models that more accurately reflect the underlying system instead of attempting to create a perfect representation of the given data.

The prototype was tested after training by giving it new data not used in the training and then checking whether the predicted test results were consistent with the actual result. For example, the prototype could be trained to predict the results of a test based on other tests that are run more frequently. The neural network or the random forest could then be examined to find what parts of the more frequently run tests affected the predictions, thereby identifying correlations between tests.

4.2.2 Second Iteration

Based on our improved understanding of the specific problems faced by testers, we focused on implementing a system to support root cause analysis. The main problems addressed by this system were **communication issues** and knowledge gaps surrounding fault reporting, the focus on **requirements in the test oracle instead of degradations**, and the issue of testers **re-running** test cases due to the non-deterministic nature of the system.

When developing applications based on the predictions from the first iteration, the neural network was utilized instead of the random forest. Because both machine learning methods had similar performance, we could choose the option that was easier to work with, which in this case was the neural network. Since the neural network was implemented without any hidden layers, this model had the advantage of having a single number representing each input's effect on the output, whereas the analysis would be more complicated with an ensemble method such as the random forest. In addition, the numerical KPI data was used for the applications instead of

the binary checkpoint data in order to remove a layer of abstraction and obtain a more accurate view of the actual system.

One expansion to the prototype was a script that takes a test case and a build as input and checks if any KPIs have changed since the previous build. There is a threshold that the change has to be greater than in order for that KPI to be listed. The threshold can be adjusted based on expectations of stability in the value. A text file is created where each changed KPI is listed with a list of input KPIs that were important to the network in predicting this change. The input KPIs are listed according to their relative effect on the prediction, with the KPI having the highest effect given a score of 100. The effect is calculated by multiplying the change in that input KPI between the two builds by the weight in the neural network between the input KPI and the output KPI under consideration. In addition, if any of the test cases the KPIs belong to have a current error report, the report will be listed in the final column. An example of the output of this script is shown in Figure 4.2.

```
Current build: 500
TestCase1               Kpi1    -9.33e+02  -5.74   495
Effect    Name          Kpi     Change     %       Previous build   Reports
100       TestCase2     Kpi2    4.00e+00   7.45    495
95        TestCase3     kpi3    -7.00e+00  -16.28  495              Report1
76        TestCase4     kpi4    -1.00e+00  -2.33   495
71        TestCase5     kpi5    1.49e+03   11.09   495
50        TestCase6     kpi6    -8.70e+01  -100.0  495
34        TestCase8     kpi5    -3.83e+03  -6.16   495              Report2
32        TestCase8     kpi7    -7.92e+03  -5.62   495              Report2
29        TestCase9     kpi8    4.79e+03   5.31    495
26        TestCase8     kpi9    -2.00e+00  -1.42   495              Report2
19        TestCase7     kpi3    1.00e+00   inf     495
```

Figure 4.2: An example of the output of the script that finds connected degradations or changes in KPIs. In this example, only one output KPI is shown. This is Kpi1 from TestCase1. The KPI has decreased 5.74% since the previous build. The following rows list the input KPIs ranked according to their effect on the prediction of the degradation in Kpi1. For example, the increase in value of Kpi2 from TestCase2 has the biggest effect on the prediction of the decrease in Kpi1. The second biggest effect came from kpi3 in TestCase3. TestCase3 also had an error report created for a build between 495 to 500, so that is listed in the final column as Report1.

This application for finding degradations was tested by simulating the scenario where a tester is investigating a degradation in their test case caused by an issue that has already been identified in a different test case. This scenario is reproduced by going through all reports with multiple test cases on the same build. The application was run once for each test case in each report with the output set as that test case. Then, we checked whether the report was included in the list of related degradations. This answers the question of whether, given a report with multiple test cases, if one is removed, our application will tell the tester that degradations in that test case are related to that report. The total percentage of reports identified as related was calculated, and the reports that were not discovered were analyzed to determine if they affected the KPIs in the test cases at all.

To the problem of focusing on whether the **checkpoint requirements** are met **instead of** looking for **degradations** in the system, we propose a few solutions. If the script discussed above was run at the end of every test execution, then the tester responsible for that test case could be automatically notified if any KPIs changed significantly in value. From the script output, the tester would already have some information about any similar KPIs that also were affected and if there are any reports about the changes. As the complete script currently takes a few minutes to run, with the exact time depending on the amount of data that has to be analyzed, adding this script to the end of test executions is feasible if it would significantly aid test analysis. In addition, the script's time and resource requirements could be further improved. As it would not be necessary to train a completely new neural network whenever one new data point is added, if an update functionality was implemented, the costs of running the script would significantly decrease.

Another way we approached this issue was to create a script that identifies if a change in a KPI is larger than normal or not. This script does not utilize the neural network but is instead a more statistical implementation of machine intelligence. Given a specific test case and build, the script identifies how much the KPIs in this test case have changed since the build before the given build. Then, it calculates the nth percentile of change in that KPI, based on the historical data that we gathered when training the machine learning algorithms. n can be specified by the user, and the $n\%$ changes with the smallest absolute value are deemed "regular". Changes bigger than this boundary are marked as "unusual" in the file. An example of the output from this script is given in Figure 4.3.

Output		Regular Change	Current change	From build	To build	
Testcase	Kpi1	2.00e+00	7.00e+00	Build1	Build2	UNUSUAL!
Testcase	Kpi2	7.49e+02	5.04e+02	Build1	Build2	
Testcase	Kpi3	1.70e+00	-5.00e-01	Build1	Build2	
Testcase	Kpi4	2.67e+04	1.92e+04	Build1	Build2	
Testcase	Kpi5	1.50e+05	1.01e+05	Build1	Build2	
Testcase	Kpi6	1.04e+03	9.21e+02	Build1	Build2	

Figure 4.3: An example of the output of the script that calculates whether KPIs' changes are within the regular variations or not. The "regular change" column is calculated as a percentile, in this case 70%, of the absolute value of historical changes in that KPI. The current change is the amount that the KPI has changed, in this case, between Build1 and Build2. If the current change's absolute value is greater than the regular change, the change is marked as unusual.

In addition to analyzing the changes, we also implemented an analysis of the actual KPI values done in the same way. A visual example of the analysis of the KPI values and their changes is shown in Figure 4.4.

Another issue this second script helps to alleviate is testers having to **re-run** test cases to figure out if the results in a given execution are caused by a temporary fluctuation in the system or an actual change in the performance caused by an update. Our script would inform the testers if the given test execution's KPI values have changed more since the previous build than what is normal. For executions with changes within the boundaries, a re-run would most likely not provide any new information, as these types of variations are to be expected. If the change is marked

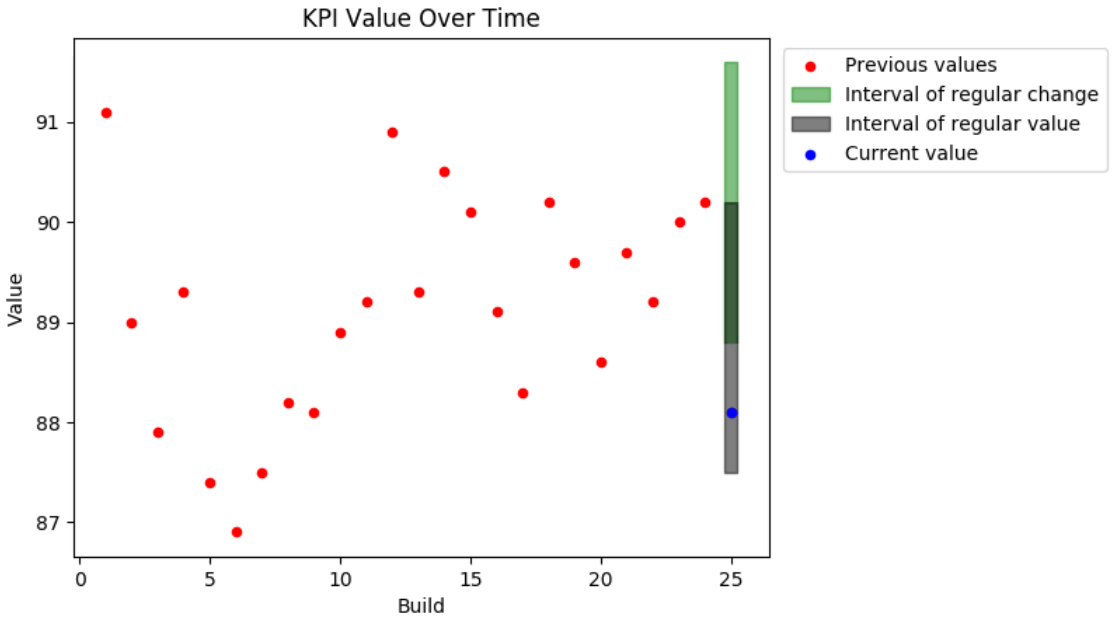

Figure 4.4: An example of a KPI's value over time and how the intervals for regular change and value are calculated. The interval of regular change is centered around the value of the KPI in the previous build and is here set to be size of the 70th percentile of the history of changes. The interval of regular value is here created between the 15th and 85th percentile of the KPI values. In this figure, the current value is within the range of regular values but has had an unusually large change in value compared to the previous value.

as unusual, the tester would then know the change is the result of either an error in the test case or a difference in the performance of the system.

The outputs of the prototype shown in Figures 4.2 and 4.3 were changed and improved several times during the course of the second iteration. Since a single tester often may not know much about other test cases, the numerical values from other test cases may be confusing. Based on discussions with our industry supervisors, we concluded that the amount of numerical information presented is overwhelming and does not need to be shown to the end user. A new output file was then created consisting of the most useful results from the two previous outputs. The file contains the suggested error reports and the information about whether the test case analyzed had any outliers in the data such as unusually large changes or unusually high or low values. An example of this output is shown in Figure 4.5.

```
Testcase1     Regular Value   Current Value   Regular Change   Current Change
KPI1          8.08e+01        7.93e+01        1.30e+00         -2.70e+00      UNUSUAL CHANGE!
KPI2          1.63e+04        1.53e+04        5.65e+02         -9.33e+02      UNUSUAL CHANGE!   LOW VALUE
KPI3          8.93e+01        8.86e+01        1.40e+00         -1.00e+00
KPI4          5.51e+05        5.20e+05        1.97e+04         -3.06e+04      UNUSUAL CHANGE!   LOW VALUE
KPI5          3.26e+06        3.06e+06        1.15e+05         -1.88e+05      UNUSUAL CHANGE!   LOW VALUE
KPI6          4.17e+04        3.86e+04        6.91e+02         -1.02e+03      UNUSUAL CHANGE!   LOW VALUE

Https://www.PathToTestcase1.se

TRs that might be conncted:
Report1: Description of problem from report
https://www.PathToReport1.se

Report2: Description of problem from report
https://www.PathToReport2.se
```

Figure 4.5: A figure showing the final output from the prototype. For each KPI in the test case, the current value and the change in that value since the previous build is shown. The median value of the KPI is shown in the column "Regular Value" and the 70th percentile of all changes in the KPI is shown in the column "Regular Change". These give the tester an idea of how the current results compare to historical trends. Changes greater than the 70th percentile are marked "UNUSUAL CHANGE!", values above the 85th percentile are marked "HIGH VALUE", and values below the 15th percentile are marked "LOW VALUE". Below the numerical data, there is a link to the test case on a database where graphs of the KPIs can be created and viewed. Below that is the list of related error reports identified by the neural network for this test case execution along with a short description of the report and a link to it.

4.2.2.1 Example scenario

The following is an example of how a tester could use the final prototype to aid in test analysis. The tester has a specific execution of a test case that they need to analyze. The tester looks at the results from the test oracle, and the execution may be marked as a fail or it may have passed, but the tester decides that they want to do a deeper analysis of this execution. The tester can then run our prototype, which usually takes around a minute but can take more depending on how much data there is for the input and output. The tester would specify their execution as the output and choose a related category of test cases for their input data.

Once the prototype is finished running, the tester would look at the final output, which will look similar to Figure 4.5. The tester can see if there is anything unusual about the KPIs in their execution, regardless of whether the value meets the requirements and is therefore accepted by the test oracle. There is also a link the tester can use to easily access graphing utilities of the KPI values' histories. If this data combined with the test oracle's result indicates to the tester that there was an error or degradation in the test case, the tester can then read through the suggested error reports and their descriptions to see if similar problems have already been noted. If they find one that matches their issue, they can use the link to read the full report and add their test case to it. If none of the suggested error reports match their problem, the tester can be more confident in the necessity of creating a new error report.

The result of this process is that the tester more easily gets access to data about their test case that helps them evaluate and analyze a test execution. This aids the tester in deciding whether a degradation necessitates a root cause analysis or not. The machine learning method helps to identify related error reports that may not have discovered otherwise, which prevents the testers from conducting a root cause analysis into a problem that has already been identified and possibly solved.

4.2.3 Summary

A prototype that predicts test execution data was created in order to demonstrate machine intelligence's ability to find relationships between test cases. In order to address the more specific problems, the prototype was expanded to provide a user-friendly output that, for each test execution, shows how the current execution relates to historical results and lists the error reports that may be related to that execution.

4.3 RQ3: The extent the implemented solutions solve the problems

The third research question was answered in the analysis of the solution phase of the design science cycles. Three different types of evaluations were made. In the first iteration, the accuracy of the prototype's predictions was assessed. In the second iteration, the two types of evaluations were the accuracy of the prototype in identifying relevant error reports and the testers' experiences using the prototype.

4.3.1 First Iteration

In this iteration, two data sets were analyzed, and both classifiers and regressors were tested.

4.3.1.1 Data Sets

Both data sets presented here consisted of data from capacity test cases executed on different hardware. One specific test case with a recent known regression was chosen as the output test case, while the others served as input for the algorithm.

Data Set 1 consisted of the checkpoint data from the test oracle and was used for classification. It contained 25 points, each one representing the test executions on a different build of the system. For each of the 25 points, there were 208 input dimensions and 6 output dimensions, and in the output, 127 of the results were pass, while 23 were fail. Since test failures are uncommon, looking at the models' ability to predict the failures specifically was important, and the MCC was a suitable performance metric.

Data Set 2 consisted of the KPI values from the same executions as Data Set 1 and was used for regression. There were 25 data points with 220 input dimensions and 6 output dimensions.

During the data collection of Data Set 1, we noticed that there was a lot of missing data. Missing data occurred because not every test case was executed for every build. In addition, if a checkpoint was created or removed, the data for that checkpoint would be missing before its creation or after its deletion. Even after discarding data points, a significant portion of the input data was still missing. In this case approximately 33% of the input data was missing, which can be seen in Figure 4.6.

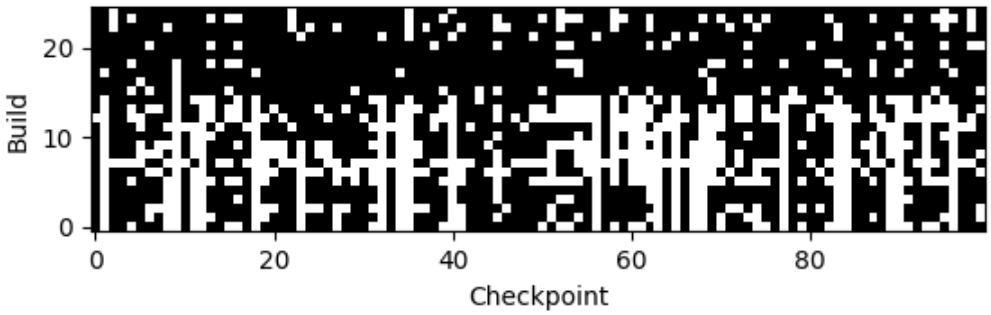

Figure 4.6: A figure showing to what extent checkpoint data is missing from Data Set 1. Existing data is marked with a black box, while a missing data point is white. All builds are shown for an arbitrary selection of 100 input dimensions.

When Data Set 2 was used, approximately 33% of the input data was missing, which is very similar to Data Set 1. A visual of the missing KPI data is shown in Figure 4.7.

In order to learn more about how the KPIs were related, the Pearson correlation coefficient was calculated for the KPIs, 30 of which are shown in Figure 4.8. Although there are some KPIs from different test cases that are highly correlated, those pairs of KPIs may still have instances where one KPI increases in value between two builds while the other decreases, as shown in Figure 4.9. This means that using a single KPI as a predictor for another will not provide accurate results.

4.3.1.2 Classification

Our first algorithms worked with the verdict data, Data Set 1, and could with very high accuracy predict the results from a particular test using others tests as input.

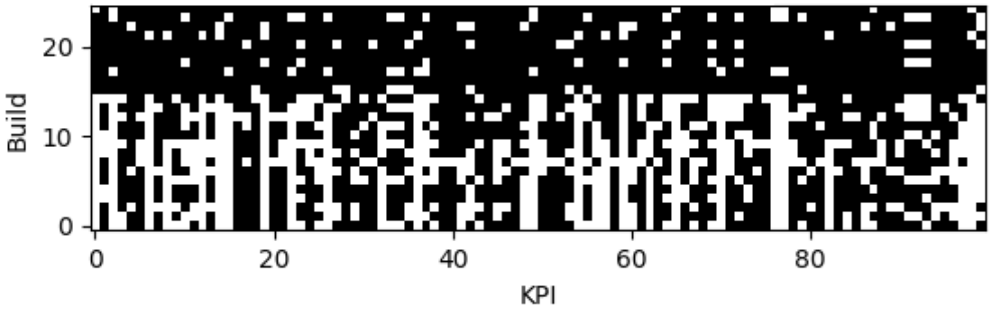

Figure 4.7: A figure showing to what extent KPI data is missing from Data Set 2. Existing data is marked with a black box, while a missing data point is white. All builds are shown for a random selection of 100 input dimensions.

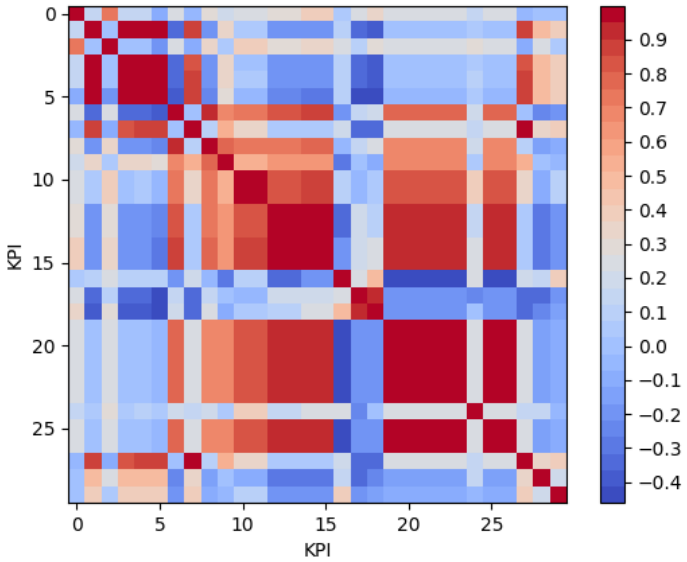

Figure 4.8: A heatmap showing the Pearson correlation coefficient between different KPIs from Data Set 2. The six output KPIs are shown first, and 24 of the input KPIs are displayed next. KPIs from the same test case are grouped together. The graph shows that there are some KPIs from different test cases that are highly correlated, but in general, the correlation varies widely between pairs of KPIs.

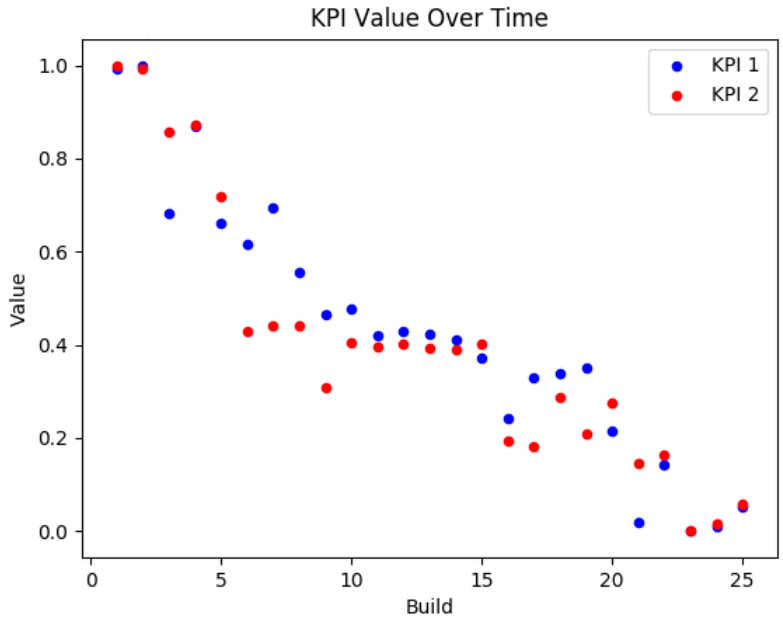

Figure 4.9: A figure showing two KPIs from Data Set 2, one from the output test case and one from one of the input test cases, with a Pearson correlation coefficient of 0.943. The KPIs are shown with their rescaled values. Although the correlation is high, there are cases where one KPI increases while the other decreases.

The logistic regression model was able to perfectly predict all of the outputs, as shown in Table 4.1. This meant that the logistic regression model had an MCC of 1.

The result from the random forest had an MCC of 0.840 and was therefore worse than the logistic regression model. The confusion matrix for the random forest is shown in Table 4.2. The biggest obstacle to the random forest's predictions was most likely the small number of data points.

		Predicted	
		Pass	Fail
Actual	Pass	127	0
	Fail	0	23

Table 4.1: Confusion matrix for the logistic regression model for Data Set 1.

		Predicted	
		Pass	Fail
Actual	Pass	127	0
	Fail	6	17

Table 4.2: Confusion matrix for the random forest for Data Set 1.

Training a neural network using the same data set, the results were better than the random forest's. The resulting network had an MCC of 1, performing as well

as the logistic regression. The confusion matrix for the neural network is shown in Table 4.3.

		Predicted	
		Pass	Fail
Actual	Pass	127	0
	Fail	0	23

Table 4.3: Confusion matrix for the neural network for Data Set 1.

For predicting whether checkpoints from the test oracle pass or fail, advanced machine learning methods were unnecessary, as the logistic regression model was able to perfectly predict all of the results.

4.3.1.3 Assessment of classification and transition to regression

Although the classifiers were accurate, knowing whether and why a checkpoint is pass or fail is not as meaningful as knowing this information about the underlying KPI values that determine the test oracle's classification. Based on input from our industry supervisors, we therefore decided that classifier predictions had limited usability for improving test analysis and therefore this project. We instead focused on the regression algorithms and moved on to using Data Set 2. The KPI values used in regression are the same values used by the test oracle to decide if a checkpoint passes, so looking at this data removes a layer of abstraction from the model.

4.3.1.4 Regression

The linear regression model was able to make extremely accurate predictions of the KPI data, having an average root mean square error of $5.77 \cdot 10^{-17}$. However, this model is very strongly overfitted, so its usability is limited (see Subsection 2.1.2 for an explanation of overfitting).

The neural network performed reasonably well at predicting Data Set 2. The root mean square error during one fold of the k-fold cross validation is displayed in Figure 4.10. Both the training and validation error improve in about the first 70 iterations, but after that the validation error is stable while the training error continues to approach 0. The period when the validation error no longer improves is caused by the network overfitting the model to the training data. In order to avoid using an overfitted model, the state of the network with the lowest validation error is saved for every fold, and the average is calculated at the end.

The final network had an average root mean square error of 0.077. However, the root mean square error for each output dimension varied, ranging from 0.058 to 0.108. This variation indicates that there are some KPIs that are easy to predict and some that the model is not able to predict as accurately.

The random forest performed similarly, with an average root mean square error of 0.088. There was a large variation between the root mean square error for individual outputs, with the errors ranging from 0.052 to 0.169. An example of one tree in the random forest is shown in Figure 4.11.

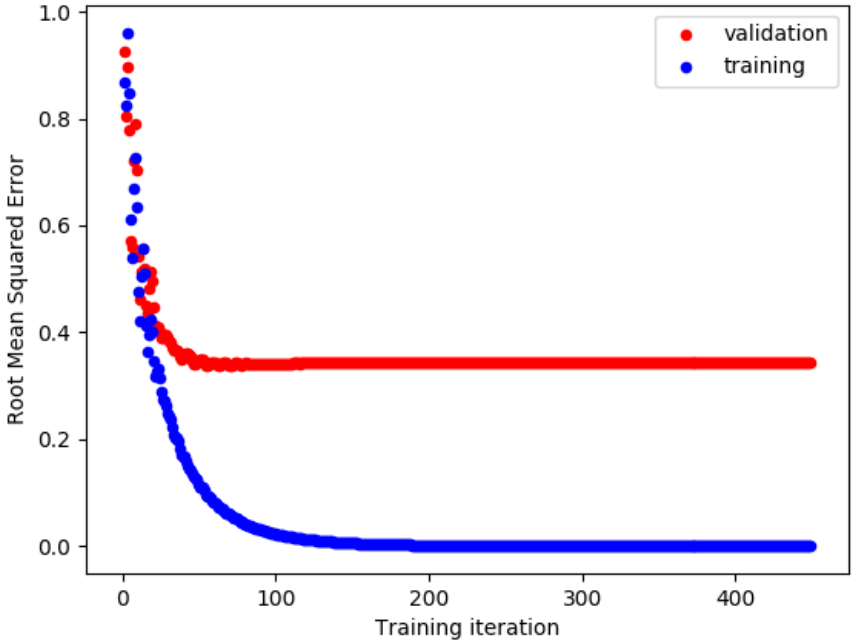

Figure 4.10: Root mean square error for the training of the neural network. In approximately the first 70 iterations, both the training error and the validation error decrease, but after that the validation error no longer improves.

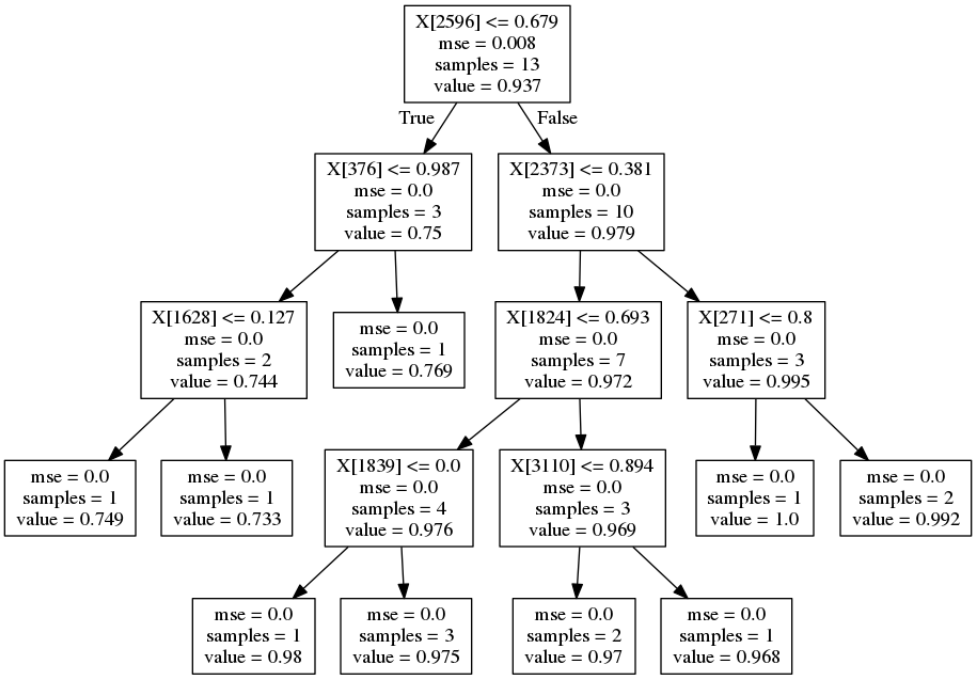

Figure 4.11: An example of a tree in the random forest. The diagram shows how the data is split up throughout the tree based on the value of certain input dimensions.

Analyzing the data sets and creating the regressors and classifiers demonstrated that there was a strong connection between the results of the executions of different test cases. The machine intelligence implementation therefore appeared better at utilizing the large amount of data to find relationships between test cases than the human testers. However, the goal was not to replace executions with the prototype's predictions, as this would require greater confidence in the completeness of the model and therefore a more thorough data set. Since testers will continue to only analyze completed test executions, these predictions were not immediately useful in improving the effectiveness of test analysis, which then became the focus of the second iteration.

4.3.2 Second Iteration

The expansions of the prototype created in the second iteration were evaluated based on their ability to identify error reports and testers' feedback.

4.3.2.1 Error Report Identification

The ability of the prototype to identify relevant error reports from the input test cases was tested using both the neural network and the linear regression model. As described in Section 4.2.2, the testing was conducted by looking at previous error reports that contained multiple test cases and seeing if the report would be included in the suggestions if the script was run for each of the test cases in the report. The test included 29 runs of our script. Of those 29, 21 reports were correctly identified, and 8 were missed. Upon examining the trouble reports for the 8 failed runs, only one of those reports was related to a degradation in the KPIs. This means that only one of the failures could be attributed to the network's knowledge of the relationships between KPIs.

If the linear regression model was used instead of the neural network, 17 of the 29 reports were correctly identified. Although the linear regression model had very accurate predictions of the KPI values, the relationships in that model did not match reality when looking at what test cases appear in the same error reports. This indicates that the linear regression model is overfitted, which explains how it was able to obtain such a low RMSE in Section 4.3.1.4. The difference in error report identification accuracy demonstrates the neural network's superiority to the linear regession model in learning general relationships instead of the specifics of the data.

The result from the error report testing also shows a flaw in this application, since only reports that are concerned with the KPIs can be identified correctly. From the 8 reports that the neural network was unable to identify, 7 were missed because the reports were about something unrelated to the KPIs but connected to the test cases. These reports were not about problems in the performance of the system that could be numerically measured, but were instead about errors in the testing itself or parts of the system that fail for some reason. Such errors will not be reflected in the KPI values. Although the 21 correctly identified error reports were not closely examined, it is clear that some of these also were about issues that did not directly affect the KPI values. The network identified these based on the fact that the test cases in the error report had related KPIs, even though the error

was not in the KPI values. One conclusion that can be drawn from this analysis is that finding related error reports based on relationships between KPIs will not be sufficient in order to identify all errors that affect multiple test cases.

4.3.2.2 Feedback

The prototype appeared very promising to the testers who tried it. However, the gathering of feedback from the testers was limited because a big update was made to the system being tested (see Section 5.1), which prevented the testers from being able to use the prototype when analyzing new issues. We were able to gather the following observations before this change occurred.

When the prototype was first used by a tester to examine a current degradation, an error report was identified, from an output similar to Figure 4.2, that the tester had not found using the normal search tool. The tester was able to confirm that the report was about the same problem that they had observed. The tester then proceeded to add their test case to the report, saving them from having to create a new report about their issue.

Another tester found a degradation of a KPI that historically had not seen any change. The network could not predict this result, and the output from the first application was not useful. From the output that corresponds to Figure 4.3, one could clearly see that the degradation was an extreme outlier. From this, we concluded that the two scripts work best together, with the first one providing good results when the changes that occur are similar to previous ones, while the second one points out the KPIs that are dissimilar to historical data.

4.3.3 Summary

For classification, perfectly accurate predictions could be made with both the logistic regression model and the neural network, while the random forest was not quite as accurate. Any of these methods is usable for predicting test results in performance regression testing. For regression, both neural networks and random forests were able to accurately predict KPI values, but their average root mean square error was much higher when compared to the linear regression model's error.

The design science method meant that our approach evolved during the course of the thesis. In the first cycle, the focus was to make accurate predictions of test execution data. However, these predictions were not inherently useful to the tester, who already has the actual test execution logs to examine. Therefore, the ability of the prototype to identify error reports and the testers' opinions were measured in the second iteration. When the neural network and linear regression were used in the application that identifies related error reports, the neural network was able to identify more of the error reports. When working with limited data, the error of the machine learning methods may be relatively high, but their ability to avoid overfitting when dealing with noise in the data increases their usability compared to conventional statistical methods.

Although feedback was limited, the testers who did use the prototype responded positively. The prototype was deemed to be very promising in dealing with the problems of **communication issues** and knowledge gaps surrounding fault reporting,

while reducing the focus on **requirements in the test oracle instead of degradations**. The issue of testers **re-running** test cases due to the non-deterministic nature of the system was also alleviated.

5

Discussion

System testing is a very time-consuming step of the development process that, at its essence, consists of analyzing the huge data set of test results. There are many possible ways to make this task easier and faster. Our prototype focused on finding and presenting important information in the data, delivering this information faster than current tools and manual analysis. Every aspect of the data can be of use when doing the root cause analysis; the history, relationships, and outliers both in value and in changes in the the value can all give indicators of when, how and why something had changed.

Communication with testers ensured that the prototype matched what the testers were currently missing. This meant both extracting useful information about a single test case and aiding communication by sharing information across test cases. Presenting as much information as possible in an understandable and informative way while also not becoming overwhelming is at the core of improving system test analysis. The current test oracle classifies an execution into meets requirements or not, but a more detailed automated data handling and analysis would ease the workload of the testers. We believe that with our tool, the testers will be able to get a better overview of the task at hand and of the previous work done with similar problems.

The results of this thesis was evaluated on three levels: the accuracy of the prototype's predictions, the accuracy of the prototype in identifying relevant error reports, and the testers' experiences using the prototype. All of these levels demonstrate the reliability of the prototype, and the last two establish usability as well. The numerical results were presented in the previous chapter and were very good. Although the feedback gathered from testers was limited, the reactions from those who used the prototype were positive.

In general, it is important to note that the correctness of the machine learning models is highly dependent on the data provided. In order for the model to be as accurate as possible, the data set should contain representatives of all possible scenarios. If the data contains a very small amount of big outliers, those outliers will be very difficult to predict accurately. On the other hand, for data sets with no outliers, the predictions can be very precise. As long as the data is not outdated, increasing the amount of data available for training the model will typically improve its accuracy. The threshold for what data counts as outdated can be hard to define and arbitrary, and especially for projects that follow Continuous Integration, obtaining enough data about the current state of the system can be difficult.

5.1 Challenges

When analyzing data, a large portion of time was spent on pre-processing. This is something we were warned about at the start of the thesis and also something we experienced firsthand. Our work in pre-processing data is something that the case company can use for any data analysis of this system, and the recommendations that follow about handling data are generally applicable and can save many hours of work for anyone interested in analyzing data collected over time.

One of the biggest challenges in the study was the incomplete nature of the data. Even after pre-processing the data, around 30% of the data was still missing. This decreases the accuracy and reliability of the model created using machine learning. An important lesson to practitioners that collect and store a large amount of data is to have consistency and thoroughness in the data collection process. This includes ensuring that the data collected is comparable to previously collected data and maintaining a consistent schedule of storing the data. Improving data handling increases the usability of the data and the potential for machine intelligence to be used in automation.

Another challenge with the data was finding comparable logs to be used for training. We decided that all logs used to create a single data point had to be from tests executed on the same build, which is part of the reason for the missing data discussed above. As new builds are constantly created at the case company, test executions that are only a few hours apart could be considered separate data points. This can be problematic if, for example, half of the test cases are run on one build, while the other half is run on the next build, creating two different data points that both have 50% of the data missing. However, the decision ensured that all test logs used to create a data point were from executions run on the same system. Even if builds are close in time, each build contains new updates that may affect the performance tests, meaning that each build needs to be assigned a distinct data point to ensure that the associations created during the machine learning reflect reality.

Four months after the start of the thesis, there was a major update to the the system tested by the case department. Although the test cases, KPIs, and checkpoints used to test this new version of the system were essentially the same, the relationships created from the data from the previous version of the system may not hold for the new state of the system. The only change required to the prototype is to change the path to the data to the new directories, but the prototype will not be usable on the new system until several months of data has been gathered. The data requirement is an inherent limitation of machine learning, and combined with the relatively infrequent complete testing of the system, causes this major drawback to this approach. On the other hand, the manual approach to system testing may either err on the side of assuming too much is unchanged or simply require more effort, as all performance changes are new types of changes, which may be unavoidable when testing a new product.

5.2 Threats to Validity

The thesis's approach to improving test analysis evolved during the course of the study because of the exploratory nature of the work. We started with the goal to use the test execution data collected by the case department and were confident that machine learning could provide benefits to the testers, but neither we nor our supervisors were certain of how to utilize the data in the best way. This approach allowed us to be more flexible in addressing the needs of the testers at the cost of having a more systematic study.

The prototype created in this study was based on the needs of a case company, and the accuracy of the prototype was then evaluated based on data provided by the case company. Although the development of the prototype aimed to be as general as possible, the results of the study may not be fully applicable to other use cases. In particular, there is no guarantee that this implementation of machine learning methods will be able to make accurate predictions for other data sets.

Ethically, there is a limit to the information we are able to provide about the study. In order to respect the needs of the case company, we cannot share the data we used or the source code of our prototype. We have also changed the outputs of our prototype included in this report to be more vague, removing references to specific test cases, KPIs, and builds. This is an obvious limitation on the reproducibility of the study. However, we maintain that the general conclusions of this study remain valid for other data sets, making the exact nature of our data less important.

Two master students worked together and validated the results of the study. A literature review was conducted, but the focus of the study was on the design and development of the prototype. The prototype was further evaluated by the industry supervisors and presented to testers at the case department. Feedback from the testers was continually collected as they worked with the prototype.

5.3 Contribution to Academic Research

This thesis adds to research into performance regression testing and specifically contributes to knowledge within the automation and implementation of machine intelligence in such testing, which is an area lacking research. Although the prototype created does not itself deliver a verdict, it does take into account a set of test runs to provide its analysis, as recommended by Hierons [16]. As Briand [5] stated, machine learning in general and this prototype in particular are well suited for systems that change over time. As seen when the major update was introduced at the case department, the prototype could adapt to this change and did not need to be reworked. An important limitation noted in this thesis is that if the change is big enough so that previous data can no longer be considered applicable to describing the current system, then sufficient data must be gathered from the current system before machine intelligence can be utilized. Another important aspect to note is that if the system is constantly changing, then eventually old data needs to be discarded, as these executions come from a system too dissimilar to the current state.

The thesis is similar to the work done by Foo et al. [10], who created a system to automatically detect performance degradations in heterogeneous environments. Although the prototype was tested on a single system, its structure is independent of the type and source of the data, so our work should be able to function in heterogenous environments as well. As long as a new training data set is given, it should perform similarly.

5.4 Future Work

As the prototype is a complement to the current test oracle, it adds another place that the testers have to check in order to gather information about their test case. A major quality of life improvement would be to integrate this with the test oracle, creating an oracle that not only considers the stated requirements but also warns testers about anything that could be a sign of a change in the system's performance. In addition, an oracle like this would also incorporate the more holistic view of testing, pointing out the test cases that have similar warnings or errors.

Although the classifiers were of limited use in our study, they could be useful to other parts of the case company and software testing in general. Accurate classifiers could, for example, be used in deciding what and in what order to execute tests. This was outside the scope of the thesis, but automatic test prioritization has previously been studied at the case company and could be applied to the case department as well [17]. Since there are several different frequencies on which tests are run and analyzed, optimizing which tests are placed into each frequency would lead to a faster detection of degradations and reduce the amount of testing systems required. In addition, it would also be possible to consider a fluid ranking of test execution, where requirements on the frequency of execution are lessened or removed. Instead, tests would be run according to their highest failure probability.

6
Conclusion

The purpose of this study was to find ways to improve software test analysis using machine learning. Software testing is a costly but necessary part of development, so reducing the efforts required to conduct it would benefit industry greatly. Automation is one of the most promising ways to accomplish this and was therefore the focus of this study. Although fully automated testing is still unfeasible, we believe that our work can ease the time and effort required of the testers conducting performance regression testing at the case company.

The goals of the thesis were achieved by creating a prototype that shows testers general information about the KPI values in their test case as well as the machine learning model's prediction of related error reports to the current test case. The predictions about test cases' KPIs used to create these recommendations was found to be accurate, as long as the current execution did not deviate too far from the historical results.

Bibliography

[1] *About Ericsson - Corporate Information*. 2018. URL: https://www.ericsson.com/en/about-us.

[2] Michael A Babyak. "What you see may not be what you get: a brief, nontechnical introduction to overfitting in regression-type models". In: *Psychosomatic medicine* 66.3 (2004), pp. 411–421.

[3] E. T. Barr et al. "The Oracle Problem in Software Testing: A Survey". In: *IEEE Transactions on Software Engineering* 41.5 (May 2015), pp. 507–525. ISSN: 0098-5589. DOI: 10.1109/TSE.2014.2372785.

[4] Sabri Boughorbel, Fethi Jarray, and Mohammed El-Anbari. "Optimal classifier for imbalanced data using Matthews Correlation Coefficient metric". In: *PLOS ONE* 12.6 (June 2017), pp. 1–17. DOI: 10.1371/journal.pone.0177678. URL: https://doi.org/10.1371/journal.pone.0177678.

[5] L. C. Briand. "Novel Applications of Machine Learning in Software Testing". In: *2008 The Eighth International Conference on Quality Software*. Aug. 2008, pp. 3–10. DOI: 10.1109/QSIC.2008.29.

[6] Tianfeng Chai and Roland R Draxler. "Root mean square error (RMSE) or mean absolute error (MAE)?–Arguments against avoiding RMSE in the literature". In: *Geoscientific model development* 7.3 (2014), pp. 1247–1250.

[7] Emelie Engström, Robert Feldt, and Richard Torkar. "Indirect Effects in Evidential Assessment: A Case Study on Regression Test Technology Adoption". In: *Proceedings of the 2Nd International Workshop on Evidential Assessment of Software Technologies*. EAST '12. Lund, Sweden: ACM, 2012, pp. 15–20. ISBN: 978-1-4503-1509-8. DOI: 10.1145/2372233.2372239.

[8] Mikael Fagerström and Emre Emir Ismail. "Automated Performance Regression Analysis: An Industrial Case Study". 59. MA thesis. Department of Computer Science and Engineering, Chalmers University of Technology, 2016. URL: http://studentarbeten.chalmers.se/publication/232235-automated-performance-regression-analysis-an-industrial-case-study.

[9] Mikael Fagerström et al. "Verdict Machinery: On the Need to Automatically Make Sense of Test Results". In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ISSTA 2016. Saarbrücken, Germany: ACM, 2016, pp. 225–234. ISBN: 978-1-4503-4390-9. DOI: 10.1145/2931037.2931064.

[10] K. C. Foo et al. "An Industrial Case Study on the Automated Detection of Performance Regressions in Heterogeneous Environments". In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 2. May 2015, pp. 159–168. DOI: 10.1109/ICSE.2015.144.

[11] Martin Fowler and Matthew Foemmel. "Continuous integration". In: *ThoughtWorks* 122 (2006), p. 14.

[12] Ahmed S Ghiduk, Moheb R Girgis, and Eman H Abd-Elkawy. "A survey of regression testing techniques". In: *International Journal of Advanced Research in Computer Science* 3.5 (2012).

[13] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press, 2016.

[14] Christoph Heger, Jens Happe, and Roozbeh Farahbod. "Automated Root Cause Isolation of Performance Regressions During Software Development". In: *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*. ICPE '13. Prague, Czech Republic: ACM, 2013, pp. 27–38. ISBN: 978-1-4503-1636-1. DOI: `10.1145/2479871.2479879`.

[15] Alan Hevner et al. "Design Science in Information Systems Research". In: *Management Information Systems Quarterly* 28 (Mar. 2004), pp. 75–106.

[16] Robert M. Hierons. "Verdict Functions in Testing with a Fault Domain or Test Hypotheses". In: *ACM Trans. Softw. Eng. Methodol.* 18.4 (July 2009), 14:1–14:19. ISSN: 1049-331X. DOI: `10.1145/1538942.1538944`.

[17] Jiayu Hu and Yiqun Li. "Implementation and Evaluation of Automatic Prioritization for Continuous Integration Test Cases". 49. MA thesis. 2016.

[18] A. Ieshin, M. Gerenko, and V. Dmitriev. "Test automation: Flexible way". In: *2009 5th Central and Eastern European Software Engineering Conference in Russia (CEE-SECR)*. Oct. 2009, pp. 249–252. DOI: `10.1109/CEE-SECR.2009.5501151`.

[19] D. Lee, S. K. Cha, and A. H. Lee. "A Performance Anomaly Detection and Analysis Framework for DBMS Development". In: *IEEE Transactions on Knowledge and Data Engineering* 24.8 (Aug. 2012), pp. 1345–1360. ISSN: 1041-4347. DOI: `10.1109/TKDE.2011.88`.

[20] Andy Liaw, Matthew Wiener, et al. "Classification and regression by randomForest". In: *R news* 2.3 (2002), pp. 18–22.

[21] Lu Luo. "Software testing techniques". In: *Institute for software research international Carnegie mellon university Pittsburgh, PA* 15232.1-19 (2001), p. 19.

[22] Thanh H. D. Nguyen et al. "An Industrial Case Study of Automatically Identifying Performance Regression-causes". In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. MSR 2014. Hyderabad, India: ACM, 2014, pp. 232–241. ISBN: 978-1-4503-2863-0. DOI: `10.1145/2597073.2597092`.

[23] Jiantao Pan. "Software testing". In: *Dependable Embedded Systems* 5 (1999), p. 2006.

[24] James J Rooney and Lee N Vanden Heuvel. "Root cause analysis for beginners". In: *Quality progress* 37.7 (2004), pp. 45–56.

[25] Raul Rosero Miranda, Omar S. Gómez, and Glen Rodriguez. "15 Years of Software Regression Testing Techniques – A Survey". In: *International Journal of Software Engineering and Knowledge Engineering* 26 (July 2016), pp. 675–689.

[26] James A Whittaker. "What is software testing? And why is it so hard?" In: *IEEE software* 17.1 (2000), pp. 70–79.