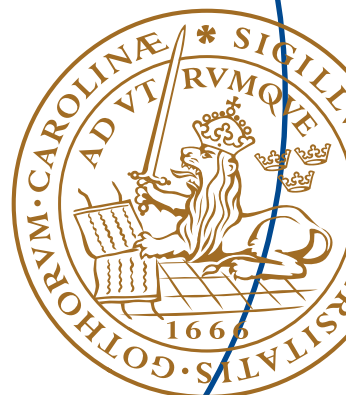


Master's Thesis

Navigating Information Overload Caused by Automated Testing

Vanja Tufvesson
Nicklas Erman



Navigating Information Overload Caused by
Automated Testing
- A Case Study at Qlik

Tufvesson, Vanja
vanjakarin@gmail.com

Erman, Nicklas
nicklas.erman@gmail.com

Department of Electrical and Information Technology
Lund University

Advisor: Markus Borg

June 5, 2014

Printed in Sweden
E-huset, Lund, 2014

Abstract

The growing demand of high quality software creates an increasing pressure on software development organizations to increase test coverage in order to meet the quality requirements. At the software company Qlik, automated testing helps improve test coverage and allows development of multiple features in parallel while maintaining a stable code base. In order to benefit from the automated testing, the results must be processed and analysed. This is currently done by an analyst reading log files, interpreting error messages and looking at screenshot images. Automated tests run every night for up to twenty development branches, each containing thousands of test cases - resulting in information overload. It is extremely difficult and time consuming for a human to process the test results and get an overview of the state of the development. Qlik is in desperate need of an automated analysis approach. In this thesis we create NIOCAT, a tool that automatically analyses test results. The output is an overview of all failed test cases, where similar failures have been grouped together. To evaluate NIOCAT, experiments on manually created subsets representing different use cases are conducted. To further enhance the evaluation methodology a focus group meeting is held with test result analyst experts at Qlik. The experiments conducted in this case study show that NIOCAT can create accurate clusters, in line with analyses performed by humans. Further, the need and potential time-savings of our approach is confirmed by the participants in the focus group. NIOCAT thus provides a feasible complement to current automated testing practices at Qlik. Future work includes deployment and calibration of the tool within the context of the company as well as adding new desirable features discovered during the focus group session.

Table of Contents

1	Introduction	1
1.1	Background	1
1.2	Proposed Solution	2
1.3	Research Questions	2
1.4	Outline of the Report	2
1.5	Contribution Statement	3
1.6	Acknowledgements	3
2	Related Work	5
3	Background	9
3.1	Software Testing	9
3.2	Information Retrieval	12
3.3	Application	15
4	Case Description	17
4.1	Qlik	17
4.2	Software Configuration Management at Qlik	18
4.3	Automated Test Development at Qlik	19
4.4	Tools for Automated Testing at Qlik	20
4.5	Test Results Artifacts	21
4.6	Analysis of Test Results	22
4.7	Bug Report Creation Tool	25
4.8	Case Summary	27
5	Methodology	29
5.1	Solution Idea	29
5.2	Selection of Components of a Test Case Failure	30
5.3	Data Collection	31
5.4	Implementation	32
5.5	Evaluation Based on Accuracy of the Output	33
5.6	Evaluation Based on Qualitative Feedback	36
5.7	Visualization in QlikView	37

6	Results	41
6.1	Similarity Between NIOCAT and Manual Analysis	41
6.2	Feedback from the Focus Group	42
7	Discussion	49
7.1	Threats to Validity	49
7.2	Discussion of Results	49
7.3	Future Work	51
8	Conclusion	53
	References	55

List of Figures

3.1	Levels of testing. Adopted from Collard et al, Practical Software Testing [11].	10
3.2	Vector Space Model.	13
4.1	Branch A passed all tests and is allowed to deliver its changes to the main branch. Branch B still has failing tests and is therefore not allowed to deliver.	18
4.2	An illustration of the overlapping tests runs between branches.	19
4.3	Horsie provides an integration between test specification and test execution within QlikView	20
4.4	Excerpt of a log file produced by the Horsie framework.	21
4.5	An excerpt of the overview showing test results of the latest test run from each branch.	23
4.6	The same code base tested nine times yields different results due to intermittent failures	24
4.7	Automated bug report creation tool. If a new error occurs a new bug report is created. If a bug report already exists for a certain error message the tool will attach data about the new failure to the existing bug report.	26
5.1	NIOCAT outputs a partition of failures based on the test results input	30
5.2	Each small filled circle in the partition is a failed test case. The test cases are grouped by similarity. Each group is referred to as a problem.	31
5.3	Illustration of the manual partition of reference set 1.	35
5.4	Illustration of the manual partition used for reference set 2.	38
5.5	Illustration of the manual partition used for reference set 3.	38
5.6	Two different configurations in the failure component configuration space.	39
5.7	An output of NIOCAT visualised in QlikView	39
6.1	The weight configurations for subset 1 that yielded the highest adjusted rand index.	45
6.2	The weight configurations for subset 2 that yielded the highest adjusted rand index.	46

6.3	The weight configurations for subset 3 that yeilded the highest ad-justed rand index.	47
-----	---	----

List of Tables

5.1	Properties of the three different data subsets used for reference . . .	34
6.1	Adjusted rand index for clusterings of each subset and clustering approach	42

1.1 Background

Software systems play an important role in today's society [11]. The pressure for the delivery of new functionality and applications has never been stronger [17]. A software failure can result in huge losses, financially and socially [11]. The growing demand on high quality software creates an increasing pressure on software development organizations to increase test coverage to meet the quality requirements. At the same time, increasing competition between development organisations has lead to the pressure of meeting shorter deadlines with static or even declining resources [17].

The software company Qlik has adopted automated testing in order to save time, improve test coverage and enable development of new features in parallel while assuring a high quality product. At Qlik automated tests run every night on multiple source code branches.

While automated testing provides the benefits of reducing manual testing, minimizing human error, and enabling a higher testing frequency [11, p. 466], new challenges are introduced. With higher testing frequency the volume of test results also increases [1]. Automated test environments produce logs to help identify the cause of failing test cases [28]. This also contributes to an increase in the results quantity. At Qlik it has proven to be both difficult and time consuming to manually analyse the volume of test results. To help overcome the information overload, Qlik has developed an automated bug report generation tool, that clusters failures based on exact textual similarity of certain test case features. Although this tool has provided a starting point for further analysis there are still several challenges with the analysis of test results.

There is currently no way to get an overview of which branches that failed the same test case. Intermittent failures due to the testing environment make the results unreliable unless the same tests are run multiple times, resulting in more time spent on testing. Another challenge is the identification of a unique problem. Two separate tests failing due to the same root cause should ideally be grouped together. Similarly should a test case that failed in different steps of the test case not be considered the same problem. A code base that is constantly growing and a constantly increasing number of test cases, result in large quantities of test results produced each day. The situation for the test result analyst could be characterized

as information overload. The challenges with the analysis become impossible to overcome manually and an automated approach is needed.

1.2 Proposed Solution

An automated analysis approach to analysing test results could potentially save a lot of time [28]. At Qlik there is a demand for an intelligent analysis tool that could process the large amount of data, cluster failures which are similar, and separate test cases which fail differently on different branches. Being able to cross reference what problems exist on which branches could decrease the time spent on localizing the origins of a certain bug. Having the ability to easily compare problems that are occurring in different branches could also help identifying intermittent failures, without having to rerun the tests. Our goal with this thesis is to create a tool of this kind that can simplify the analysis of results from automated testing. The tool should be able to navigate test results from one or multiple branches over varying time spans and by using information retrieval techniques be able to group similar failures together.

1.3 Research Questions

The background and challenges described in this chapter has lead us to use the following questions as a basis for our master thesis.

- How can automated analysis of test results help Qlik with the manual navigation of the information overload caused by automated testing?
- How can information retrieval techniques be incorporated into the automated analysis to simplify the process of identifying different problems found in test results?
- Can consideration of a combination of execution data and textual information improve the accuracy of the test failure clustering compared to a clustering based on textual information alone?

1.4 Outline of the Report

The Background chapter contains a theoretical background of all the different concepts that this thesis is based on, mainly automated testing, information retrieval and clustering. Next chapter is a description of the case we have been working on, i.e. the situation for a test results analyst as Qlik. In this chapter we dig deeper into the challenges that the analyst is facing. Following the Case Description is the Methodology chapter, where we outline our solution idea, the clustering algorithm and our methods of evaluation. The results are presented in the following chapter and further discussed in the Discussion chapter. The report ends with our conclusions and future work.

1.5 Contribution Statement

Throughout the whole process we have worked closely together. All implementation design decision were made together even though we divided the implementation and report writing work load.

Nicklas Erman worked with the test results data collection. He conducted analysis around the build scheduler API and Qlik's automatic bug report creation tool in order to implemented methods for calling the build scheduler API, download and save the test results artifacts.

Vanja Tufvesson implemented methods that processed the downloaded artifacts, created the data structures for the failure data and saved the data in memory. Vanja Tufvesson also implemented methods for processing of the log files in order to extract test case names and error messages, while Nicklas Erman implemented the HTML extraction.

A program for calculation of the adjusted rand index was implemented by Nicklas Erman, while Vanja Tufvesson worked with visualisation of the data. The work with visualisation included writing the output to an excel file, loading the file into QlikView and creating QlikView applications based on the data.

As to the report writing, we have written most sections together. The structure, all the figures and tables have been designed together. Nicklas Erman has been responsible for creating the figures and Vanja Tufvesson has been responsible for the creating the tables and formulating the mathematical equations.

1.6 Acknowledgements

We are grateful to our supervisor at Qlik, Lars Andersson for constantly providing us with the industry perspective on our research and his technical expertise. Lars has motivated us along the way by explaining how our tool could help him and his colleagues at Qlik.

We would also like to thank our advisor Markus Borg for all the help and feedback throughout the project. Markus continually reading and giving constructive feedback on the report and our methods have been invaluable to us.

The help of our examiners Per Runeson and Anders Ardö is also much appreciated. We are grateful to our examiner Per Runeson for expressing his interest in the project and understanding the complexity of the study. We would like to thank Anders Ardö for providing us with clear directives on what is expected from us and pointing out potential risks with the project.

Related Work

Our work is related to information overload within software engineering projects. In the first part of this chapter we present previous studies that have addressed challenges with information overload in software engineering projects and proposed ways to manage the challenges. Also, our proposed solution is related to previous work on duplicate detection of bug reports which is also presented in this chapter. We conclude this chapter by describing our work in the light of previous studies.

Robillard et al. suggest using recommendation systems to navigate information overload in the context of software development in their article Recommendation Systems for Software Engineering [24]. The authors explain that information overload is a problem within software engineering. Software engineers need to have knowledge about many large code libraries and the dependencies between them. To manage large quantities of information the authors present different recommendation systems for software developers. The tools may assist with recommending which files to edit in order to accomplish a certain task or to reveal patterns for how to work with certain objects.

A case study spanning six companies was conducted by Bjarnason et al. in order to investigate challenges and practices in aligning requirements with verification and validation [10]. The authors express the importance of testers knowing when and how a requirement changes so that the testing may be aligned with the updated requirement specification. As a consequence of updated requirements old test cases may need to be deleted or updated as well as new test cases introduced. The alignment process between requirements and verification and validation is thus highly dependant on communication and traceability practices. The study also identifies one of the challenges with the alignment process as managing large quantities of requirements and test cases.

Feldt performed a study to investigate if test cases in automated testing suites grow old [15]. The study compared test case failures with hardware component failures. It is known that hardware component failures are more frequent early in their lifetime, components that survive their infancy will most likely not fail until age begins to deteriorate the component. The paper continues to search for a similar relationship for test cases. The author shows that test case failure frequency is higher in the infancy time of the test case and slowly decreases as the software matures. According to the study test cases do not seem to wear out in the same manner as a hardware component would. As long as the test case remains in use over time the amount of failures caused by the test case will decrease to

zero and remain stable. The author points out that as a system keeps growing and evolving, the number of test cases also typically increases since new test cases are added.

Previous research has shown that it is possible to identify duplicate bug reports in a bug tracking system using Information Retrieval techniques on the natural language text in the report. Runeson et al. developed and evaluated a prototype tool during a case study at Sony Ericsson [25]. Using the Vector Space Model on the textual content in the reports the authors were able to find up to 2/3 of the duplicates. The tool ranked the most relevant duplicate report candidates and presented a list of them to the user. Lists of size 5, 10 and 15 were used in the evaluation. The evaluation was based on a measurement called recall, i.e. the number of duplicate bug reports whose target reports are in the suggested list divided by the total number of duplicates in the experiment. The evaluation showed a recall of 40% for a list of size 10.

Multiple projects have addressed the problem of duplicate bug reports by introducing recommendation tools that let the bug reporter view a list of existing reports similar to the new one, so that the new bug can be discarded if it is a duplicate of an existing one. In order to enhance the quality of the list, Wang et al. added consideration of the execution information of runs that cause a bug to be reported [29]. The authors used a machine learning approach that was firstly calibrated using the Eclipse project. It was then evaluated using a subset of the Firefox bug repository and compared with approaches only using textual similarity. The recall for lists of size 1-10 ranged 67% – 93%, compared to 43% – 73% using natural language data alone.

Lerch et al. have addressed in a study that it can be frustrating for reporters to compose a whole bug description and later discard it anyway [20]. The authors present an approach based on stack traces alone, which does not require the user to write a whole new bug description unless it is determined that the bug is not a duplicate. To assess the performance of the duplicate detection it was compared to a baseline text-based implementation. The authors observed that the baseline performed much better on reports containing stack traces, which corresponds to what was shown by Wang et al. earlier. The recall was 30% – 50% higher. Duplicate detection based on stack traces alone had slightly better performance than the baseline approach. The authors argue that an approach like this would require much less effort from the reporting user though. Experiments on what part of the stack trace are most important for duplicate detection showed that method calls are the most effective. This approach differs from most previous work since it does not require the user to type a whole bug description before it can provide duplicate suggestions.

The study by Bjarnason et al. indicates that information overload is a challenge within software engineering and Robilliard et al. suggests using recommendation systems to manage the challenge. As far as we know, there has not been any previous research done on managing large amounts of automated test results using information retrieval techniques. Our project is inspired by the ability to use information retrieval techniques to find duplicate bug reports, i.e. to cluster similar software failures together. The work done by Wang et al. and Lerch et al. showed that execution data combined with natural language outperforms the

natural language approach. We chose to focus only on failures found through automated testing, which data presentation have slightly different characteristics compared to the bug reports. The text is machine generated and not written by a human reporter. However, this thesis will investigate further if execution data, in this case an HTML snippet, can be incorporated into the analysis in order to improve the accuracy of the clustering of software issues, compared to focusing on textual information alone.

This chapter contains two sections, Software Testing and Information Retrieval. The first section begins with a definition of testing and how testing needs to be done in an Agile development environment. The concept of Automated Testing is then introduced and its two main parts, driving the software and validating the results, are described. The second section introduces the problems with Information Overload and some Information Retrieval techniques. The Vector Space Model is presented, followed by the concept of Clustering. The section ends with a presentation of measurements of similarity between different clusterings.

3.1 Software Testing

To ensure that a software product behaves as it is expected to, software testing is needed.

Testing

Testing is the process of exercising a software component using a selected set of test cases, with the intent of (i) revealing defects, and (ii) evaluating quality [11].

Software testing can generally be divided into different levels, or phases, as shown in Figure 3.1. At each level there are specific goals. At unit test, single components are tested separately. The focus at this level is on functional and structural defects. At integration, the units are assembled and tested together as a group. Here the goal is to investigate interactions between different components. The system as a whole is tested at the System level, where reliability and performance are important attributes under investigation. At the Acceptance Test level the software is tested against the requirements. This can often involve actual users of the system and/or clients' approval [11].

3.1.1 Test Execution

The usual approach of testing a piece of software is to execute a Test Case.

Test Case

A set of test inputs, execution conditions, and expected results developed

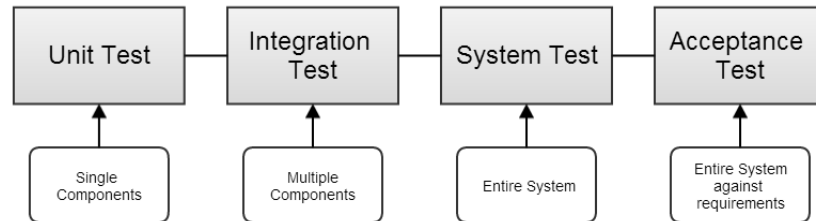


Figure 3.1: Levels of testing. Adopted from Collard et al, Practical Software Testing [11].

for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement [6].

In a practical matter, a test case needs to contain three pieces of information. The first is a set of inputs. This is data received from an external source, like hardware, software or a human, during the test. The next is the execution conditions required for the test to be able to run, for example a certain state of a database. The last information is the expected outputs, which should be compared to the results produced during the test. Related test cases are usually run together and can be referred to as a test suite [11].

3.1.2 Testing in an Agile Environment

The Agile software development methodology is based on iterative and incremental development, where the customer is involved all the way through the process. There are several approaches that could be considered Agile, but they all have iterative development cycles with significant testing and customer involvement. In Agile testing the key is to involve the customer in an early stage of the development cycle. As soon as there is a stable code base, customers should begin testing and the developers should be given feedback. This means that testing is not a separate phase of the development life cycle, instead it is integrated along the way to ensure progress and customer satisfaction. Since the development cycles are short, timely feedback is required [23]. Manual testing is time-consuming and labour-intensive, especially if done thoroughly and repetitively [17]. For a large application there might be thousands of unit tests. But even for the smallest application testing can be a daunting task [23].

3.1.3 Automated Testing

Adopting Automated Testing is a way to reduce testing time and save money [17].

Automated Testing

Automated Testing is the use of special software (separated from the software being tested) to control the execution of the tests and the comparison of actual outcomes to predicted outcomes [19].

Automated testing can help increasing the test coverage so that errors can be detected before they have a chance to do real damage in production. The tests are repeatable and reusable and thus helps save time testing [17]. As suggested by Chris Dickens at Microsoft Office Test, Automated Testing can be divided into two big parts: driving the software and validating the results [12].

The concept of driving the software is basically about making the test run the software in the intended way. For example, in order to test that a button works correctly, the test needs to press the button. This could be done in few different ways. The button click event handler code typically calls an API upon a button click in the software. The test could either call the API directly or it can override the system and move the mouse to the corresponding screen coordinates of the button and send a click event, so that the event handler code gets executed. Calling the API might be the easiest approach to implement, while the drawback is that the user interface (UI) testing needs to be done manually. Simulating a mouse click requires that the coordinates of the button are known or can be calculated [12].

Every automated test script needs a verification step, where the test outputs are compared to the expected outputs. If the produced and expected outputs match, the test case passes. The mechanism for determining if a test case has passed or failed is called an oracle [11]. There are different approaches to do the verification. By assuming that some of the functionality is working as intended, the verification can be limited to parts of the functionality thus making tests easier to verify. This approach would ensure test coverage of part of the functionality but some of it would still need to be tested manually. Another approach is to do the full verification manually. This approach is suitable when it is complicated to construct an oracle. A big suite of tests can be complicated and time-consuming to set up. The manual approach enables taking advantage of the time saved in setting up and running the tests automatically, while time is not spent on implementing a meaningful oracle. Using a comparison tool as an oracle can help, but even an advanced comparison tool can fail a test because of a small difference in the output that a human would consider meaningless [12].

The use of automated test tools can reduce some of the work within the testing process. For instance, there are test tools that provide a driver module that runs the tests. Flow-analysis tools enumerate paths through a program, find statements that can never be executed (“unreachable” code) and identify places where a variable that has not been assigned a value is used [23].

3.1.4 Test Case Specification

Behaviour Driven Development

Behaviour Driven Development (BDD) is a specification technique that automatically certifies that all functional requirements are treated properly by source code, through the connection of the textual description of these requirements to automated tests [7].

BDD is based on Test Driven Development (TDD), which in turn is a development practice that implies the writing of test cases before the actual implementation of

a certain task [7]. BDD lets developers specify test cases in structured natural language in a Given-When-Then format as seen in the following listing.

Listing 3.1: Example of test case specified with behaviour driven development.

```
Scenario 1: E-mail application should let a user read a new e-mail
Given there is a new e-mail
When the user clicks on the new e-mail
Then the contents of the new e-mail should be displayed
```

The behaviour of an application is specified in a structured manner which enables source code to be attached to statements. Test cases specified this way lets less technical oriented stakeholders participate in the process of formulating test specifications and application behaviour.

3.2 Information Retrieval

The age of information technology has provided the means for almost anybody to distribute content to all of the world without much effort. Because of ease of distribution, finding things you are looking for on the Internet can be a challenging task. Because resources available on the Internet varies a lot, both in content and in quality, and because of the sheer amount of information available one could say the Internet is plagued by Information Overload [9].

Information Overload

Information Overload, a state where individuals do not have time or capacity to process all available information. [13]

Yet the problem with Information Overload on the Internet is not something that most people experience nowadays. The problem did not just go away, search engines based on techniques for navigating information overload emerged. The techniques used by search engines belong to a field of study called Information Retrieval. Thus reduction of information overload can be achieved by using Information Retrieval [4].

Information Retrieval

Information Retrieval deals with the representation, storage, organization of, and access to information items such as documents, web pages, online catalogues, structured and semi-structured records, multimedia objects. The representation and organization items should be such as to provide the users with easy access to information of their interest [8].

In this section we aim to describe some Information Retrieval concepts that will be used later in the report.

3.2.1 Textual Similarity Measure

A common way to measure similarity between objects is to use the vector space model. In the vector space model, similarity can be computed as the cosine sim-

ilarity between two vectors. This can be applied to different kind of objects, but one common type of object is text documents. The main requirement to be able to use the vector space model for a certain kind of objects is that a comparable vector representation can be created for a given object. When comparing text documents, the different terms within a document are usually considered the components of the vector. The size of an individual component can be determined by the frequency of the corresponding term within the document.

Once vectors have been established the similarity between two objects can be determined by calculating the cosine similarity. Cosine similarity between two objects can be computed with the following equation, where a and b are vector representations of two objects.

$$\cos(\theta) = \frac{\vec{a} \cdot \vec{b}}{\|\vec{a}\| \times \|\vec{b}\|} \quad (3.1)$$

The equation can be derived to a more practical form which is displayed in equation 3.2. The subscript i for vectors a and b represent the component index for the vectors respectively and N is the number of components.

$$\text{sim}(a, b) = \frac{\sum_{i=1}^N a_i b_i}{\sqrt{\sum_{i=1}^N a_i^2} \sqrt{\sum_{i=1}^N b_i^2}} \quad (3.2)$$

The cosine similarity is the cosine value of the angle between two vectors. Because cosine similarity is used in positive space the value ranges between zero and one where a value of one indicates that two vectors aligned and thus both objects include the same components [8]. Figure 3.2 illustrates two vectors, with two components each, but in reality more dimensions are usually used, typically one dimension for each term in the text. The cosine of the angle θ between the vectors can be calculated with the equation 3.1.

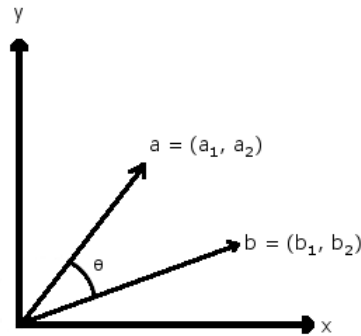


Figure 3.2: Vector Space Model.

3.2.2 Clustering

The concept of grouping similar items together is called clustering. A similarity measure between two items is typically computed using associations and similarities among different features of the items. In a text collection, features can be words or phrases. If there is a way to calculate a similarity between two items, it is possible to automatically create clusters of the items in the collection. Apart from the advantage of being unsupervised, clustering can potentially reveal trends in a group of documents, that would have been difficult to find otherwise. The disadvantages of clustering are that it can be hard to predict the form and the quality of the outcome and that it can be difficult to label the groups automatically [8]. For example, a clustering algorithm for restaurants, might manage to cluster restaurants that serve the same type of food, i.e. Italian, Chinese or fast food, without being able to name the food category. Although automatic labelling of the clusters might be a difficult problem, clustering has the potential to give insight to the data by revealing natural patterns that exist in the data [8]. While clustering refers to the grouping of similar items, the concept of breaking down a set of items to multiple non-overlapping subsets (or clusters) is called partitioning. The set of all clusters can thus be referred to as a partition [16].

3.2.3 Clustering Similarity Measurements

The rand index is a statistical method for determining the similarity between two different partitions of the same data set. More specifically it is a measurement of the fraction of pair wise agreements between two clusterings. The rand index between two partitions range between 0 and 1, with 1 indicating that the partitions are identical.

To compute the rand index each pair of data points are considered. A pair of data points can either be in the same cluster or not. When comparing a partition to a control set, each pair can be classified as either of the following classifications:

True Positive The pair of data points are correctly clustered together.

True Negative The pair of data points are correctly separated from each other.

False Positive The pair of data points are falsely clustered together.

False Negative The pair of data points are falsely separated from each other.

The rand index, RI , is then calculated using the equation

$$RI = \frac{tp + tn}{tp + tn + fp + fn} \quad (3.3)$$

where tp , tn , fp and fn are the number of pairs classified as true positives, true negatives, false positives and false negatives respectively. Thus, the rand index is a measurement of the fraction of correctly classified pairs of data points to all pairs of data points. [21].

The rand index is intuitive but has several known drawbacks, such as the fact that the index of two random partitions does not yield a constant value. It is also highly dependent on the number of clusters. A completely random clustering

could end up with a high index if it happens to have a large number of clusters [27]. The Adjusted Rand Index was proposed by Hubert and Arabie with the intention of overcoming these issues [18]. According to Milligan and Cooper, *ARI* is the best suited index to use for measuring similarity between partitions with different number of clusters [22].

As suggested by Santos and Embrechts, *ARI* can be calculated based on the variables within equation 3.3 for *RI* [27]. *ARI* can thus be computed with the following equation

$$ARI = \frac{ab - c}{a^2 - c}, \quad (3.4)$$

where a, b and c are defined as:

$$a = tp + fn + fp + tn, \quad (3.5)$$

$$b = tp + tn, \quad (3.6)$$

$$c = (tp + fn)(tp + fp) + (fp + tn)(fn + tn). \quad (3.7)$$

3.3 Application

In this thesis we develop a tool that helps navigating information overload caused by automated testing. The tool clusters similar test case failures together using a similarity measurement that is based on the vector space model. The output of the tool is evaluated using the adjusted rand index.

Case Description

We have conducted a case study at the research and development department at Qlik. The first section of this chapter contains a brief description of Qlik, the software it provides and its development project. The next section is about the configuration management of the development project, followed by an introduction of the tools used for automated testing and the process of analysing the test results, as of the time of writing this thesis.

4.1 Qlik

Qlik is a software company that delivers a product called QlikView, which is a business intelligence¹ solution. QlikView is used world wide for decision support at different functions within a wide range of industries, such as banking, life science, retail, and telecommunications. For example, banks can use QlikView to manage cost and risks and meet regulatory compliance while life science companies can use the product for improvement of physician feedback to manufacturers. The company has more than 31000 customers in 100 countries. QlikView helps visualizing data from multiple sources into an application so that users can analyze, search, explore associations and uncover trends within the data. Users can create and share apps and access QlikView from computers or a mobile devices [2].

QlikView has been under continuous development since the company was founded in Lund, Sweden in 1993. More functionality and features have been added and the complexity has grown. A few years ago, Qlik began clean slate development of the next major release of QlikView, a development project called QlikView.Next. One of the visions for QlikView.Next is “One client to rule them all”. All user interactions with QlikView.Next will be performed through a web browser. Previous versions of QlikView featured a wide array of different clients; Desktop client, Internet Explorer plugin client and the AJAX Browser client. Having many clients often resulted in inconsistent behaviour among different clients and higher complexity in the maintenance work. Standardizing on a single client should make it easier to develop QlikView apps and give users a consistent experience across all devices and platform [5]. Internally, the standardization has lead

¹Business Intelligence is a set of methodologies and techniques to transform data into useful information for strategical business decisions [14].

to the possibility to have a higher test coverage with one single automated test framework.

4.2 Software Configuration Management at Qlik

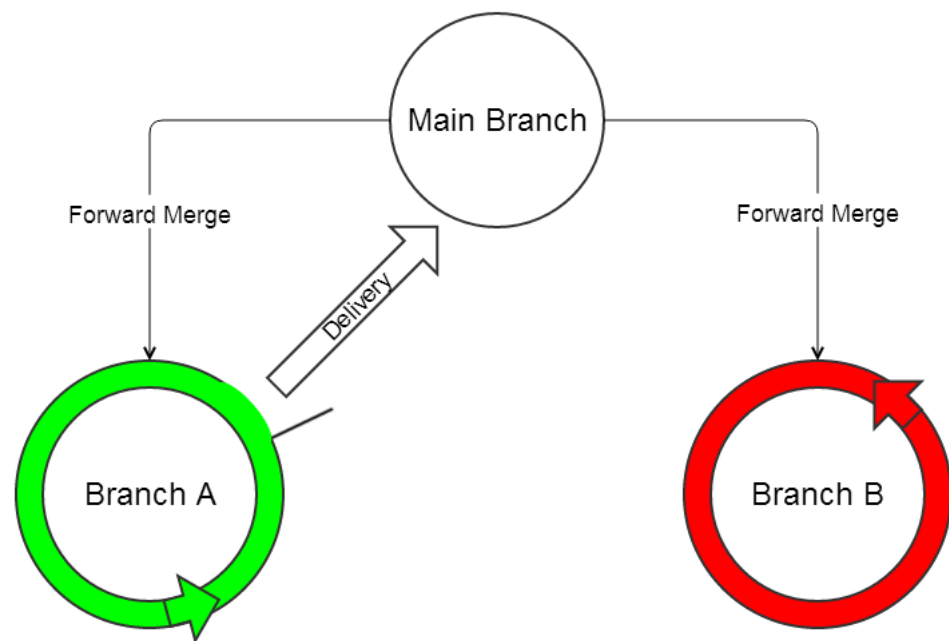


Figure 4.1: Branch A passed all tests and is allowed to deliver its changes to the main branch. Branch B still has failing tests and is therefore not allowed to deliver.

The development of the QlikView.Next project is divided into several feature teams. To allow teams to develop and maintain the same code base in parallel, a branching strategy is in place where each team has at least one development branch. When a team has successfully completed a development iteration the new code will be delivered to the main branch. Automated testing is scheduled to run regularly for all active branches. Normally this means that a full system and integration test run every night for every branch. This enables the teams to detect regression in the software at an early stage. By constantly keeping the software in a stable state it is easier to provide continuous integration, which means making small and regular deliveries from the team branches to the main branch. The teams are not allowed to deliver their code changes of a completed development iteration to the main branch unless all tests have passed.

In figure 4.1, an example of the branching strategy is illustrated. For simplicity, only two branches are shown, but in reality there are up to thirty branches which

code continually integrates with the main branch. In the example, Branch A has completed a development iteration and all the automated tests have passed. This means that the branch is allowed to deliver its code changes to the main branch. The testing works like a wall around the development branch with a closed door. Once all the tests pass, the door opens and there is a road to the main branch. Branch B on the other hand, has also completed a development iteration but some of the tests failed. The wall around the branch now stops the delivery to the main branch by keeping the door closed. Branch B is not allowed to deliver its changes until all the tests pass.

While the teams work with their development, the development branches are kept up to date with the main branch by regularly performing a forward merge from main. This is illustrated by the thin arrow from main to the two branches in figure 4.1.

4.3 Automated Test Development at Qlik

The development of test cases is performed simultaneously on the different branches. New test cases are continuously checked in to the development branch during a development iteration. This leads to an increasing amount of test cases as development continues. Figure 4.2 illustrates the overlap between test cases being run on the main branch and test cases being run on a development branch. This means that all previous functionality is continuously tested on all branches. The new test cases are delivered to the main branch together with the rest of the code. Once the test cases have been delivered they are immediately adopted into the testing suite for the main branch. The other teams will not see the new test cases until they perform a forward merge from the main branch to their development branch.

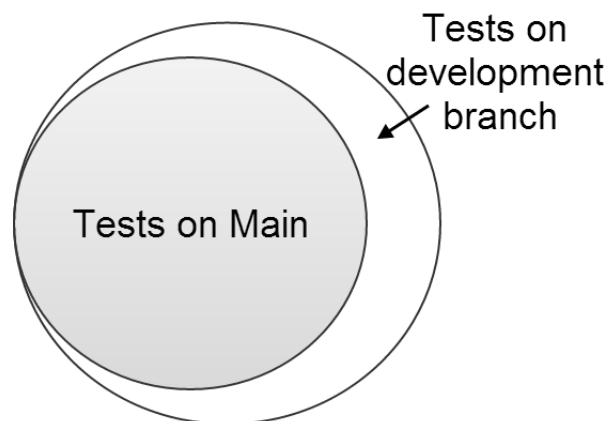


Figure 4.2: An illustration of the overlapping tests runs between branches.

4.4 Tools for Automated Testing at Qlik

As described in section 3.1.3 about automated testing, a tool that can drive the software is necessary for automated testing. Qlik has developed their own automated testing framework, Horsie, that drives the software according to test cases specified by scenarios written in structured natural language. In order to allow the developers to specify test cases in structured natural language, Qlik uses behaviour driven development, see section 3.1.4. The different steps specified in the scenarios are implemented with the help of Horsie's capability to control the software and access the QlikView API. Horsie thus provides an integration between test specification and test execution. Controlling software the way a user would control it provides the possibility to run automated system tests. The relationship between Horsie and QlikView is illustrated in figure 4.3.

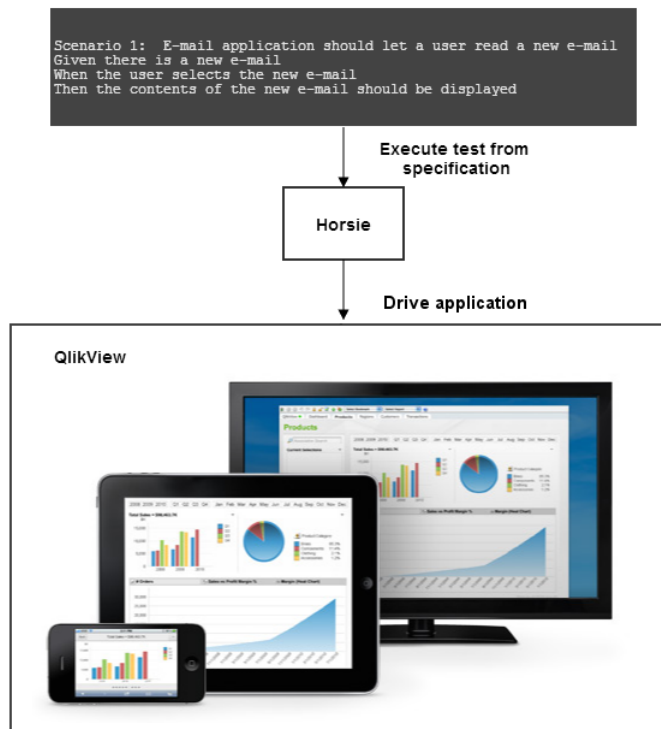


Figure 4.3: Horsie provides an integration between test specification and test execution within QlikView

Qlik uses a continuous integration system from Atlassian called Bamboo to run the automated tests. Bamboo is used to run scheduled tests for all development branches of the source code [3]. Upon finishing a test suite, the produced artifacts are made available through Bamboo's web interface.

4.5 Test Results Artifacts

The artifacts are files such as log files providing detailed summaries of what actions the test framework has performed to drive the software in the way specified by the test case. The artifacts are created by the QlikView software and the test framework, Horsie. QlikView produces log files directly tied to its execution and Horsie produces log files and screenshots which represent problems found externally through testing. For example, if QlikView crashes, the QlikView log file could contain a stack trace. The Horse log file on the other hand, could in that case contain information about the symptom, such as the user interface malfunctioning. Figure 4.4 shows an excerpt of a Horsie log file for a failing test case. An entry marked with *INFO* (yellow) contains information of the test case step currently being executed. If something goes wrong, the entry will have a tag *ERROR* instead and the entry will contain an error message. The red background color of such entry indicates that it is an error. Horsie creates a log file of this kind for each failing test case, as well as screenshots and multiple other log files with information in different formats.

horsie	INFO	Driver.Browser.Selenium.Driver.UIInteraction.Mouse.PriscillaClicker (line: 40) - Clicking on (Toolbar button: Edit mode) "#qv-toolbar-c
horsie	INFO	TestBase.Harness.SpecflowLogTraceListener (line: 0) - -> done: FirstExperienceSteps.GivenIamInAnalyzeMode() (1.1s)
horsie	INFO	TestBase.Harness.SpecflowLogTraceListener (line: 0) - Then I click on coordinates 972,479 in the barchart
horsie	INFO	Driver.Browser.Selenium.Driver.UIInteraction.Mouse.PriscillaClicker (line: 77) - Double clicking on (Grid cell) ".qv-gridcell:eq(0)" @
horsie	INFO	TestBase.Harness.SpecflowLogTraceListener (line: 0) - -> done: VisualizationsSteps.ClickOnCoordinatesInObject(972, 479, "barchart") (0.
horsie	INFO	TestBase.Harness.SpecflowLogTraceListener (line: 0) - And I click and confirm on coordinates 628,284 in the barchart
horsie	INFO	Driver.Browser.Selenium.Driver.UIInteraction.Mouse.PriscillaClicker (line: 77) - Double clicking on (Grid cell) ".qv-gridcell:eq(0)" @
horsie	INFO	Driver.Browser.Selenium.Driver.UIInteraction.Mouse.PriscillaClicker (line: 40) - Clicking on (Confirm selection button) "[tid='selectio
horsie	ERROR	QvCommon.HorsieException (line: 12) - Selection bar did not close after confirming selection
horsie	INFO	Screenshot: R:\CI-IRON4-TDESKCHROME\Packages\TestClientInDesktop\test-results\20140403T031223.416+02_00-horsie-exception.jpg
horsie	INFO	QvCommon.ManagedScreenshotter (line: 137) - Screenshot saved R:\CI-IRON4-TDESKCHROME\Packages\TestClientInDesktop\test-results\20140403
horsie	INFO	TestBase.Harness.SpecflowLogTraceListener (line: 0) - -> error: Selection bar did not close after confirming selection
horsie	INFO	QvCommon.ManagedScreenshotter (line: 137) - Screenshot saved R:\CI-IRON4-TDESKCHROME\Packages\TestClientInDesktop\test-results\ClientTe
horsie	ERROR	TestBase.Harness.SpecflowTestHarness (line: 112) - Test failed: ClientTests.Objects.BarchartFeature.PageAndThenMakeASelectionInA2DimBar QvCommon.HorsieException: Selection bar did not close after confirming selection at QvTest.Interaction.Web.ElementFinderExtensions.RequireElementToNotExist(IElementFinder elementFinder, CssSelector selector, String e at TechTalk.SpecFlow.Bindings.BindingInvoker.InvokeBinding(IBinding binding, IContextManager contextManager, Object[] arguments, ITes at TechTalk.SpecFlow.Infrastructure.TestExecutionEngine.ExecuteStepMatch(BindingMatch match, Object[] arguments) at TechTalk.SpecFlow.Infrastructure.TestExecutionEngine.ExecuteStep(StepInstance stepInstance) at TechTalk.SpecFlow.Infrastructure.TestExecutionEngine.OnAfterLastStep() at ClientTests.Objects.BarchartFeature.PageAndThenMakeASelectionInA2DimBarchart(String browser) in c:\CI-CIA-BHORSIE\test\horsie\src\Te
horsie	DEBUG	Driver.Browser.Selenium.SeleniumModule (line: 53) - Closing browser

Figure 4.4: Excerpt of a log file produced by the Horsie framework.

The artifacts also contain a final error message from the failing test case and an HTML dump of the last element the test framework managed to successfully interact with before failing. An example of what an HTML dump can look like is shown in listing 4.1. The HTML dump is simply a linked list with HTML elements starting with the top level parent element, namely the HTML tag. Following the HTML element is a list of children down to the specific element the test framework interacted with. Every element holds the same set of attributes that the real element held in the web browser during the test execution.

Listing 4.1: Example HTML dump

```

<html class="touch-off" lang="sv">
<body style="-ms-touch-action: none;">
<div class="qv-hub-container">
<div class="hub-content ng-scope" ng-class="{ 'hub-global-search':
  isGlobalSearch}">
<div class="hub-sidebar ng-scope" ng-if="!isGlobalSearch">
<div class="hub-myw-fav">
<ul class="hub-ul">
<li class="hub-li myWork-li qv-active" ng-class="{ 'qv-active':
  activeKey === 'my-work' }" qva-activate="gotToUrl('my')">
<div class="hub-div hub-s-name-style"
  q-translation="Hub.NavigationPanel.MyWork">

```

4.6 Analysis of Test Results

There are multiple types of challenges that are associated with the test result analysis at Qlik. This section aims to describe three of the biggest challenges with the analysis of the test results. These three are:

- Inability to cross reference test case failures
- Intermittent failures
- Determination of problem uniqueness

These challenges can seem trivial in theory but become harder to deal with when plagued with Information Overload.

4.6.1 Cross Reference Failures

Since automated testing at Qlik runs on multiple development branches, each branch produces its own set of test results. Currently there is no easy way to get an overview of what failures that have occurred on which branches, although an overview would be valuable for the results analysis. The number of test cases for a branch can typically exceed a few thousand, where the number of failing test cases may range between zero and a few hundred. With automated tests that run every night for between ten and fifteen branches the amount of test results produced can make it impossible for a human to get an overall overview of the current problems. A manual investigation of the test results would also be very time consuming. See figure 4.5 for an excerpt of the overview of the latest test results for each branch, as it is presented in Bamboo. From the excerpt we can conclude that we have multiple failing branches. Three of the failing branches have only one failure, “Ver12.00-dev-ft-personal”, “Ver12.00-dev-ft-ratatosk” and “Ver12.00-dev-ft-ratatosk-responsive-grid”. Bamboo does not provide a way to cross reference test failures between the branches. Thus, it is not possible to determine if the same test case has failed on all three branches without manually navigating into

each branch and do further investigation. If we would instead want to analyse ten different branches, with tens or hundreds of failing test cases it would be nearly impossible to accomplish an overview by manually navigating to each failure on each branch. The difficulty of finding if a problem exists on multiple branches increases as the amount of results increase, thus we can say this is a challenge that has its root in Information Overload.










Plan	Build	Completed	Tests
--Ver12.00-dev-ft-personal	 #9	3 days ago	1 of 3545 failed
--Ver12.00-dev-ft-phoenix	 #74	6 hours ago	3544 passed
--Ver12.00-dev-ft-phoenix-QLIK-2877	 #5	N/A	No tests found
--Ver12.00-dev-ft-phoenix-QLIK-5237	 #4	N/A	2301 passed
--Ver12.00-dev-ft-prometheus	 #123	15 hours ago	3550 passed
--Ver12.00-dev-ft-publishing	 #23	15 hours ago	4 of 3581 failed
--Ver12.00-dev-ft-ratatosk	 #76	6 hours ago	1 of 3582 failed
--Ver12.00-dev-ft-ratatosk-bugs	 #42	5 hours ago	11 of 3565 failed
--Ver12.00-dev-ft-ratatosk-codearea	 #2	1 week ago	2 of 3314 failed
--Ver12.00-dev-ft-ratatosk-responsive-grid	 #20	5 hours ago	1 of 3405 failed
--Ver12.00-dev-ft-responsive-client	 #6	2 weeks ago	3304 passed
--Ver12.00-dev-ft-sdk	 #53	5 hours ago	2 of 3576 failed
--Ver12.00-dev-ft-thejungle-refactor	 #94 	11 hours ago	3594 passed
--Ver12.00-dev-ft-theming	 Never built		
--Ver12.00-dev-horsie-sdk	 #46	11 hours ago	331 of 3720 failed
--Ver12.00-dev-main-for-stability-testing	 #159	8 hours ago	3582 passed
--Ver12.00-dev-MergeFromVer11.20	 #1	2 weeks ago	5 of 3198 failed
--Ver12.00-dev-QLIK-1476	 #6	4 days ago	113 of 3402 failed

Figure 4.5: An excerpt of the overview showing test results of the latest test run from each branch.

4.6.2 Intermittent Failures

One of many challenges with the test results at Qlik is the management of intermittently failing test cases. For example, timing issues when interacting with external dependencies such as browsers or overloaded testing machines might cause a test failure one day but not the day after. When we mention intermittent failures in this thesis it will refer to false negatives caused by the testing environment and not actual intermittent failures in the software. In figure 4.6, it is shown that

several test runs of the branch “main for stability testing” yield different results. Note that neither the code being tested nor the test cases have changed between the different runs.

Status	Reason	Completed	Duration	Test results
🟢 #111	Scheduled main-for-stability-testing-57	5 hours ago	202 minutes	3350 passed
🔴 #110	Scheduled main-for-stability-testing-57	8 hours ago	229 minutes	1 of 3350 failed
🟢 #109	Scheduled main-for-stability-testing-57	11 hours ago	200 minutes	3350 passed
🔴 #108	Scheduled main-for-stability-testing-57	14 hours ago	193 minutes	2 of 3350 failed
🟢 #107	Scheduled main-for-stability-testing-57	17 hours ago	197 minutes	3350 passed
🔴 #106	Scheduled main-for-stability-testing-57	21 hours ago	183 minutes	2 of 3350 failed
🔴 #105	Scheduled main-for-stability-testing-57	1 day ago	182 minutes	1 of 3350 failed
🟢 #103	Scheduled main-for-stability-testing-57	1 day ago	219 minutes	3350 passed
🔴 #102	Scheduled main-for-stability-testing-57	1 day ago	221 minutes	3 of 3350 failed

Figure 4.6: The same code base tested nine times yields different results due to intermittent failures

Code changes from the main branch are continuously being merged into the development branches on a daily basis. This can lead to an intermittent failure that originated from one branch, being copied to other branches. Without the overview of failures across branches, each team might think they are responsible for the failure, although it had nothing to do with their development. Thus, analysing intermittent failures becomes an even bigger challenge because of the inability to cross reference failures (see section 4.6.1).

Currently, to be confident when deciding if a test case failure is due to a real problem on a branch or to an intermittent failure, the most common procedure is to run the tests several times. If an overview of all branches with a particular failure was available, the time spent running the tests again could be saved.

4.6.3 Determination of Problem Uniqueness

Besides examining single failures there is an issue with test cases failing in different ways, i.e. a test case may fail differently from branch to branch. A naive success/failure comparison based on test cases between two branches will indicate that the same problem is present in both branches, but this may not be the case if the test case failed differently in the two branches, i.e. at different stages within the test case. For example, a test case with six steps, could fail at the first or the last step but it will be the same test case name appearing in the logs for both scenarios. In order to discover the difference between the two failures, additional information about the failure, such as the error message, has to be taken into account. This information could be extracted from the test result artifacts. Because of the high amount of test runs and test cases it is hard to manually look up all the relevant information for each test case.

Similarly there is the situation where two different test cases fail in the same way, for example during a setup phase which both test cases have in common. Then there is no need to treat the problem as two different issues just because they originated from two different test cases. The real issue is still in the setup phase in this situation. Once again, a naive comparison based on the test case name would not reveal the pattern of the problem. A clustering of all the test cases that failed because of the same cause, would help with the identification of the actual problem.

4.7 Bug Report Creation Tool

Qlik has developed a tool that automates the process of creating a bug report in their bug tracking system from automated test failures on the main branch. The code on a development branch is not considered complete until it has been delivered to the main branch. Thus, it is not desirable to generate bug reports based on failures on the development branches. The tool downloads the test result artifacts from all the failed tests associated with the main branch. Each of the failed tests is then investigated by the tool. The process is illustrated in figure 4.7. A failed test contains an error message generated by the Horsie framework. The error message is considered an identifier for the problem which caused the test case to fail. The tool then proceeds to check if this error message has been encountered before by making a search in the bug tracking system for any bug reports with the same error message as the title. If the error message has not been previously encountered, the tool submits a new bug report to the bug tracking system. In the case where a report already exists with the error message as the title, additional information about the current failure is attached to the already existing bug report. A bug report can thus be considered a group, or a cluster of test case failures, as in the concept of Clustering, presented in section 3.2.2. The similarity measure $sim(f_1, f_2)$ between two failures, f_1, f_2 , with error messages err_1, err_2 , would then have two possible outcomes, 1 if the error messages are strict equal or 0 if they are not equal.

$$\text{sim}(f_1, f_2) = \begin{cases} 1 & \text{if } \text{err}_1 = \text{err}_2 \\ 0 & \text{otherwise} \end{cases} \quad (4.1)$$

Although the automatically generated bug reports provide a starting point for investigations of bugs for the main branch, some of the issues with the analysis still need to be taken care of. The tool only processes the failures for the main branch and thus, there is still no way to cross-reference the failures. Additionally, if the bug reports are considered as clusters of failures, there is still no way to cluster failures without creating bug reports. There is also a problem with the determination of unique problems. The error message comparison has resulted in many similar failures not being clustered together because their error messages have not been identical. Listing 4.2 shows an example of three typical error messages. Two of them are very similar but using an exact string match they would be separated. Additionally, usage of the tool has introduced false duplicates. Failures that have identical error messages could potentially have different root causes and should thus ideally be described in different bug reports.

Listing 4.2: Two of the three error messages have the same base with different variables and the third error message is completely different.

```
[The HTTP request to the remote WebDriver server for URL
http://localhost:7055/hub/session/d111075b-c10b-4da6-bd7a-a3509f208b5e/elements timed out
after 60 seconds.]
[The HTTP request to the remote WebDriver server for URL
http://localhost:7055/hub/session/2d87ed00-b432-4e8f-9c5e-cc1b1c25b500/elements timed out
after 60 seconds.]
[Selection bar did not close after confirming selection]
```

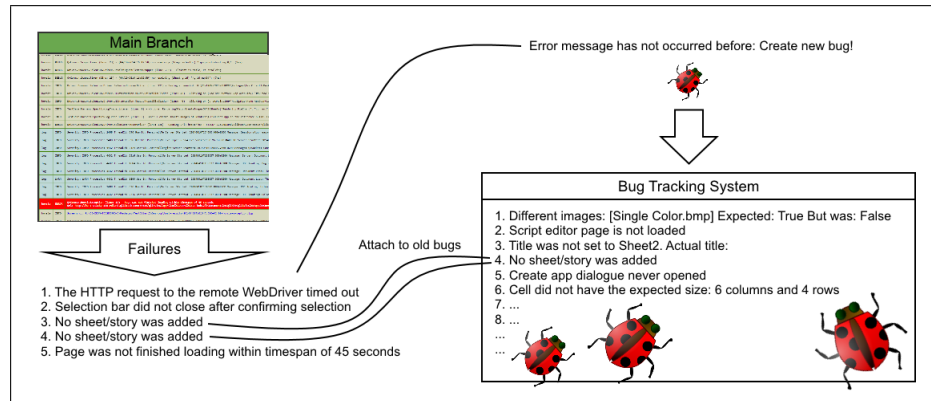


Figure 4.7: Automated bug report creation tool. If a new error occurs a new bug report is created. If a bug report already exists for a certain error message the tool will attach data about the new failure to the existing bug report.

4.8 Case Summary

At the software company Qlik, automated tests run every night for multiple branches, resulting in information overload. The test results are available through a web interface, but for a human the analysis is very challenging and time consuming. There is currently no way to cross reference failures across branches, determine specific problem areas or identifying intermittent failures. Qlik has developed a tool that automatically generates bug reports based on the error message of the test case failure. The tool has limitations and has also introduced false duplicates.

In this chapter we outline our methods for approaching the problems defined in section 1.3. The overall idea of the solution is presented in the first section. Our choices of which components of a failure to take into account are discussed in the next section, followed by a description of how the test failure data was collected. The Implementation section is the core of this chapter. The algorithm and the formulas used for our computations are presented here. The last section of this chapter is about how we conducted the evaluation of our tool and all the steps involved in that process.

5.1 Solution Idea

The aim of this project is to create a tool that analyses the artifacts created by Horsie during test runs. We choose to call the tool NIOCAT – Navigating Information Overload Caused by Automated Testing. The output of NIOCAT is a file containing the partition of the failed test cases. The output file should thus define different clusters of test case failures from the input data. We would like to cluster test cases which have failed because of the same problem. Each cluster would thus represent a unique problem with the software based on the input data. A problem can contain one or multiple test case failures.

Figure 5.1 illustrates a solution that processes test results and produces a clustering of failures. Each of the small circles represent a failed test case where the larger circles illustrate which failed test cases that have been grouped together. As can be seen in figure 5.2 the test cases that are grouped together should be similar. In the figure three failures are highlighted, two of them are caused by the inability to read a new e-mail while the third test case has failed when downloading an attachment on an iPad-device. The two test cases regarding the ReadEmail-feature have been grouped together because they represent the same problem even though they failed in different web browsers. From now on we will refer to a group of similar test case failures as a problem and a single test case failure is called a failure.

The partition of the failed test cases is aimed to serve the test results analyst with a starting point for further investigation and analysis. The input could be data from test runs on a single branch or multiple branches. The use case for a development team leader might be to analyse data from test runs within the last

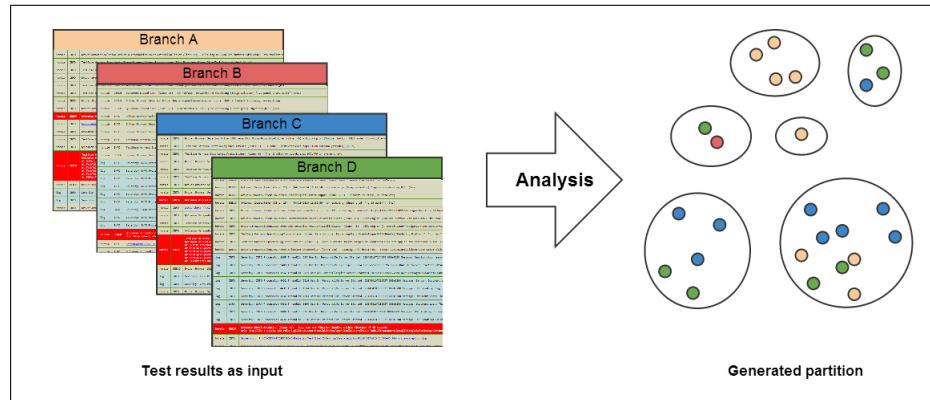


Figure 5.1: NIOCAT outputs a partition of failures based on the test results input

seven days, for the team’s branch only. The partition would then help identifying which failures that are related to the same problem and which problems that have been introduced or fixed. A configuration manager though, might look for a bigger picture and a use case could be to analyse the latest test run for each development branch. Since there is currently no way to get an overview across multiple branches, even a naive clustering based on the test case names has potential to support the results analyst.

5.2 Selection of Components of a Test Case Failure

A tool that provides an overview of failed test cases across branches would be helpful in the analysis of test results. Although, as explained in section 4.6.3, a clustering based on the name of the test case alone would not be ideal for the needs at Qlik. Further components of a failure needs to be taken into account.

A tool currently being used for automatic bug report creation was presented in section 4.7. The tool uses the error message as identifier instead of the test case name. We wanted our tool to overcome the issues introduced by their current bug creation tool. Not only did we want to allow cross referencing test case failures between branches, we also wanted our tool to achieve a more accurate clustering, by which we mean a more accurate grouping of failures that are similar. In order to achieve the improvement, we decided that our tool should account for a combination of the test case name and the error message.

In addition, we wanted to investigate if any additional information about the failure could improve the accuracy of the clustering further. Both the error message and the test case name are components that contain textual information about the failure. Inspired by related research, we wanted to go beyond the textual information and add a component consisting of execution data. In the Qlikview.Next project, the only client that the user interacts with is the browser (see section 4.1 about Qlik). Therefore we chose to consider the underlying HTML structure of

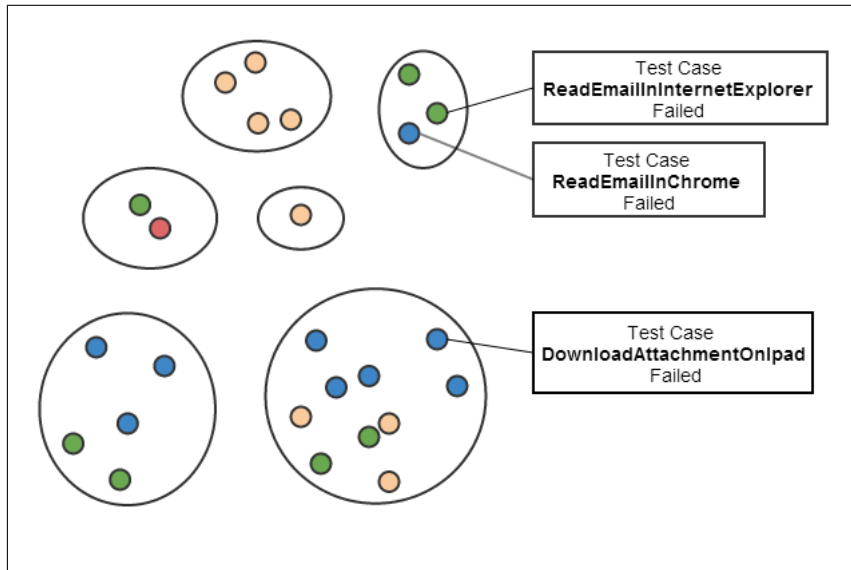


Figure 5.2: Each small filled circle in the partition is a failed test case. The test cases are grouped by similarity. Each group is referred to as a problem.

an element interacted with in the browser as a representation of the current state of the QlikView application. Thus, the three components we used as a basis for our clustering of failures were:

- the test case name
- the error message
- the HTML structure for the last element that was interacted with before the test failure occurred.

5.3 Data Collection

We created a tool that via a Bamboo API downloaded all the artifacts produced during test runs for all the development branches. For more information about the artifacts, see section 4.5. The artifacts are only available with the Bamboo API for one week. Artifacts older than one week are removed. Thus we needed this tool to be able to store artifacts over a longer period of time and to be able to access and review the same data over the course of a longer period.

From the start of the project there were some uncertainty about how our methods were to be evaluated, thus we decided to save all the possible data to not limit our future opportunities.

5.4 Implementation

NIOCAT processes artifacts placed in a certain input directory. The directory structure provides information about which branch artifacts belong to and which test run is associated with what artifacts. The mission of NIOCAT is to distinguish different problems throughout a set of artifacts. To be able to cluster failures together we introduced a similarity measure between two failures and then defined a threshold for how similar two failures had to be in order to be clustered together. NIOCAT will automatically create its own grouping of failures based on the input data.

In order to cluster one or several test failures into groups, or problems, we used an algorithm that processed all the failures and calculated the similarities between them. When assigning a failure to a cluster, we did not want to favour large clusters. Therefore, we defined the similarity between a failure and a cluster as the average of the similarities between the new failure and all failures already in the cluster. The similarity measure between two failures was based on a weighted average of the cosine similarity of the different components of a failure. The components we chose to take into account was the name of the test case that failed, the error message extracted from the log file and the underlying HTML structure of the element that was last interacted with before the failure occurred, as discussed previously in section 5.2. The weighted average thus introduces a possibility to put more emphasis on a certain part of information about the failure. We will come back to our tuning of weights in section 5.5.5. The algorithm NIOCAT used to cluster similar failures follows.

Algorithm

1. Receive the input as a collection $B = \{b_1, b_2, \dots, b_n\}$ of branches, with their respective test results for the set of test runs $b_i R = \{b_i r_1, b_i r_2, \dots, b_i r_l\}$, where n is the number of branches and l is the number of runs for branch b_i .
2. Represent each failed test case as a document d , so that a cluster c of documents will represent a problem with the software. A document can only be present in one cluster. The set D of documents represents all documents within the whole collection of test runs for all branches. Let the set C represent all clusters.
3. For each document $d_i \in D$ do
 - (a) Represent the document with three vectors \vec{d}_{i1} , \vec{d}_{i2} and \vec{d}_{i3} , one for each component. Each vector is built using the terms within the document for that component, as in the Vector Space Model described in section 3.2.1.
 - (b) Retrieve the clusters $c_j \in C$ that have been created so far. Let the documents $D_j = d_j^1, d_j^2, \dots, d_j^k$ be all the documents belonging to cluster c_j .
 - (c) For each pair (d_i, c_j) , compute a similarity score $\text{sim}(d_i, c_j)$ between the document and the cluster. The score is based on the average

similarity score between the document d_i and the documents D_j within the cluster c_j , such that

$$sim(d_i, c_j) = \frac{\sum_{t=1}^k docSim(d_i, d_j^t)}{k} \quad (5.1)$$

where

$$docSim(d_i, d_j^t) = \frac{\sum_{l=1}^3 w_l \cdot cosSim(d_{il}, d_{jl}^t)}{\sum_{l=1}^3 w_l}. \quad (5.2)$$

The document to document similarity score is based on a weighted average similarity score $cosSim(a, b)$ for each document component and w_l are the weights for the components, respectively. The component similarity $cosSim(d_{il}, d_{jl}^t)$ is computed according to the cosine similarity, as in section 3.2.1

$$cosSim(d_{il}, d_{jl}^t) = \frac{\vec{d}_{il} \cdot \vec{d}_{jl}^t}{\|\vec{d}_{il}\| \times \|\vec{d}_{jl}^t\|} \quad (5.3)$$

- (d) Retrieve the cluster c_{max} with the highest value of $sim(d_i, c_j)$. If $sim(d_i, c_j)$ is greater than a predefined threshold T , add d_i to the cluster c_{max} .

5.5 Evaluation Based on Accuracy of the Output

Since there is currently no way to cross reference test case failures between different branches NIOCAT would provide a new feature to the test result analysis. Currently, when processing the test results presented by Bamboo, the information about a problem is based on the name of the test case. The automatic bug report generation tool used on the main branch is based on the error message. NIOCAT creates a clustering based on a combination of those two components and an HTML dump. We wanted to determine if NIOCAT was more accurate than the current tools being used. Thus, we needed a way to compare the accuracy of our produced partition and the two baseline partitions created using exact string comparisons of the test case names or the error messages.

To evaluate the accuracy of the different clusterings we chose to calculate a similarity measure between the different clusterings and their corresponding reference clusterings for those particular input data sets. Currently there are no such reference sets available, thus part of our work was to create the reference partitions manually. Similarity measures between a produced partition and its corresponding reference partition would provide us with a measurement for how well NIOCAT manages to automatically classify the failures. This measurement could then be compared to the same measurement for a clustering based on the test case name and a clustering based on the error message.

At the start of this project we found that the most intuitive similarity measure for different partitions of the same data set was the Rand Index, presented in section 3.2.3. Because of some flaws of this index which were also presented in

the same section, we have chosen to use the Adjusted Rand Index as similarity measure instead, which was introduced with the intention of overcoming these issues. Computing the adjusted rand index between an established control set and a set generated by NIOCAT will give us the percentage of correct decisions and thus provide us with a benchmark for our method.

5.5.1 Creation of Subsets for Reference

NIOCAT is intended to be used to support the daily analysis of test results from the different development branches. The daily analysis can involve test runs from a single, multiple or all branches. The total amount of test cases on a normal run ranges in the tens of thousands, where typically zero to a few hundred fail. There is no specific requirement for the size of the subset for reference, but it should be a representative of the output from a normal test run. It is desired that the subset contains both failures that can be clustered together as well as failures which represent a unique problem. We wanted to evaluate NIOCAT against a typical use case but found it impossible to define a single representative reference data set. Therefore we chose to create several reference sets, that would represent a few slightly different use cases. Three different points in time were selected randomly from a recent timespan. We wanted to use recent data so that it would be familiar to the expert during the analysis and thus allow a more accurate creation of the reference subsets. To create the subsets, data from test runs at each point in time was extracted. A brief description of each of the data sets follows in the next three sections and a summary of their properties can be found in table 5.1.

The clustering of failures within each set was done by evaluating each failure manually and comparing it to the other failures within the set. Manual evaluation at Qlik basically means going through the produced artifacts for the failed test cases. The work includes looking at screen shots, reading log files and interpreting error messages. This procedure is currently the way the analysis is performed and is thus a good baseline for our reference set. To help perform the analysis, we acquired assistance from an expert within the domain, who works with the test analysis at Qlik on a daily basis.

	Subset 1	Subset 2	Subset 3
First Test date	2014.03.27	2014.03.23	2014.03.28
Sample period	1 day	1 day	1 week
Number of branches	2	10	1
Total number of test runs	2	10	9
Total number of test cases	6696	33160	26464
Failing test cases	25	11	61
Number of clusters	4	9	13

Table 5.1: Properties of the three different data subsets used for reference

5.5.2 Subset 1 - Main and Development Branch

The first subset consisted of data from two different branches, the main branch and a development branch. A test run from each of the two branches was selected so that both runs had been run simultaneously in time. The test runs for the main and the development branch resulted in 6 and 18 failed test cases, respectively. When the manual evaluation of the results was done we could conclude that the set only contained four distinct problem areas. As seen in figure 5.3, one of the problems caused ten test case failures across the two branches while another problem caused only one failure in one branch. Two of the problems caused seven failures each.

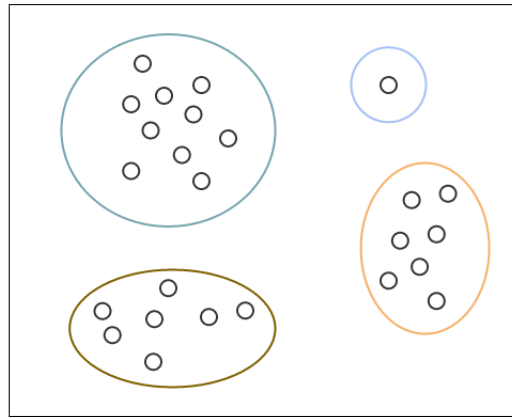


Figure 5.3: Illustration of the manual partition of reference set 1.

5.5.3 Subset 2 - All Branches

For the second subset, data from one run from each of the development branches was selected. Although the total number of test cases exceeded 30000, only 11 of them failed. Note that even though Subset 2 contains almost 5 times more test cases than Subset 1, the number of failing test cases is lower. Since the two subsets are from different points in time, the software is in a different state and thus results in completely different test results.

In contrast to the previously described subset, which had only four problem areas, with multiple failures for each one, Subset 2 has a large spread of problems for its included failing test cases. Nine out of the eleven failing test cases in Subset 2 originated from unique problems. Thus, subset 2 provides us with a reference set with characteristics different from Subset 1. The partition is illustrated in figure 5.4.

5.5.4 Subset 3 - Single branch

Data for the third subset was selected from one week's testing on a single development branch. The scheduled nightly tests and a few manual test runs added up to 9 runs and 61 failed test cases in total that week. The manual partition resulted

in 13 different clusters, of various sizes. Figure 5.5 contains an illustration of the partition. The largest three clusters contain 18, 11 and 8 failures, respectively, while the remaining 10 clusters contain only 5 failures or less each. Subset 3 thus provides us with a data set containing multiple problems, resulting in single or multiple test failures.

5.5.5 Component Weight Tuning

The specified algorithm requires four predefined parameters, the weight for each component of the failure and the similarity threshold. Figure 5.6 illustrates two different configurations in the space of weight components. The circular marker indicates a configuration with weights equally distributed between the different components while the diamond represents a configuration where more weight is placed on the error message and the rest is equally distributed between the HTML and the test case name. The adjusted rank index was calculated for the output for all three subsets in order to find the combination of values that would yield the highest adjusted rank index. Each combination of parameter values from 0.0 to 1.0 with incrementation size of 0.05 was evaluated. The total of all three weights added together during the computation was 1, which for all possible threshold values resulted in almost 5000 different combinations of parameters. The weight combinations corresponding to the highest found index for each subset are presented in section 6.1 in the Results chapter.

5.6 Evaluation Based on Qualitative Feedback

As stated by Runeson et al. much of the knowledge that is of interest for a case study researcher is possessed by the people working in the case [26]. Thus, as a complement to the evaluation based on the pairwise agreement between the output and the reference partition, we conducted a focus group to receive qualitative feedback of our work. A focus group is basically a session where data is collected by interviewing several people at the same time [26].

Three people from the research and development department at Qlik participated in the focus group. Two of them work with configuration management and process automation, and their daily work involves analysis of results from automated testing. The third participant works with development of the automated testing framework and is also working with analysis of test results on a regular basis.

As suggested by Runeson et al. we conducted the focus group by executing a number of phases. Firstly, we explained the concepts of a focus group to the participants, followed by a brief description of our purpose with our work. In the next phase we explained how NIOCAT works and demonstrated an example of how to navigate the output within QlikView. After the demonstration the participants got to navigate and play around with the application within QlikView themselves. The interview phase, which is the main part of the focus group, was based on five questions. The purpose of the first three questions was to investigate the usefulness of NIOCAT. The fourth question was asked in order to open up for

suggested improvements and the last question was there to cover everything that did not fit in under the previous questions. The five questions were:

1. Do you have a clearer overview of the test results now than you had before?
2. Looking at the result you can see and navigate through in QlikView, can you draw any conclusions?
3. Would NIOCAT be of use for you in your daily work?
 - If yes, how?
 - If no, what is needed for you to use it in your daily work?
4. Is there anything else that you would have wanted to see, or anything you would have liked to change?
5. Do you have any other comments?

After conducting the interview we summarized our major findings in order to confirm that the opinions and ideas had been properly understood.

5.7 Visualization in QlikView

NIOCAT will group test case failures found in the test result artifacts according to the algorithm described in this chapter. To be able to visually represent the partitions created by NIOCAT we used QlikView. The choice of data discovery software came naturally since it is the product of Qlik. An advantage of this choice is that all possible users of NIOCAT are familiar with QlikView. QlikView enables a user to interactively explore large quantities of data. By combining the analysis from NIOCAT with QlikView, it becomes possible to browse among problems and failures found in analysed test artifacts. QlikView allows creation of charts and other visualization items. In QlikView, the user can make selections of one or multiple items in order to examine data related to the selected items. The related items will be displayed with a white background and the excluded items will have a grey background.

In figure 5.7 an example is shown of how a QlikView application with NIOCAT data could look. In the figure a problem (with id 7) is selected and the items with white background are related to the selected problem. A lot of information can be obtained using QlikView. For example, in the list of branches to the top left in the figure we can see that the selected problem is occurring on four different branches. More specifically two separate test cases failed with the same error message, which is seen in the lists of error messages and test cases that have one and two white items respectively.

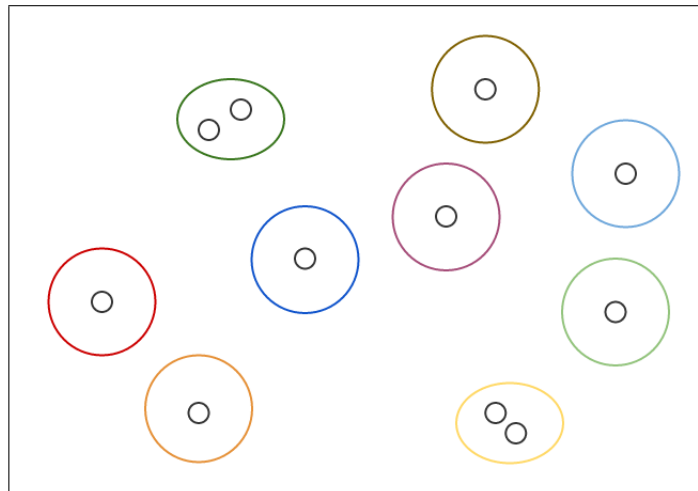


Figure 5.4: Illustration of the manual partition used for reference set 2.

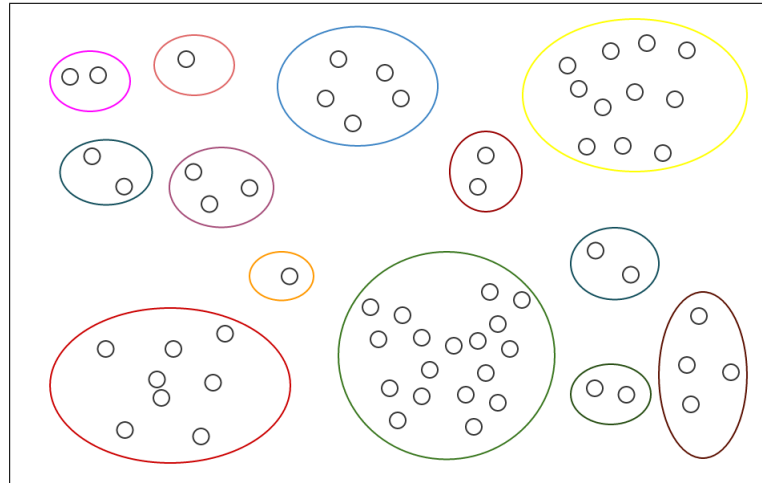


Figure 5.5: Illustration of the manual partition used for reference set 3.

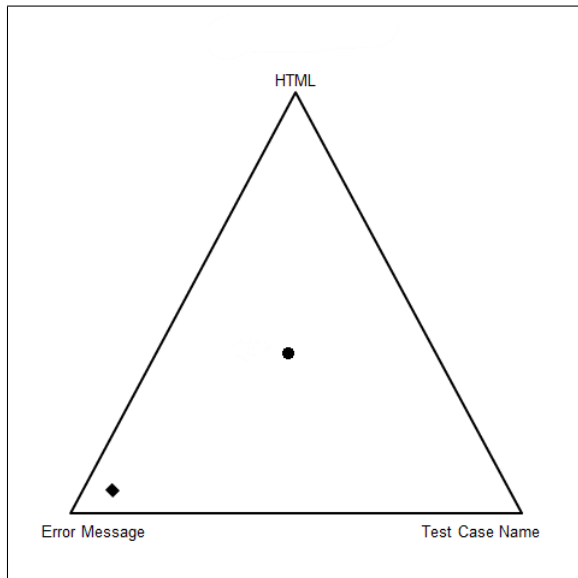


Figure 5.6: Two different configurations in the failure component configuration space.

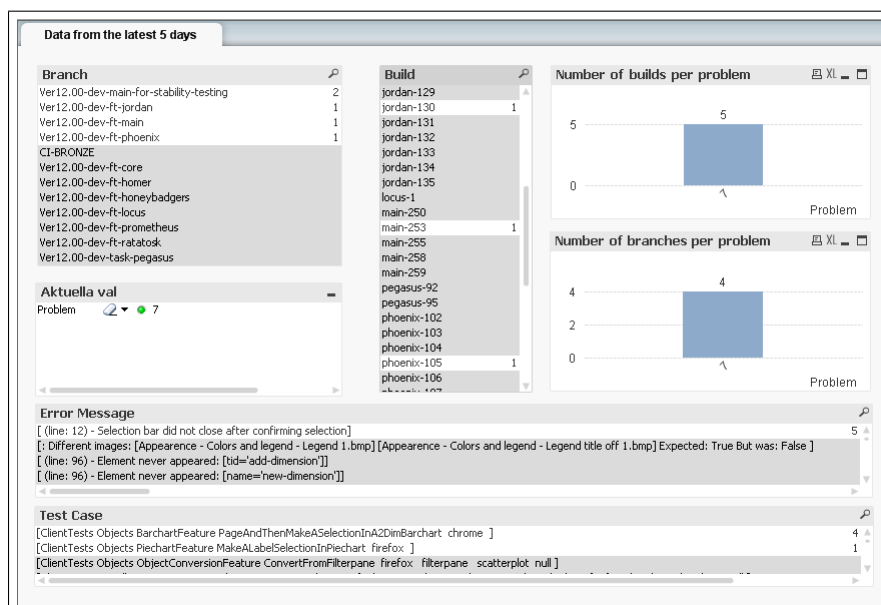


Figure 5.7: An output of NIOCAT visualised in QlikView

In this chapter, results for the two different evaluation techniques will be presented. We will present the highest achieved adjusted rand index for each subset and the corresponding weights retrieved through the weight tuning for each specific subset. A comparison of the accuracy of the output of NIOCAT compared to the existing clustering approaches will summarise the first evaluation. The chapter ends with the results from the second evaluation, the focus group.

6.1 Similarity Between NIOCAT and Manual Analysis

6.1.1 Subset 1

The highest achieved value of the adjusted rand index for Subset 1 was 0.59, more than half of the pairs were classified correctly. This value was achieved with 22 different combinations of parameters. The threshold value varied from 0.55 to 0.85. The combinations for each threshold corresponding to the highest rand index are shown in figure 6.1. As seen in the figure, the highest rand index was achieved by configurations where more weight was placed on test case name and error message than on HTML. Another observation from the figure is that the best configurations for this subset tend to move from test case name to error message as the threshold rises.

6.1.2 Subset 2

The output of NIOCAT for Subset 2 reaches the adjusted rand index value of 1 which corresponds to a produced partition identical to the reference partition created manually. There are almost 400 combinations of parameters that yield an identical partition. The threshold can vary from 0.6 to 0.95. The weight combinations for each threshold that give an adjusted rand index value of 1 are shown in figure 6.2. In the figure we can see that as the threshold increases the weight of the configurations tend to move from test case name towards HTML and error message

6.1.3 Subset 3

The adjusted rand index for the output of NIOCAT for subset 3 reached 0.96 for 4 different combinations of component values. The combinations for each threshold value are shown in figure 6.3. It is shown that the value of the threshold component can be either 0.65 or 0.7 for the output to correspond to the highest index. The weight configurations that produced the highest adjusted rand index were mostly equally distributed among the three components, with a slight movement towards the error message.

6.1.4 Comparison Against Current Approaches

As seen in table 6.1, NIOCAT performs better than approaches currently being used at Qlik, when it comes to accuracy of clustering compared to the manual approach for our three chosen subsets.

Subset/Approach	Error Message	Test Case Name	NIOCAT
Subset 1	0.15	0	0.59
Subset 2	0.65	0	1
Subset 3	0.5	0.2	0.96

Table 6.1: Adjusted rand index for clusterings of each subset and clustering approach

In the table we can see that the adjusted rand index for Subset 1 is 0.15 for the error message approach and 0 for the test case name approach, meaning that neither of the baseline approaches manage to create a partition close to the partition created manually. NIOCAT, on the other hand, reaching the index 0.59 for Subset 1, manages to classify more than half of the failure pairs correctly.

The test case name approach does not yield a better result for the second subset, although the error message approach reaches the index 0.65. NIOCAT still outperforms the error messages approach with the ability to produce an output identical to the manual partitioning.

Even for subset 3 NIOCAT outperforms the current approaches. The adjusted rand index 0.96 indicates that almost all of the failure pairs have been classified correctly. The error message approach results in half of the pairs correctly classified and the test case name approach only manages to classify 20% of the pairs correctly.

6.2 Feedback from the Focus Group

6.2.1 Purpose Accomplishment

The answers to all the questions regarding the usefulness of NIOCAT were positive. All the participants expressed that after viewing the output of NIOCAT they had a clearer overview of the current development status.

6.2.2 Identified Use Cases of NIOCAT

Regarding what conclusions could be drawn by exploring the output in QlikView (the second question), the participants confirmed that being able to cross-reference failures and problems across branches, enabled drawing conclusions, which in turn can help making decisions. The participants further identified three additional use cases of NIOCAT that we had not previously thought of. These use cases will be presented in this section.

An intended characteristic that the participants observed was the wide spread of problems through the software, meaning that, given a specific problem, an analyst can quickly find how many branches that are affected. This applies to both problems or specific failing test cases. It is possible to see on how many branches a NIOCAT-classified problem is occurring as well as to see on how many branches a specific test case has failed.

Global frequency for either a specific test case or for a particular problem was mentioned as a further benefit of NIOCAT, where global frequency means how often a problem is occurring or how often a specific test case fails across all of the branches. A participant mentioned that there is a value in seeing how many failures in total that a problem has caused. This would not have been possible without being able to cross-reference failures across branches.

One of the participants is responsible for deciding if a development team is allowed to deliver its code changes to the main branch or not. Sometimes a team is allowed to deliver its changes even though all the tests have not passed. Using NIOCAT, the participant could quickly determine which problems only occurs on one branch. If the problem only occurs on one branch, that team is obviously responsible for the failure and thus may not deliver its changes to main. The participant expressed that the decision, that was previously challenging to make, could quickly be made using NIOCAT.

The overview provided by NIOCAT introduced the possibility to see what problems were most common across all branches and test runs. The participants quickly figured out that a measurement of priority thus could be established, which was not previously possible. This is a use case we had not previously thought of. The participants had slightly different ideas of how to define the priority order. One of the participants considered the problem that were occurring at most branches the most important to resolve, while another participant wanted to prioritise the problem that caused the most test runs to fail.

Another comment from the group was that the information provided by NIOCAT could be useful for the development teams as well as the test results analysts. The teams can now quickly determine if a test case failure is occurring on other branches. This could help them determine if they should invest more resources in investigating the failure or if it originates from another team. Yet another use case that we had not had in mind.

The third use case that was new to us was suggested as a long term perspective of the tool. A participant pointed out the possibility to identify problem areas with the help of NIOCAT. The test developers could then extend their test suites around the areas where a lot of problems occur. Even the manual testing could be extended around these areas identified by NIOCAT.

6.2.3 Potential Usage

Regarding the potential usage of NIOCAT, two of the three participants explicitly stated that they would use NIOCAT in their daily work if it was available to them. The third participant estimated that his potential usage would be on a daily to weekly basis.

6.2.4 Suggested Improvements

All the participants had ideas and suggestions for how the work with NIOCAT could be continued. One suggestion was to incorporate the analysis done with the vector space model with the current automatic bug creation tool. Another suggestion was to keep NIOCAT separate from the current bug report creation tool but still create a connection to the bug repository so that unique problems could be referenced with an ID from the repository and that failures could be tied to them.

To further benefit from the output of NIOCAT the focus group would like to see direct links to even more information about the test case failures. This information could include the original log files and screenshots generated by Horsie.

6.2.5 Additional Findings

The group members were positive about including the HTML data in the analysis, since they experience that there are a lot of failures residing in the user interface.

During the focus group meeting, the participants acknowledged the potential of NIOCAT and requested a full analysis across all development branches with data from a week back from the time of the meeting. During a short break we let NIOCAT perform an analysis of the requested data and presented the output to the group. The group members were fascinated by what could be accomplished within a few minutes and the results caused an intense discussion. Based on the output, participants were eager to take action and discuss problems with development teams. A quote from one of the participants was “an overview like this does not currently exist”. Another participant expressed immediate need and eagerness to start using the tool. Other quotes from the group members were “the following few weeks until the tool is put into production will be painful since we know how much the tool could help us” and “imagine all the noise and administration you would get rid of using this tool”.

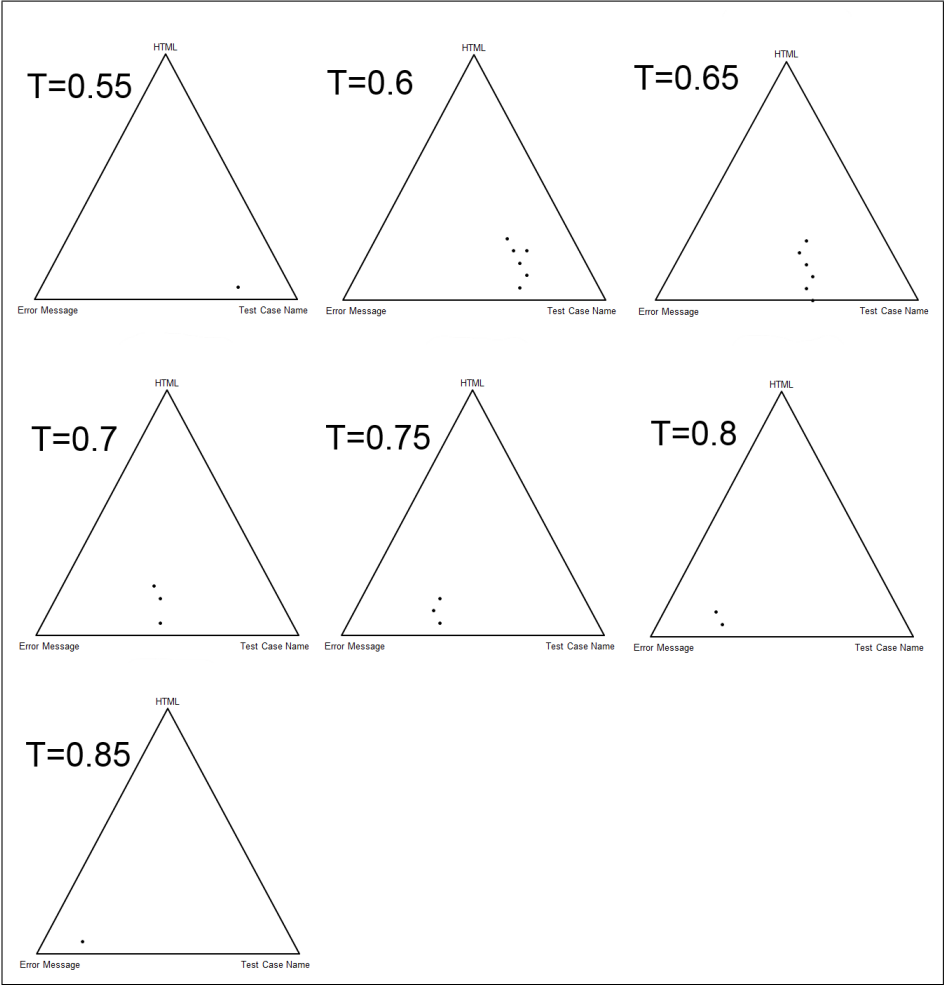


Figure 6.1: The weight configurations for subset 1 that yielded the highest adjusted rand index.

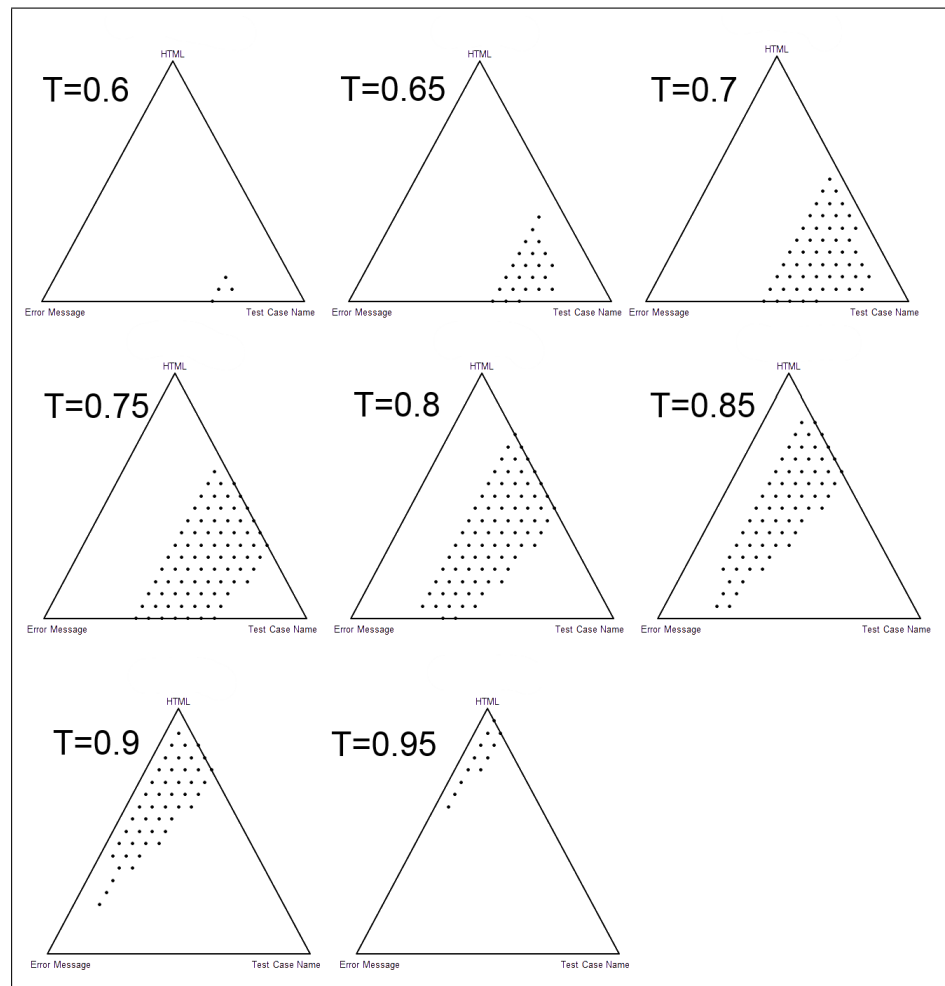


Figure 6.2: The weight configurations for subset 2 that yielded the highest adjusted rand index.

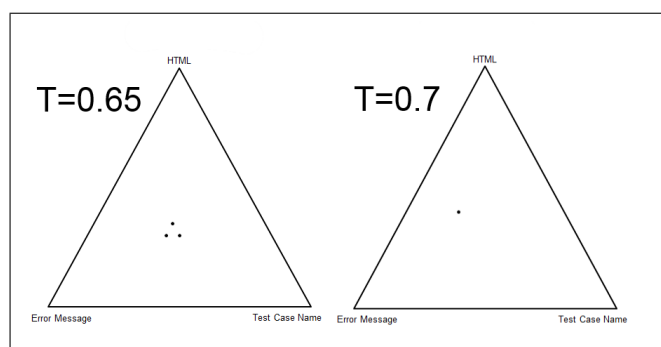


Figure 6.3: The weight configurations for subset 3 that yielded the highest adjusted rand index.

7.1 Threats to Validity

As seen in the results chapter, NIOCAT performs exceptionally well on the three chosen subsets, compared to the baseline approaches. Although, there are many different methods of measuring clustering similarity. The results could possibly be different if measurements other than the adjusted rand index were used. Additionally, the reference sets that were created through manual analysis may contain faults. Human error and the flaws of the current manual analysis approach might have caused incorrect clusterings.

The analysis of the feedback from the focus group provided a complement to the results based on the adjusted rand index. Apart from being able to accurately create clusters in line with a manual analysis, NIOCAT's potential to save time and support analysts in making decisions was confirmed. Although it became clear that deploying the tool at Qlik was greatly desired, one should take into account that this study was specific for the context at Qlik. The results do not prove the usefulness of NIOCAT in another context. More research and analysis might be needed to investigate if NIOCAT could be useful outside Qlik.

7.2 Discussion of Results

The results for Subsets 2 and 3 exceeded our expectations by receiving the adjusted rand indices of 1.0 and 0.96 respectively. Subset 1 on the other hand received a lower adjusted rand index of 0.59. The varying results between the subsets can have many reasons. One reason for subset 1 receiving a lower adjusted rand index might be that several of the test case failures which the expert from Qlik identified as distinct problems had similar error messages. The expert identified the failures as different to each other by investigating events in the log files. NIOCAT fails to separate these problems since a deeper log file analysis is not implemented.

One should keep in mind that while Subset 1 was the most challenging subset for NIOCAT it was also the most challenging subset for the baseline approaches to partition into problems. Even though the adjusted rand index for the output from NIOCAT is lower compared to the other subsets, it is still much higher than the baseline approaches, that yield the indices 0.15 and 0. The weight tuning results showed that in order for NIOCAT to achieve the highest accuracy the HTML

should be given lower weight. But even with the HTML weight being lower than the other weights, the vector space model implemented in NIOCAT that uses a combination of the failure components, results in a much more accurate clustering compared to exact string comparisons of individual components in the baseline approaches.

Subset 2, which attained the adjusted rand index of 1.0, had only 11 failures, where most of the failures represented a unique problem. During the manual analysis of this subset we noticed that the problems seemed to be of quite different character to each other and thus, easier to distinguish. It seems that this was also the case for NIOCAT, since 400 different combinations of parameter values yielded an output identical to the reference partition. The failures were simply so different compared to each other that it was easy to separate them. Most of the failures for this subset had very distinct error messages, which in combination with the test case names made it easy for NIOCAT to achieve the perfect partition. But the results from the weight tuning also shows that the perfect partition could be achieved with higher weight on the HTML. Since the problems were distinct, one can imagine that the underlying HTML for the different problems differed enough for NIOCAT to be able to separate the problems based on higher HTML weight.

The output of NIOCAT yielded the adjusted rand index value 0.96 for subset 3 and was thus almost identical to the reference partition. The few test case failures that were incorrectly classified for, were probably because of the same reason as for subset 1. One theory as to why separate problems with similar error messages did not impact the result as much for subset 3 as it did for subset 1, is dependencies on the characteristics of the data set. Subset 3 was extracted from test runs on an individual branch. The branch belongs to a team working exclusively with the user interface, thus resulting in most failures being closely tied to the HTML. Our theory is that the HTML helped distinguish separate problems even though the error messages were similar. This can be confirmed in figure 6.3 as well, the weight of the HTML component is still a rather large factor in the total feature selection for determining similarity between failures in subset 3.

The weight tuning performed for each subset showed that the weight configuration is highly dependent on the data being analysed. Although, we noticed that the threshold value for the combinations yielding the highest rand indices varied from 0.55 to 0.85 for subset 1, 0.6 to 0.95 for subset 2 and 0.65 to 0.7 for subset 3. Thus, this gives us an indication that a threshold with a value between 0.65 and 0.75 is a good starting point.

The weight configuration being highly dependent on the data being analysed has the unfortunate impact that performing analyses on unknown data may not yield partitions of the same quality shown in this study. Although, an analyst using NIOCAT could possibly use its knowledge about the data to tune the parameters accordingly. For example, if an analyst is currently investigating problems that are suspected to be related to the back end, the HTML weight could be turned down, since it probably would not add value to the analysis. On the other hand, if the data to be investigated comes from a team working mostly with front end features, the analyst could choose to turn up the HTML weight. Furthermore, the output from NIOCAT is not intended to be considered as a final reference partition for the analyst. It should serve as a starting point for further investigation. At Qlik,

their automated bug report creation tool does not manage to group all duplicates together and yields many false duplicates but it still a valuable tool for results analysts and team members. As stated by the members of the focus group there is currently no overview, like the one from NIOCAT, available and usage of the tool would thus help in the daily work, even if the clustering accuracy is not perfect. Keep in mind that the way to navigate among failures in Bamboo is based on the test case name approach where the only identifier is the test case name. One can imagine the frustration for an analyst trying to get an overview of current problems, by manually navigating through test case names branch by branch.

7.3 Future Work

While working with this thesis, many thoughts and ideas for further research came up, but due to the time frames, we had to restrict ourselves. Early on we considered taking more information about the failures into account. We were especially interested in analysing the log files. During the focus group, considering the log files came up as a suggestion, as well as considering the test agents. The log files contain a lot of information that could be useful, but requires specific handling. The name of the test agents could be added to the similarity comparison and thus provide the human analyst with more useful information and connections between failures. A test agent that keeps having failing test cases might itself be malfunctioning and thus be the reason why the tests are failing while the software is functioning correctly.

Different methods of measuring similarity between failures could also be further investigated. In this thesis we used a fairly naive implementation of the vector space model which only takes the term frequency into account. There is thus much room for improvement in this area of the project. Only adding consideration of the Inverse Document Frequency (IDF) to the current computations could possibly improve the results. Other similarity measures such as the Okapi BM25 function could possibly yield a different result as well.

Furthermore, it would be interesting to research if considering the order of the terms within the different components could improve the clustering. Perhaps the ordering could be important for some of the components but not necessarily all of them.

The whole solution could even be implemented using a completely different approach, such as machine learning. When we analysed the log files manually, we could see clear patterns for each distinct problem. If a human first classifies a certain amount of errors, perhaps a machine can take over. The downside of this approach is that the machine needs to continuously be recalibrated and thus limit the time savings of the automated tool.

At Qlik, the next step for NIOCAT is deployment, which includes streamlining the whole process, from selecting data to analysing, to visualizing the results in QlikView. After deployment, the real world evaluation can begin. The system can then be used on a daily basis, on real data, by analysts required to make decisions. The usefulness of NIOCAT and the correctness of its partitions will be put to test in a real world environment.

The overall research questions of this thesis is how to help Qlik address the information overload caused by automated testing. To help the results analyst navigate the results we developed NIOCAT, a tool which analyses test result and produces a partition of the failed test cases that can visualised in QlikView. Using QlikView, a user can navigate the analysed test results in a much simpler manner than what was previously possible. The partitioning created by NIOCAT allows a user to quickly discover information such as on what branches a problem is occurring on and how many test runs failed because of a certain problem. Qlik currently uses an automatic bug creation tool that provides a starting point for investigation of failures on the main branch. NIOCAT on the other hand produces a much more accurate partition and enables utilisation of data from both the main and the development branches. The members of the focus group confirmed that with NIOCAT the test results were much easier to interpret compared to current methods.

To be able to create the partitioning of problems NIOCAT uses an analysis based on an information retrieval technique called the vector space model. Even though a somewhat naïve implementation is used, the results shows that NIOCAT manages to produce partitions in line with manual analyses by experts. Regardless of size and character of the input data, NIOCAT outperforms the two baseline approaches by a large margin in regards to partitioning the failures into clusters of problems. Thus, considering a combination of execution data and textual information improved the accuracy of the clustering compared to clustering based on textual information alone.

The participants in the focus group expressed that NIOCAT provides an overview that currently does not exist and they are eager to start using the tool in their daily work. Although there is room for further improvements and enhancements the feedback was exclusively positive and the life of NIOCAT will continue with deployment and real world evaluation.

References

- [1] <http://www.softwarequalitymethods.com/papers/star99%20model%20paper.pdf>. Accessed: 2014-03-19.
- [2] Business discovery: Business intelligence for everyone | qlik. <http://www.qlik.com/>. Accessed: 2014-04-14.
- [3] Continuous integration and build server. <https://www.atlassian.com/software/bamboo>. Accessed: 2014-03-18.
- [4] KDIR 2014 - international conference on knowledge discovery and information retrieval. <http://www.kdir.ic3k.org/>. Accessed: 2014-03-20.
- [5] The vision for qlikview.next. <http://www.qlikview.com/~media/Files/next/The-Vision-for-QlikView-Next.ashx>. Accessed: 2014-03-19.
- [6] IEEE standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84, Dec 1990.
- [7] R. Atem de Carvalho, F. Luiz de Carvalho e Silva, and R. Soares Manhaes. Mapping business process modeling constructs to behavior driven development ubiquitous language. <http://arxiv.org/ftp/arxiv/papers/1006/1006.4892.pdf>. Accessed: 2014-03-19.
- [8] Ricardo A. Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
- [9] Hal Berghel. Cyberspace 2000: Dealing with information overload. *Commun. ACM*, 40(2):19–24, February 1997.
- [10] Elizabeth Bjarnason, Per Runeson, Markus Borg, Michael Unterkalmsteiner, Emelie Engström, Björn Regnell, Giedre Sabaliauskaite, Annabella Loconsole, Tony Gorschek, and Robert Feldt. Challenges and practices in aligning requirements with verification and validation: a case study of six companies. *Empirical Software Engineering Journal*, July 2013.
- [11] Jean-Francois Collard and Ilene Burnstein. *Practical Software Testing*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2002.
- [12] Chris Dickens. Automated testing basics. <http://blogs.msdn.com/b/chappell/archive/2004/04/01/106056.aspx>. Accessed: 2014-03-19.

- [13] Martin J. Eppler and Jeanne Mengis. The concept of information overload: A review of literature from organization science, accounting, marketing, mis, and related disciplines. *Inf. Soc.*, 20(5):325–344, 2004.
- [14] Boris Evelson and N Norman. Topic overview: business intelligence. *Forrester Research*, 2008.
- [15] Robert Feldt. Do system test cases grow old? *ArXiv e-prints*, abs/1310.4989, October 2013.
- [16] P.R. Halmos. *Naive Set Theory*. Undergraduate Texts in Mathematics. Springer, 1960.
- [17] L.G. Hayes. *The Automated Testing Handbook*. Software Testing Institute, 1995.
- [18] Lawrence Hubert and Phipps Arabie. Comparing partitions. *Journal of Classification*, 2(1):193–218, 1985.
- [19] Adam Kolawa and Dorota Huizinga. *Automated Defect Prevention: Best Practices in Software Management*. Wiley-IEEE Computer Society Press, 2007.
- [20] Johannes Lerch and Mira Mezini. Finding duplicates of your yet unwritten bug report. In *Computer Science Master Research*, pages 69–78, 2013.
- [21] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, Cambridge, UK, 2008.
- [22] Glenn W. Milligan and Martha C. Cooper. A study of the comparability of external criteria for hierarchical cluster analysis. *Multivariate Behavioral Research*, 21(4):441–458, 1986.
- [23] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, Hoboken and N.J, 3rd ed edition, 2012.
- [24] Martin P. Robillard, Robert J. Walker, and Thomas Zimmermann. Recommendation systems for software engineering. *IEEE Software*, 27(4):80–86, July 2010.
- [25] Per Runeson, Magnus Alexandersson, and Oskar Nyholm. Detection of duplicate defect reports using natural language processing. In *International Conference on Software Engineering*, pages 499–510. IEEE Computer Society, 2007.
- [26] Per Runeson, Martin Host, Austen Rainer, and Bjorn Regnell. *Case Study Research in Software Engineering*. Wiley Blackwell, 2012.
- [27] Jorge M. Santos and Mark Embrechts. On the use of the adjusted rand index as a metric for evaluating supervised classification. In Cesare Alippi, Marios M. Polycarpou, Christos G. Panayiotou, and Georgios Ellinas, editors, *ICANN (2)*, volume 5769 of *Lecture Notes in Computer Science*, pages 175–184. Springer, 2009.

-
- [28] Edward G. Smith. Automated test results processing. http://www.stickyminds.com/sites/default/files/article/file/2012/XDD2720filelistfilename1_0.pdf. Accessed: 2014-03-17.
- [29] Xiaoyin Wang, Lu Zhang 0023, Tao Xie, John Anvik, and Jiasu Sun. An approach to detecting duplicate bug reports using natural language and execution information. In Wilhelm Schäfer, Matthew B. Dwyer, and Volker Gruhn, editors, *International Conference on Software Engineering*, pages 461–470. ACM, 2008.



LUND
UNIVERSITY

<http://www.eit.lth.se>