Clustering of back-end failures in automated testing

Fredrik Folkesson
dat11ffo@student.lu.se

Johan Nyholm
dat11jny@student.lu.se

August 25, 2016

Master’s thesis work carried out at Qlik.

Supervisors: Emelie Engström, emelie.engstrom@cs.lth.se
Lars Andersson, Lars.Andersson@qlik.com

Examiner: Per Runeson, Per.Runeson@cs.lth.se
Abstract

Automated Software testing is becoming increasingly popular, which in turn creates more information that has to be analyzed. At the software company Qlik a tool called NIOCAT is used to create clusters of failed test cases thought to originate from the same code defect. The clustering is done in order to decrease the ever increasing amount of manual analysis needed to be done with regards to software testing. However, the existing tool currently only clusters by using information from the front-end of the system under test. This makes clusterings harder to create when the code defects which cause the tests to fail are originating from the back-end.

In this thesis we have looked into different types of back-end information and different methods for using this information in order to create clusters of failed test case executions originating from the same code defect. We created a prototype that clusters failed test case executions by analyzing methods names used in requests sent to the server. We did this using the vector space model in which we evaluated multiple approaches for weighting terms. The best approach seemed to be weighting the methods using a suspiciousness rating. The prototype shows great promise of working well at Qlik but further work and research has to be done to be conclusive.

Keywords: clustering, test case, back-end, TF-IDF, suspicious statements, vector space model
Acknowledgements

We would like to thank our supervisors Emelie Engström and Lars Andersson for all their support and guidance during this thesis as well as our examiner Per Runesson for providing feedback after iteration one as well as on the report. We would also like to thank the Qlik employees Nicklas Erman and Tomas Bylander for their help with explaining the NIOCAT tool, the testing process and testing frameworks used at Qlik.
## Contents

1 **Introduction** ................................................................. 7  
1.1 Context ................................................................. 7  
1.2 Problem ................................................................. 8  
1.3 Research Questions ................................................... 8  
1.4 Contribution ........................................................... 9  
1.5 Outline of the Report ................................................. 9  

2 **Background** ................................................................. 11  
2.1 Software Testing ....................................................... 11  
2.2 Document Clustering .................................................. 12  
2.3 Vector Space Model .................................................... 13  
2.4 Adjusted Rand Index ................................................... 14  

3 **Case Description** ...................................................... 17  
3.1 Qlik ................................................................. 17  
3.2 Qlik Sense .............................................................. 18  
3.2.1 Architecture ....................................................... 18  
3.2.2 Communication between the Client and Engine ............. 20  
3.3 Horsie ............................................................... 21  
3.4 ProtocolTester4Net ................................................... 22  
3.5 Running the Tests ..................................................... 23  
3.6 NIOCAT ............................................................... 23  
3.7 Case Summary ......................................................... 24  

4 **Related Work** ............................................................... 25  
4.1 Clustering using Call Stack ......................................... 25  
4.2 Identifying Defects using Code Coverage and Suspicious Statements ......................................................... 26  
4.3 Clustering Test Cases using Control Flow and Markov Models ......................................................... 27  
4.4 Clustering using Multiple Types of Information ............... 27  

5
Chapter 1

Introduction

1.1 Context

Software testing has for a long time been a practice that is considered essential in order to create reliable systems. As a software project grows and its complexity increases, the amount of test cases needed in order to evaluate the software project escalates. Automated testing is therefore widely used to save time and effort. It is used both as regression testing but also to test new features.

Manually analyzing the information from test case executions and linking the test failures to defects in the code takes more and more time when the number of test case executions increases. Switching to automated testing can therefore decrease the amount of time spent testing but heavily increase the amount of time spent analyzing the results of the tests. As multiple failed test cases may actually fail due to the same defect in the code, it makes sense to try to group together executions of test cases into clusters where each test case failure in each cluster is hypothesized to originate from the same defect in the code. The testers and developers would then only have to map each cluster to a defect in the code instead of each individual test case.

By clustering test cases together automatically and not manually, the amount of work needed to pinpoint a test failure to a code defect can be dramatically reduced. The reason for the reduction in work is that the task of linking each individual test case failure to a code defect can be reduced to only linking a smaller amount of clusters, each containing multiple test cases, to their respective defect.

At Qlik, a large software company where this thesis was conducted, the need to handle the large amount of information generated from running automated tests has resulted in
a tool with the purpose of doing just that, clustering test cases together. The tool, NIOCAT, short for Navigating Information Overload Caused by Automated Testing, clusters failed test cases that the tool assumes failed due to the same defect in the code. It does this by using the test case name, the HTML element last interacted with and the error message from the test case. The software that the NIOCAT tool monitors is called Qlik Sense, it is a business intelligence service intended for navigating, managing and visualizing large sets of information. Qlik Sense has a tiered architecture with a front-end layer that communicates via a websocket with a back-end layer where the heavy calculations are done. Calculations are requested by the front-end after which they are executed by the back-end and the results are returned to the front-end where they are in turn visualized and presented it such a way as to make the data easy to interpret and work with.

1.2 Problem

At Qlik, NIOCAT has been a great asset and has made it easier to handle defects in front-end code. However when the test case failure is not the manifestation of a front-end defect, but instead a bug on the back-end, the NIOCAT tool is not as precise when it comes to clustering failed test cases since the information from the last interacted HTML element is often not very useful and the tool can only really use the information from test name and the error message to create the clustering of the failed test cases.

In this thesis we investigate ways to improve the clustering of failed test cases when the defect in the code is in the back-end. We are trying to find which information would be useful to improve clustering of back-end failures at Qlik and if that information actually is attainable from the failed test cases or if we can modify the test execution to be able to attain it. We are also looking at different ways to cluster the test failures using this information and building a prototype that uses our findings. Finally we compare clusterings made by our prototype tool to those made by the current NIOCAT tool.

1.3 Research Questions

In this thesis we aim to answer the following questions:

RQ1 Which back-end information can be used to improve precision when clustering test case executions? - Find out which back-end information that has been used or could potentially be used in order to cluster test executions.

RQ2 Which types of back-end information can be made accessible after a test execution? - Based on the answer to the first research question, which types of information can reasonably be made accessible considering the amount of work needed to extract them, mainly focusing on the context of Qlik.

RQ3 How can this back-end information be used to improve clustering precision? - Find out how the information can be used when clustering and how it is best used in
order to create the most useful clusterings of failed test cases.

1.4 Contribution

We worked tightly together during this thesis and all major design and other decisions were taken together. We created and evaluated the clustering prototype as well as wrote the report together. In the report Johan created most of the pictures while Fredrik created most of the tables. Johan created a program which calculates the adjusted rand index between a given optimal clustering and an actual clustering. Fredrik added the logging to the proxy layer of Qlik Sense to save the web traffic between the client and the engine. Johan made modifications to the prototype to enable it to run several different configurations in serial. Fredrik created most of the defects in Qlik Sense that was used to generate our different test sets.

1.5 Outline of the Report

The report is divided into seven chapters. Chapter 1 provides a brief introduction to the thesis. Chapter 2 defines terms and explains concepts and methods which will be used in the thesis. Chapter 3 describes the company Qlik and their working process. Chapter 4 covers relevant work and research which has been done regarding clustering code executions. Chapter 5 describes the methodology we used to answer our research questions. Chapter 6 describes how we built and evaluated the prototype. Finally chapter 7 gives our conclusions from the thesis.
1. Introduction
Chapter 2

Background

In this chapter we briefly describe terms and concepts which are mentioned and expanded on later in the report. We first describe software testing, then we describe clustering and how the vector space model can be used to measure similarity between text documents. The chapter finishes by describing the adjusted Rand index and how it can be used to measure similarity between different clusterings.

2.1 Software Testing

Software testing is a great and often necessary way of assisting in the process of making sure that a software system functions as expected. When working on a large and complex software project with extensive requirements, testing so that new changes to the code base does not break old functionality is important. The practice of writing tests with the purpose of preventing functionality regression is fittingly called regression testing[16].

Regression tests should optimally be run every time the code base changes, such as at every new commit. Doing testing manually after every commit would for a large project be very much work and almost practically impossible. Therefore, automated test executions are used instead [12]. Automated test executions are run automatically by a program or automated testing framework, not manually by a tester. Each test case that are used for automated test execution is usually specified so that it does some action on the system under test after which it compares the result of the action with the expected result. The testers and developers can then see if any of the automated test executions have failed and in such a case they may fix the problem and then run the tests again.

Automated regression test execution can be done at all test levels but in this thesis we
2. Background

will focus on analyzing automated regression tests at the system test level \[8\]. The system testing level is a testing level which purpose is to verify that a complete integrated system is meeting its specified requirements. In comparison the unit testing level tests small units of code like a function or a class and the integration test level tests the integration of these small units. System testing is usually abstracted from the actual implementation, as it tests the system as a whole and does not regard how the system works but only if, and how well, it works. Because the implementation is not considered when running the tests, system testing is a form of black box testing \[8\]. System testing aims to test for flaws in a systems functionality but also for qualitative flaws in areas such as security, performance, reliability and usability \[8\]. A typical system test case may, for example, replicates the behavior of an intended end-user \[22\]. When testing a product, system testing is usually the last level of testing before the acceptance test level.

The practice of automated test execution is in many cases necessary for a software project to be successful. Alessandro Orso et al. claims that when working with continuous integration \[13\], where the code base changes occur frequently, testing must be automated \[26\]. In the same paper they state that “In more general terms, automated testing, CI, and other related practices provide clear evidence of how the testing process has improved in practice.”

A problem with automated test execution compared to manual testing is that much more information is generated since the tests run much more often compared to when manual testing is used. A result of this may be that much time is needed in order to link each failed test case to the code defect which it originated from. One of the solutions to this problem is to automatically create clusters of failed test cases where every cluster contains test case failures thought to be from the same code defect \[39\].

2.2 Document Clustering

Document clustering is the process of grouping different documents or texts together into clusters in such a way so that the documents in the same cluster are more similar to each other according to some measurement than the documents in the other clusters. When clustering documents a similarity measure is usually used to compare the similarity between the documents. The measure may for example be used such that documents that are more similar, according to the similarity measure, than a chosen threshold are clustered together. This similarity measure can be calculated using different features of the documents. The similarity measure may for example be the percentage of words shared between the documents. A big benefit of clustering is that is can reveal trends in the information that would have been hard to find otherwise \[4\].
2.3 Vector Space Model

One way to compare textual documents is to model them using the vector space model [35]. After which the similarity measure between the documents can be calculated with the use of cosine similarity. The vector space model is used for modeling documents as vectors in a multidimensional room. Each vector represents the terms used in its corresponding document and each term-type will be a dimension in the room.

A document may for example be the following string: “xyz”. In the case where the letters are used as terms, the vector modeling this document would have a non-zero value for the ’x’, ’y’ and ’z’ dimensions. The document containing only the string: “xz” would in turn have non-zero values for the ’x’ and ’z’ dimensions but a zero value for the ’y’ dimension, see Figure 2.1. The vector space model is one of the most common ways to model documents in information retrieval [6].

There are different ways in which the terms can be weighted. The terms can for example be weighed using Term Frequency (TF) [31]. Using TF, the vector-values for each dimension scale based on the number of times the term occurs in the document. As such, “xxy” would have a different vector value in the ’x’ dimension than the document “xy” would.

There are multiple approaches which can be taken when calculating term frequency. One such approach is to use Boolean frequency, where a terms frequency is weighted as one if...
2. Background

it exists and 0 otherwise. Another type is raw frequency, essentially counting the number of occurrences a term has in a document, this frequency will be referred to as $r_{f,t,d}$. A third approach is to use augmented term frequency. The benefit of using augmented frequency is that it eliminates the bias towards longer documents. This is done by dividing the raw frequency with the largest raw frequency for any term in the document. The following equation defines augmented frequency:

$$t_{f,t,d} = 0.5 + 0.5 * \frac{r_{f,t,d}}{\max\{r_{f,t',d} : t' \in d\}}$$

Equation (2.1)

Another approach when weighting the terms is to use the Term Frequency - Inverse Document Frequency (TF-IDF) method [31]. In a paper by Runeson et al. analyzing 79 different publications TF-IDF is shown to be the most common weighting scheme by far [6]. When using TF-IDF, each term is assigned a certain weight where the weight is intended to reflect how important the term is. The weight increases proportionally to the number of times the term is used in the document, but decreases based on the frequency of the term in the corpus. The weighted vector for document $d$ in the vector space model, where TF-IDF is used for weighting is defined a such:

$$v_d = [w_{1,d}, w_{2,d}, ..., w_{N,d}]$$

Equation (2.2)

Where

$$w_{t,d} = t_{f,t,d} * \log \frac{|D|}{||d' \in D| t \in d'||}$$

Equation (2.3)

and $t_{f,t,d}$ is the term frequency for term $t$ in document $d$. The remaining factor in the equation is the IDF, where $|D|$ is the number of documents in the document set and $||d' \in D| t \in d'||$ is the number of documents that contains the term $t$.

In order to calculate the similarity between documents modeled using the vector space model, the Cosine Similarity measure can be used [4]. It is essentially the cosine value of the smallest angle between the two vectors being compared. The cosine similarity between two vectors, $u$ and $v$ can be calculated with the following formula:

$$\text{cosine\_similarity} = \cos(\theta_{u,v}) = \frac{u \ast v}{||u|| \ast ||v||}$$

Equation (2.4)

$\theta_{u,v}$ is the angle between the vector $u$ and $v$.

Assuming no negative weights are used for modeling documents in the vector space model, the cosine similarity between two vectors will always have a value in the interval 0 and 1. As the cosine value for the angle 0 is defined as 1, a higher cosine similarity value indicates a higher similarity between the vectors, and in turn, a higher similarity between the documents modeled by the vectors. Cosine similarity is the most common way to measure similarity when using the vector space model [6].

2.4 Adjusted Rand Index

Rand index is a measure which is used to determine the similarity between two different clusterings of the same data set. It can essentially be explained as the ratio between the
2.4 Adjusted Rand Index

Figure 2.2: An example of a control clustering (C) and one arbitrary clustering (A). In the context of Rand index calculation, the pair (a,b) is a true positive, (a,d) is a true negative, (c,d) is a false negative and (a,c) is a false positive.

agreements between the clusterings and the total number of ways to pick two elements among the total number of elements.

Using the Rand index algorithm, it is possible to evaluate how close an arbitrary clustering is to an optimal control clustering. The Rand index algorithm for comparing an arbitrary clustering to a control clustering is defined as such:

Given a set of \( n \) elements \( S = \{e_1, e_2, ..., e_n\} \) and two clusterings of those elements, the control clustering \( C = \{C_1, C_2, ..., C_r\} \) and an arbitrary clustering \( A = \{A_1, A_2, ..., A_s\} \) such that clustering \( C \) has clustered the \( n \) elements into \( r \) clusters while \( A \) has clustered the same elements into \( s \) clusters.

The Rand index can now be calculated as

\[
RI = \frac{tp + tn}{tp + tn + fp + fn}
\]  

(2.5)

Where:

- **tp (True Positives)** is the number of element pairs in \( S \) that are correctly clustered together in \( A \). Meaning those that are clustered together in both \( A \) and \( C \).

- **tn (True Negatives)** is the number of element pairs in \( S \) that are correctly not clustered together in \( A \). Meaning those that are not clustered together in neither \( A \) nor \( C \).

- **fn (False Negatives)** is the number of element pairs in \( S \) that are incorrectly not clustered together in \( A \). Meaning those that are clustered together in \( C \) but not in \( A \).

- **fp (False Positives)** is the number of element pairs in \( S \) that are incorrectly clustered together in \( A \). Meaning those that are not clustered together in \( C \) but that are clustered together in \( A \).

The Rand index always has a value between 0 and 1 where the value 1 represents perfect similarity between the clusterings. An example of true positives, true negatives, false positives and false negatives, is shown in Figure 2.2.

A critic of using Rand index for measuring similarity between partitions or clusterings is that when comparing two random clusterings, the similarity value may vary, and is
2. Background

often far from 0. In order to address this problem, a different measure, the adjusted Rand index, exists. The adjusted Rand index measure compares the similarity between two clusterings to that of two, assuming hypergeometric distribution of randomness, random clusterings.

The adjusted Rand index can be calculated using the following formula:

\[
ARI = \frac{ab - c}{a^2 - c}
\]  

(2.6)

\[
a = tp + tn + fp + fn
\]  

(2.7)

\[
b = tp + fp
\]  

(2.8)

\[
c = (tp + fn)(tp + fp) + (fp + tn)(fn + tn)
\]  

(2.9)

The Adjusted Rand Index has an expected value of 0, which corresponds to the adjusted Rand index of two random clusterings. The value of the adjusted Rand index may, in contrast to the regular Rand index, be negative, which corresponds to two clusterings which are more dissimilar than two random clusterings. J.M. Santos and M. Embrechts concludes in a report that adjusted Rand Index is a valid measure for evaluating classification algorithms, especially for multi-class data sets [37]. Also, according to Milligan and Cooper, when comparing clusterings which contain different numbers of clusters, ARI is the best suited index [25].
Chapter 3

Case Description

We have conducted an exploratory case study at Qlik [36]. In this case description we first give a brief description of Qlik as a company and then we describe the product Qlik Sense. We also provide a description of the tools and processes used to run the automatic testing at Qlik, as well as those used to analyze the test results.

3.1 Qlik

Qlik was founded in Sweden in 1993. The company is operating in the Business Intelligence area developing two products, QlikView and Qlik Sense that helps companies and people to interpret large amounts of information by transforming and visualizing it in such a way that it is easy to analyze, which in turn will help users make informed business decisions.

In this thesis we have focused on the testing of Qlik Sense which is the latest of Qlik’s core products and it is the clustering of failed tests, aimed at testing that product that is the focus in this thesis. However most of the approaches we evaluate could also be applied when clustering test cases from QlikView and other software which is based on layered architectures such that communication between a client and a server can be recorded.

Qlik has more than 33 000 customers in over 100 countries worldwide and are considered to be a market leader in Business Intelligence software [33].
3. Case Description

3.2 Qlik Sense

Qlik Sense is a product developed by Qlik. It aims at making it easy to manage and visualize large amounts of information. Qlik Sense gives the user an ability to create personalized visualizations and dashboards from multiple different information sources using a simple drag and drop interface. The product comes in different versions, a desktop version, letting the user run the software locally, and a server based version which allows for remote access. An example of Qlik Sense running can be seen in Figure 3.1.

3.2.1 Architecture

The server version of Qlik Sense is essentially built in three layers, the client layer, the application layer and the data layer. A simplified view of the architecture and the communication between the layers can be seen in Figure 3.2.

The desktop version of Qlik Sense is a bit simpler with the app layer removed and the client communicating directly with the data layer. A simplified view of the desktop version architecture can be seen in Figure 3.3.
3.2 Qlik Sense

The data layer consists of two components, a repository and an engine. In this report, we will not focus on the repository component of Qlik Sense, therefore it will not be covered further in this report.

The client layer covers the application that runs on the client side. It is a webclient built

---

**Figure 3.2:** The Qlik Sense server version architecture.

**Figure 3.3:** The Qlik Sense desktop version architecture.
using HTML5/JavaScript/CSS3. The webclient makes requests through REST and WebSockets in order to communicate with the Qlik Sense back-end. In the server version of Qlik Sense, all communication between the client and the data layer goes through the app layer, which among other things acts as a proxy for the communication between the front-end and the back-end. The desktop version differs in that the communication goes directly between the client and data layer.

The client is run through a web browser, either the custom browser which is specifically built for Qlik Sense and which comes bundled with the software, or a standard internet browser like Internet Explorer or Google Chrome. The application layer is built using C# while the core engine is built using mostly C++ and some C. The engine is shared between both of the products, Qlik Sense and QlikView, and is where the programs main computations take place.

3.2.2 Communication between the Client and Engine

Communication between the client and the engine works differently for the server and the desktop version. In the server version, the client initiates the communication by doing HTTP REST calls to the proxy which then connects the client directly to the engine via a websocket. In the desktop version there is no proxy in the middle, instead the client initiates communication by directly connecting to the engine and establishing a websocket connection.

When the websocket between the client and engine is connected, the communication is conducted using an engine API [29]. The API defines a large set of commands that can be requested of the engine, the signature of these as well as the signature of the response from the engine. The engine API uses a protocol called JSON-RPC [15], as defined by the protocol, JSON is the message format used for requests and responses to and from the engine.

The main parts of a typical engine request consists of a method name, a set of parameters to the method, and also some other fields, including a message-id, a handle defining which object the method should be applied to and a protocol-version number. The method name has to be defined in the API and the parameters can vary in size and complexity, from a short string to a dictionary containing large matrices of data or there may simply be no parameters. The names of the request methods, defined by the API, are camel cased.

After a request in the form of a query is received by the engine, the method specified in the request is executed. The response from the engine, back to the client, consists of the result of the method. After the client receives the response, it updates its state and view based on the response, e.g. changes the visuals of a graph due to the result of a calculation made by the engine. A typical Engine API request and response can be seen in the figures 3.4 and 3.5 respectively.
3.3 Horsie

Horsie is the name of an automated test execution framework intended for setting up and running system tests on QlikView and Qlik Sense, both the desktop and the server versions. The tool is developed internally at Qlik and allows each separate developer-team to create and manage their own system tests.

The tests are specified in a language called SpecFlow [2] which allows users to write the tests in a structured natural language. Horsie interprets these tests and drives QlikView or Qlik Sense by acting as a user and sending commands such as clicks and key presses to the program under test and reacting to the responses. Some parts of the testing such as set-up is not done by simulating a user but instead via direct access to the QlikView and Qlik Sense API. Since Horsie controls the software as a user would, it is possible to use it for running automated acceptance and system tests. The relationship between a test case, Horsie and Qlik Sense can be seen in figure 3.6. After the tests have been run, information about the execution for each specific test, such as logs and images is gathered and stored in a database.
3.4 ProtocolTester4Net

The Qlik Sense engine has its own automated test framework called ProtocolTester4Net. The tests run by this framework are only focused on testing the Qlik Sense engine API and are not full system tests. As such they are not dependent on any other layers than the data layer. The tests are of the functional type and are run by generating engine API traffic between the engine using websocket communication. The tests are not linked to the system testing framework Horsie, but run separately during the testing phase. Still, tests are specified using the language SpecFlow, which allows for human readable syntax. After an Engine API test execution has finished, information about the test execution is, similar to how Horsie operates, collected and stored.
3.5 Running the Tests

At Qlik, the process of building and testing the software is done in multiple steps. The processes are automated using the tool Jenkins\(^\text{[18]}\). Jenkins is an automation server tool with which specific projects can be set up with the purpose of for example building a software product or testing one.

The build process of Qlik Sense is set up such that one Jenkins-project builds the engine, a second project builds the Qlik Sense system as a whole, using the previously built engine, and a third Jenkins project initiates a testing phase. Both Horsie and the ProtocolTester4Net tests are run during the testing phase. After the testing phase is finished, the results from the test execution and all and the test related information can be accessed via a common database.

3.6 NIOCAT

NIOCAT is the tool currently used at Qlik for clustering failed test cases together. Its purpose is to make it possible to navigate the large amount of information generated by test case failures resulting from automated testing of multiple branches and builds of the Qlik Sense software.

By grouping failed test cases together in clusters with the intent that each cluster should represent a certain defect in the code base, NIOCAT is able to diminish the workload required when identifying defects in the code base. NIOCAT uses front-end information gathered during test execution in order to create the clusters.

The NIOCAT tool clusters the executed failed test cases based on three metrics: the name of the test case, the message generated as a result of the test case failing and the HTML element last interacted with by the framework before the test case failure occurred. The values of these three metrics are then compared among failed test case executions in order to create clusters.

In NIOCAT, the similarity between two test case executions is defined as the average similarity between the values for the three metrics. The similarity for each metric is calculated in the Vector Space Model using cosine similarity. A threshold defining how similar test case failures has to be in order to be clustered together can be set. This value has to be set between 0 and 1.0 where 1.0 represents exact similarity. The default setting is 0.8.

One of the weaknesses of NIOCAT is that is is not using any back-end information, thus test failures where the failure originates from the back-end code is not clustered correctly nearly as often as when the defect is located in the front-end code. Another problem is that NIOCAT can not cluster tests together when the tests belong and are executed by different test frameworks. E.g. if a bug in the code makes both a test case executed by Horsie and another test case executed by ProtocolTester4Net fail they can not be clustered together by NIOCAT.
3. Case Description

The NIOCAT tool is widely used at Qlik and has resulted in a simpler debugging process. The tool is the result of a master thesis done in 2014 by Vanja Tufvesson and Nicklas Erman [39] [14]. NIOCAT is developed in the Scala language [1].

3.7 Case Summary

At Qlik, automated tests are run multiple times every day in order to test Qlik’s products. These tests are run by multiple different test frameworks such as Horsie and ProtocolTester4Net. The result of the test executions are analyzed by the tool NIOCAT which, using front-end information, groups test case failures into clusters, where each cluster is thought to contain test cases that have failed due to the same code defect. The clustering process works very well when the failure originates from a front-end defect. But when the defect is located in the back-end code, the clustering is not as good, since the tool only clusters based on front-end information. There are also currently no way to cluster test cases together which are executed by different testing frameworks.
Chapter 4

Related Work

This chapter provides a summary of previous work related to the research questions we tried to answer in this thesis. There has been a number of attempts at evaluating approaches for clustering execution failures of different types using back-end data.

4.1 Clustering using Call Stack

Lerch and Mezini presents a set of approaches for detecting and classifying duplicates among bug reports which were evaluated [23]. One type of approach was the to create the classification using stack traces resulting from crashes. Three main attributes were extracted from each stack trace: the exception type, the exception message and the method calls. Three different approaches of detecting duplicates were then evaluated, one for each attribute type. The approaches each used TF-IDF. Through experiments they established that the method calls was the superior attribute of the three to use when determining if two bug reports were duplicates. In addition, the impact of varying the depth of the stack trace was investigated, essentially meaning how many method calls that were used when comparing bug reports. According to experiments, the top 20 method calls were those that had the most impact on classifying the crashes but the authors still recommended the use of the full stack in their approach as each additional method call did still improve the clustering result.

In 2012, Yingnong Dang et al. evaluated a method for clustering crash reports generated from different Microsoft products [10]. The evaluated approach focuses on analyzing and comparing the call stack generated from the crash reports. One part of the algorithm focuses on comparing the distance and location of common elements in the call stacks and their relative position to the top of the stack. They worked based on the assumption that
the top of the call stack is more important than the bottom part, which is backed up by the results of Lerch and Mezinis experiments [23].

The problems that the two reports discuss and addresses are somewhat different from what we address in this master thesis. In the report, a call stack is generated from the failed execution, the top of the stack is thus most probably where the failure manifests itself. A failed test case is different from a crash, as for a failed test case there might not be an accessible call stack. In addition, even if you have an accessible call stack after a failed test execution, the defect which caused the test to fail could manifest itself anywhere in the call stack since the execution of the test does not end immediately when the failure occurs as it does when a program crashes.

### 4.2 Identifying Defects using Code Coverage and Suspicious Statements

A method for clustering using code coverage is described by DiGiuseppe and Jones [11]. They refer to the method as Concept-based Clustering. The clustering approach focused on first recording and parsing all executed lines of code for each test case, this is then referred to the execution profile. All word and their frequencies in each execution profile are then stored. The words are acquired by first collecting all variable names and method calls and thereafter split each word into smaller words such that `getMethodName` is split into three words `get`, `method` and `name`. The frequencies of the words are then used as metrics in order to calculate the similarity between executions. The words are weighted using TF-IDF or Term Frequency - Inverse Document Frequency [31]. The clusters are then created based on the similarities between execution failures.

In another paper Jones et al. presents a technique for visually highlighting potentially defect parts of a code base by using something called suspicious statements [19]. In the paper it is described how the code coverage, essentially consisting of all statements executed by each test in a set of tests are stored. After all tests in the set have been executed, two values are calculated for each executed statement, the number of passed tests that executed the statement and the number of failed tests that executed the statement. The suspiciousness value for each statement is then calculated using the following equation, called the Jaccard equation:

\[
\text{JaccardSuspiciousness}(s) = \frac{\#\text{failed}(s)}{\#\text{failed}(s) + \#\text{passed}(s)} \tag{4.1}
\]

In the paper, the value is translated into a color intended to represent how likely it is that the statement contains a fault. Resulting in suspicious statements being colored red while non suspicious statements are colored green.

Six years later, in his dissertation Jones compares different techniques for locating faults in code. In it, the Jaccard equation is compared to the Ochiai equation [20]. The Ochiai
equation is similar to the Jaccard equation but also uses a third value, the total amount of failed test cases. It is defined as such:

\[
Ochiai_{\text{Suspiciousness}}(s) = \frac{\#\text{failed}(s)}{\sqrt{\text{totalFails} \ast (\#\text{failed}(s) + \#\text{passed}(s))}}
\] (4.2)

In the study, the techniques performed similarly but according to another study called Tester Feedback Driven Fault Localization, the Ochiai equation seemed to be superior at finding suspicious statements when only one single fault exists in the code [5].

### 4.3 Clustering Test Cases using Control Flow and Markov Models

The complete control flow of a program execution can be seen as an extension of code coverage. Whereas a program execution's code coverage contains which statements that have been executed, its control flow can be described as the complete set of statements executed and the order in which they were executed.

Bowring et al. describes how control flow can be used to cluster software behavior so that program executions can be labeled with labels such as "pass" and "fail" [7]. The approach in the thesis is based on training a classifier using a set of executed test cases. This is done by first creating a Markov model for each test case execution. Assuming \( n \) test case executions are to be used to train the classifier, \( n \) clusters, each represented by a Markov model of the test case execution and a label, fail or pass, are initially formed. Then, for all clusters which have the same label, the two clusters whose Markov models are most similar, determined using Hamming distance, are merged into one cluster. When two clusters are merged, their Markov chains are merged such that a new Markov chain is formed which reflects the two. Then this process is continued until the patterns established in the current iteration deviates too much from the previous, according to some threshold, or until only two clusters remain. The trained classifier is then used to label other program executions. When determining if a program execution should be labeled as a pass or a fail each of the resulting clusters are used to calculate a probability score for how likely it is that the execution's control flow would occur based on the clusters Markov model. The execution is then given the label of the cluster for which the highest probability was calculated.

### 4.4 Clustering using Multiple Types of Information

Podgurski et al. proposes an approach for automatically clustering software crashes based on the defect they originated from by using a combination of multiple types of execution information [27].
A program execution may have all kinds of different types of features such as: the program counter value, the call stack, input values, the data flow, event sequences, def-use chains or a manually written message. The approach the authors proposed in 2003 is to first find the most useful features for clustering failures originating from the same causes together. Their process for identifying these features is based on the hypothesis that "the profile features that are most relevant to classifying failures according to their causes are the features that are most useful for distinguishing reported failures from successful executions".

Following the hypothesis, a number of failed program executions are grouped together with a set of successful executions in order to create test sets. These test sets are then used to train classifiers to distinguish failures from successful executions by iteratively using random subsets of the available feature types. The features used in the classifier that performed best at distinguishing failures from successful executions are then the features that are used when performing cluster analysis to group together failures that are thought to come from the same code defect. When clustering failures using these features with the clustering algorithm clara based on the k-medoids clustering criterion a majority of the failures in each cluster appears to originate from the same fault [21].

4.5 Clustering using Database Operations and Alterations

In a paper from 2016, Erik Rogstad et al. describes an approach of how test cases, designed for regression testing a system which communicates with a database, can be clustered together according to the deviations which cause them to fail [34]. In the paper, run time database changes is used in order to cluster the test case executions. In addition, the paper also investigates how the use of test case specifications can be used in the clustering process.

The approach is such that for each test case, the difference in changes to the database between an older version and the current version of a software are stored. The set of deviations are then considered the output of the testing. The deviations are in the form of tables and columns, along with the new and old values of the fields that have changed. The SQL operations which cause the deviations are also stored and used in the clustering process, the operation may either be INSERT, UPDATE or DELETE. The clustering is then performed using the probabilistic Expectation Maximization algorithm. In the report, the impact of using different combinations of types of information as input to the algorithm were investigated. The algorithm was evaluated on four test sets where different input combinations were evaluated. Columns and operations was found to be the combination of information types which resulted in the best clusterings. The variant of the approach where columns and operations were used as input did produce perfect clustering in two out of four cases and produced good results in the remaining two.
Chapter 5

Method

This chapter gives a brief description of the methodology we used in order to go about answering our research questions.

5.1 Process Structure

We decided to take an iterative approach to the work in this thesis. Initially we went through a set up phase, in which we focused on setting up the testing environment and getting an understanding of the Qlik Sense software. Then we went through the main phase, where we found and tested different approaches to clustering back-end failures, this phase was conducted in two iterations. This was followed by a conclusion phase in which we further reflected on the result generated by the main phase, how future work could expand on the solution and how the approaches could more concretely be deployed in order to improve the software testing process as Qlik (see figure 5.1).

In addition to the iterations, a lot of additional overhead work was conducted, such as time spent understanding the environment of different systems used during the Qlik Sense development process. These additional challenges were spread out and dealt with during and in between different stages of the work process.

5.2 Iterative Phase

We designed the prototype in two iterations where we at the end of each iteration would evaluate how well the prototype performed clustering. The first iteration included building
5. Method

Figure 5.1: The process used to reach the goal of the thesis.

a prototype in order to test an approach and the second iteration expanded the prototype, introducing an additional new approach for clustering. Each approach was either be to based on incorporating new information or by changing the algorithm by which the information was utilized, or both. After the first iterations we reflected on what would be the focus of the literature study in the second iteration. Figure 5.2 depicts the four main steps involved in each iteration cycle.

Figure 5.2: The steps involved in each iteration.

The four main steps were the following:

1. **Literature Study** - Each iteration initially began with a literature study aiming at identifying methods and information which could improve the clustering precision further. The main purpose of this step was to find answers to the first research question "Which back-end information can be used to improvise precision when clustering test case executions?" but also to find different ways to utilize this information. Each study resulted in a set of approaches which could potentially be used to cluster test executions with good performance. Each of them requiring some type of information from the test executions. During the literature studies, we searched for literature using multiple publication databases, such as ACM Digital Library [3] and IEEE Explore Digital Library [17], using the Lund University library digital archive search tool LUBsearch [24]. We initially started by searching for articles using relevant keywords and then we inspected the references of the relevant papers that we found in order to find even more relevant articles.

2. **Check Extractability** - Following the literature study, the next step was investigating which of the approaches that were actually applicable and promising, and for which
the required type of information was available or extractable from Qlik Sense test executions. This was done by talking to employees at Qlik and investigating Qlik Senses code base to see if the information was extractable. The purpose of this step was to answer the second research question "Which types of back-end information can be made accessible after a test execution?". The result of this step was a set of approaches which was to be implemented and tested in the upcoming version of the prototype.

3. Develop Prototype - In this phase we developed the prototype for this iteration based on the approaches we decided on in the previous step. The purpose of the prototype is to enable us to easily and automatically create clusterings so that we could later evaluate how well the different approaches performed clustering. We developed the prototype in such a way that variables in the clustering approach were easily configurable in a settings file and so that we could run several configurations of the prototype automatically in sequence. To create the prototype we also extracted and processed the relevant types of information from Qlik Sense and the testing frameworks needed by the approaches.

4. Evaluate Prototype - We then evaluated the prototype by first creating test sets, consisting of code defects which triggered test case failures. We created test sets by fault seeding the Qlik Sense code base [8]. We also searched for actual commits containing defects created in the development of Qlik Sense to use as test sets. After creating our test sets, we evaluated the prototype by running it on the test sets, systematically varying the parameter settings and thereafter evaluating the produced clusterings using the adjusted Rand index but also with subjective evaluations by employees at Qlik. We also compared the prototype to how NIOCAT performs given the same test set. All our test sets were manually annotated so we knew the correct clustering for each test set. The purpose of this step was to find answers to the third research question "How can this back-end information be used to improve clustering precision?".
5. Method
Chapter 6

Building and Evaluating the Prototype

In this chapter we describe the work done in our two iterations in which we researched and built the prototype for clustering failed test cases using information originating from the back-end. We also describe how we evaluated the prototype and the results of the evaluation after each iteration as well as our reflections on the results.

6.1 Iteration #1 Client-server Traffic

In the first iteration we first researched different types of information and techniques which could be used to cluster test cases based on back-end data. We then built a first prototype for clustering using the names of the requested methods found in the client-server traffic. The method names were represented as vectors in a vector space model where terms were weighted using term frequency. The similarity between the vectors were then used as a metric to cluster together test cases. This prototype was then tested and evaluated against a small test set. Afterwards we reflected on the results of the evaluation.

6.1.1 Literature Study

To get a grip of how to utilize back-end data to cluster failed test case executions, originating from the same defect, we initially conducted a broad literature study to investigate which methods and types of information that had previously been used to cluster program executions and failed test cases. The focus of the study was broad in the sense that it focused not just on clustering failed test cases, but also on clustering more general code executions such as crashes and bug reports.
After doing the literature study and having multiple discussions and meetings with employees at Qlik, we had a set of ideas about what kind of information we wanted to investigate and whether or not we could access this kind of information.

The types of information sources that was used for clustering that we found while doing the literature study and having discussions with Qlik employees included:

- **Call Stack** In many of the reports we read where program crashes were clustered, the clustering approaches rely on the existence of a stack trace or some form of call stack [23] [10]. In the literature we found multiple different approaches that was used to extract information and patterns from the call stack. The approaches which we found the most interesting both focused on utilizing the method calls contained in the stacks.

- **Code Coverage** We also found examples of how the approach of analyzing and comparing code coverage between different program executions was used to cluster executions of test cases [11] [40].

- **Exceptions** Before we started with the literature study, we had discussed the possibility to cluster test cases using exceptions. We found a report presenting such a approach. The authors attempted to cluster program crashes using the exceptions generated during the crash [23]. In the report, both the exception message and the exception type was used for clustering. However, the result of using exceptions was not very successful at clustering crashes. The report also evaluated clustering using method calls found in stack traces and found those method calls to be a more reliable type of information to use when clustering.

- **Client-server Traffic** After talking to testing experts at Qlik, we found out that in the specific case of Qlik Sense, the traffic between the client and the back-end could potentially be extracted. Erik Rogstad et al. presents an approach where database operations in the form of SQL commands are used to cluster test case executions, testing database systems [34]. In the report, only three operations, *INSERT*, *UPDATE* and *DELETE* were used. In the case where the database is seen as the server, the database operations may thus be thought on as requests sent from the client, this was the closest thing to using client-server traffic for clustering that we found in the literature that we encountered.

### 6.1.2 Check Extractability

The challenge of extracting relevant information was harder than we initially expected. In order to find out which information we could actually access in the case of Qlik Sense, we talked to experts on the testing framework Horsie as well as experts on the back-end of Qlik Sense. We found that there were currently no functionality to generate code coverage information from the back-end at a per test case basis. Neither was there any functionality to extract call stack information during the test executions and the implementation of generating code coverage or extracting call stack data would be out of scope of this thesis. According to developers at Qlik, extracting exception information would potentially...
be possible. The use of exceptions was not a very successful approach according to the literature, still we investigated the possibility of using it for clustering but found that few test failures occurred due to crashes and exception information would therefore only be present in a small subset of the test failures we would cluster. Instead we started analyzing the information accessible to us after a typical Horsie test.

After analyzing log files generated from Horsie tests, in collaboration with our supervisor at Qlik, we found that the requests and responses between the client and the server were logged. As such, client-server traffic was what we decided to utilize in the first iteration, more specifically the traffic between the client and the Qlik Sense engine.

We soon realized that the communication between the client and the engine was only logged during the set-up phase of the tests. When the test was actually executed, the client-engine traffic generated from front-end to back-end was not logged. This was because during the set-up phase Horsie used a separate module to communicate directly with the engine bypassing the client.

As the communication between the client and the engine during the actual testing phase goes through the proxy, we decided to investigate if we could log all traffic passing through the proxy. We therefore talked to an expert on the proxy part of Qlik Sense and found out that it should be possible to do so. He also pointed us in the right direction regarding how we could log the relevant traffic.

6.1.3 Developing the Prototype

In order to log the client server traffic we implemented a change in the proxy logging process such that all client-engine traffic was logged. The result of this was that all Horsie tests, testing Qlik Sense server version had additional client-engine traffic information available which could be used for clustering.

Now that we had access to the traffic via the Horsie logs we manually analyzed the requests and the responses sent between the client and the engine for different test cases in order to find out how we more specifically could utilize this information. We decided to only focus on the requests which were sent from the client to the engine. This was due to a number of reasons. The response message corresponding to a given request was usually very large and often consisted of multiple nested value which made responses harder to interpret. In addition, if a request generates an execution error on the engine side, a response may simply not be sent. As such we wanted to instead extract information from the requests.

Each request message can somewhat simplified be said to consist of only a method name and a list of parameters. The parameter list may contain values which triggers a failure on the back-end but the parameters are all linked to the specific method in the request. We chose to only use the method name in order to simplify the development process and because of the report by Johannes Lerch et al. where method calls were identified as the most significant part of the stack trace when identifying duplicates among bug reports [23].

In order to use our information for testing, we decided to base the prototype on an instance
6. Building and Evaluating the Prototype

Figure 6.1: The process of converting an execution profile to a map of terms and their frequencies.

We defined the algorithm so that it had two parameters which could be altered. The two parameters were the number of method calls that was used, starting from the most recent call as well as a threshold determining the similarity needed for tests to be clustered together. Where the number of method calls can essentially be set to any positive integer and if it is set to a value higher than the amount of method calls available for any given test case, all methods will simply be used.

We let each test case execution be represented by an execution profile. We created the profiles by extracting the names of the method used in the last client-engine requests, based on the number varied based on the number of method calls parameter value. These method names were then split according to camel casing, as such the hypothetical method *ExecuteQuery* would be split into the two terms *execute* and *query*. We then defined the execution profile as a set of terms as well as the raw frequency for how many times each term occurred in the test execution, as depicted in Figure 6.1.

We based the clustering algorithm on the vector space model. Each execution profile was represented as a vector in the model where the terms were weighted using augmented term frequency. The initial state of the algorithm was defined as an empty set of clusters. All test cases were then iterated through. Each test case would then be added to the cluster
6.1 Iteration #1 Client-server Traffic

with the highest average cosine similarity to the test case vector. But in the case where the
cluster with the highest average cosine similarity produced a similarity score lower than
the threshold, a new cluster, only containing the test case would be formed and added to
the set of clusters. The result of the algorithm would then be a clustering of the test cases.
The pseudo code can be seen in algorithm[7].

\textbf{Algorithm 1:} The clustering algorithm used in iteration one.

6.1.4 Evaluation

Creating a Test Set

In order to evaluate our progress, we had to create a test set by fault seeding which we
could use to test and evaluate our approach [8]. As we focused on clustering back-end
failures, we created our first test set by introducing bugs in the back-end, more specifically
the engine.

Introducing bugs was a greater task than we expected. We initially planned to use code
coverage in order to find relevant lines of code to sabotage, but as no such functionality
was available to us we instead decided to sabotage code so that the defects would affect
specific Engine-API requests sent from the client.

We chose to introduce defects that would affect API-request-methods which were not too
commonly used. The reason for this was that we wanted a smaller test set which would be
easier to analyze. In order to accomplish this we calculated which API-requests that were
called by which test cases. This was done by temporarily modifying out prototype such
Building and Evaluating the Prototype

Figure 6.2: The correct clustering for the test set created in iteration one.

that it mapped API-requests to the test cases which had used them. We then introduced one single fault which would affect a method called `DestroyBookmark`.

In addition to the the `DestroyBookmark` fault, the branch we were working on also contained an additional defect which manifested itself as a timeout error. We tried to introduce a few more faults but due to time limitations, the repeated occurrence of faults getting caught by unit tests or code standard checks during the build process or the fact that some defects were not caught by the system tests run by Horsie but only by tests run by the `ProtocolTester4Net` framework, we settled on using the Horsie run tests with the `DestroyBookmark` defect and the timeout defect as our first test set. We think the defect we introduced was fairly realistic since our defect is a change of a Boolean from true to false, an error we ourselves have done previously.

In our test set a total of 11 test cases had failed, 10 due to the `DestroyBookmark` defect and one due to the timeout defect. An optimal clustering of the test set can be seen in Figure 6.2.

Adjusted Rand Index Calculator

In order to evaluate the clustering which our approach would produce, we created a Java program which based on an optimal clustering could automatically calculate the adjusted Rand index between the optimal clustering and the clustering produced by our algorithm.

Clustering using the Prototype

In order to test our initial prototype we manually ran it using different values on the algorithm parameters after which we calculated the adjusted Rand index compared to the optimal clustering for each of the resulting clusterings. Using trial and error with regards to the parameter settings we found that the settings $\text{threshold} = 0.65$ and $\text{number of methods used} \in \{2, 3, 4\}$ produced the best clustering with regards to the adjusted Rand index. The settings
resulted in three clusters being created, essentially breaking up the DestroyBookmark-cluster in two, one containing two failures while the other contained eight, the timeout error failure was correctly clustered in a separate cluster. When comparing this clustering to the optimal clustering, the resulting adjusted Rand index was 0.397. We also compared our prototype to NIOCAT by running NIOCAT with the same threshold. NIOCAT produced six clusters resulting in an adjusted Rand index of 0.073, a comparison between the two can be seen in Figure 6.3. NIOCAT run with its default threshold of 0.8 produced an additional cluster resulting in a somewhat lower adjusted Rand index.

6.1.5 Reflection

The initial iteration made us cautiously positive to using the method names sent in the traffic between the client and the engine for clustering. We could see that for the one test set we had, our simple prototype was performing better than the current NIOCAT tool, however since we only had a very small test set it was hard to draw any real conclusions.

We understood that we would need a larger test set and do a more complete evaluation of how the different parameter values such as the number of methods used and the clustering threshold influenced the clustering. This would be part of what we focused on in the next iteration.

6.2 Iteration #2 TF-IDF and Suspicious Statements

In the second iteration we started with a literature study trying to find new sources of back-end information to use when clustering as well as new ways to cluster using the method names from the client-server traffic. We then improved our prototype using TF-IDF and
Suspicous statements instead of only TF when weighting the method names in our clustering process. We also crafted a larger test set and the improved prototype was evaluated against it. Using the test set, we also compared the new approached to of NIOCAT and to the prototype from iteration one. Finally we reflected on the results of the improved prototype.

6.2.1 Literature Study

After conducting the first iteration and noticing positive results with clustering using the method names sent in the traffic between the client and the server we decided to investigate how we could further utilize the method names. We therefore conducted a literature study focusing on this as well as on finding different information that could be used for clustering instead of or in addition to the traffic between the client and the server. In the end we did not find any new sources of information that seemed applicable to Qlik that we had not already found in the literature study during iteration one.

We also thought about the possibility to extend our clustering from only including the Hor-sie system tests to also include the tests from the ProtocolTester4Net framework. This was after realizing that the ProtocolTester4Net framework also saved the client-server traffic from its tests in a log for each test.

After conducting the literature study and after further discussions with employees at Qlik we had come up with a set of approaches which further utilized the method names:

- **Boolean TF** In iteration one, we chose to weight the terms using augmented term frequency as to prevent a bias towards tests which requested a large number of methods. We were however interested in using other types of term frequency, mainly Boolean frequency, where the weight is either 1 if the term is in the document or 0 if it is not. Our hypothesis was that as we were only using a fixed number of method calls, when only using a small number of them, the augmented term frequency might not have a large impact. We were thus interested as to how augmented and Boolean term frequency would affect the clustering. We did however not find any information about using Boolean frequency in combination with the vector space model in any reports we read.

- **TF-IDF** We found that in multiple reports TF-IDF is used in order to weight terms when clustering program executions [11][9]. We were therefore interested in testing this approach in order to increase the weight of rarer terms in favor of more common ones.

- **Suspicious Statements** In a couple of reports we found that suspicious statements had been used in order to locate defects which cause program executions to fail or to find similar test executions [38][19]. In one report, suspiciousness is used to highlight lines of code in a color based on how likely the algorithm estimates that that line is defective [19]. We did however not find any reports examples of suspiciousness being used when clustering program executions. We still thought that the approach of using suspiciousness in order to weight methods in the vector
6.2 Iteration #2 TF-IDF and Suspicious Statements

The space model could generate positive results concerning the clustering. As we had established in the previous iteration, we did not have access to which statements that were executed in the source code, we therefore intended to use the method names from the client-server traffic instead of statements. We would then weight methods in the vector space model using the suspiciousness values for each method.

6.2.2 Check Extractability

Accessing the client-server traffic from the executed ProtocolTester4Net tests required some guidance from testing experts at Qlik. After getting pointed in the right direction we found out that for each failed test case, there were logged information containing which requests and responses that had been received and sent by the engine. These logs were accessible though a database which our current prototype had access to, we would simply have to modify our prototype and create a parser for these logs.

In order for us to be able to weigh the terms using Boolean term frequency we would need no additional information, we would simply need to alter the prototype. As such, we did find Boolean TF as a relatively simple approach to evaluate. In contrast, we found that if we wanted to approach weighting using TF-IDF, we would need to extract additional information from the executed test cases. To use IDF we needed to be able to access all methods that had been requested, both for failed and successful test cases. This, due to the fact that IDF is calculated using all the documents in a corpus to calculate how common a term is. As such, as the tests in our case are the documents, we would need all tests and all methods executed by each test.

For the Horsie tests all methods that had been requested by the client was available in the logs, both for successful and failed test cases, but for ProtocolTester4Net only the methods that had been requested by the client in the failed test cases were logged. We therefore, once more consulted a testing expert at Qlik in order to find out how we could make sure that the client-server traffic was accessible for successful tests in the case of the two testing frameworks. We were told that it was possible with some small changes to the ProtocolTester4Net framework.

As with TF-IDF, suspiciousness required that we did have access to the all methods that had been called for each executed test case, both failed and successful tests. Essentially, if we would be able to extract this information in order to test the TF-IDF approach, we would need no additional information in order to create some form of suspiciousness weighting.

6.2.3 Developing the Prototype

We started by focusing on implementing functionality so that ProtocolTester4Net tests could be accessed by the prototype. As the ProtocolTester4Net tests do not use the proxy which we had previously modified to log traffic, the traffic had to be extracted in a different manner.
We modified our prototype to being able to parse and process log files containing in and outgoing client-server traffic, such that Horsie and engine only tests run by the ProtocolTester4Net framework could be treated equally when it comes to back-end data available to our prototype.

We then modified the ProtocolTester4Net framework such that the traffic generated by the successful tests were also logged in the same manner as how failed tests were. The prototype now had access to a list of which methods that were requested by both successful and failed tests executed by both frameworks.

The next task we attempted was to modify our prototype so that IDF could be used. We therefore let the prototype request all tests that were executed, both failed and successful and made sure that each test case were linked to the methods which it had requested. We now calculated the inverse document frequency for all methods according to the IDF formula. We did this by for each method calculate the logarithm of the fraction between the total number of available documents and the number of documents in which the method appeared:

\[
IDF(m) = \log \frac{\text{total number of tests}}{\text{number of tests which uses method } m}
\]  

We proceeded by implementing functionality such that TF-IDF could be used with both augmented term frequency but also with Boolean term frequency.

After implementing inverse document frequency we continued with implementing the suspicious method functionality. We did this in a similar manner as to how we implemented IDF, but using a formula such that each method was assigned a suspiciousness value calculated as the fraction between the number of failed test cases which used the method and the total number of tests which used the method:

\[
\text{Suspiciousness}(m) = \frac{\text{number of failed tests which used method } m}{\text{number of tests which uses method } m}
\]

We now had three different approaches to test more in depth: Term Frequency with Camel Case splitting, as tested in iteration one, TF-IDF with whole method names and suspicious methods also using whole method names.

The algorithm used in the prototype now had multiple parameters for each approach, which are listed, among with their potential values in Table 6.1 Table 6.2 and Table 6.3.

We decided to set the maximum number of methods we would use to 40, since most test cases made no more than 20 method requests, we decided to let 40 represent maximum number of methods we would use as we thought that it was a reasonable max limit.

<table>
<thead>
<tr>
<th>TF Camel Case Splitting Parameter</th>
<th>Possible Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of method names</td>
<td>1 – 40</td>
</tr>
<tr>
<td>Threshold</td>
<td>0.0 – 1.0</td>
</tr>
</tbody>
</table>

**Table 6.1:** The parameters which could be set for the approach used in iteration one, TF with Camel Case splitting.
6.2 Iteration #2 TF-IDF and Suspicious Statements

<table>
<thead>
<tr>
<th>TF-IDF Parameter</th>
<th>Possible Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of method names</td>
<td>1 – 40</td>
</tr>
<tr>
<td>Threshold</td>
<td>0.0 – 1.0</td>
</tr>
<tr>
<td>TF type</td>
<td>Augmented / Boolean</td>
</tr>
</tbody>
</table>

Table 6.2: The parameters which could be set for the TF-IDF approach.

<table>
<thead>
<tr>
<th>Suspicious Methods Parameter</th>
<th>Possible Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of method names</td>
<td>1 – 40</td>
</tr>
<tr>
<td>Threshold</td>
<td>0.0 – 1.0</td>
</tr>
</tbody>
</table>

Table 6.3: The parameters which could be set for the Suspicious Methods approach.

We also made sure that the prototype could be set up to run multiple different configurations consecutively. By essentially providing the prototype with a list of values for each parameter, we let the prototype create a clustering for each set of permutations. All of these configurations were easily modifiable through the use of a configuration file. This meant that we could easily set up and test the algorithm using thousands of different permutations. Figure 6.4 shows an example of how the parameter permutations work.
6.2.4 Evaluation

Creating Test Sets

In order to create more test sets with which we could evaluate our prototype we spent a considerable amount of time fault seeding and finding ways in which we could sabotage the back-end engine component of Qlik Sense such that we would have multiple defects in each test set. Our hopes were that we, in addition to artificial test sets, would be able to find an actual commit with a number of defects which had been created during actual development of Qlik Sense which we could use as a test set. We did however not find such a commit. There were multiple reasons for this, the main reason was that lately, few known back end defects had been detected on the branches we had access to.
After a considerable amount of trial and error attempts at sabotaging the Qlik Sense engine, we had created three test sets, built up from six different defects which we had introduced in the Qlik Sense back-end engine component. Due to the limited number of introduced defects, we built the test sets by combining the six defects. All the defects we introduced were fairly realistic in the way the methods were sabotaged. However, all defects were introduced in methods directly requested to run by the client which does not exactly mirror all real defects since real defects can also be in parts of the back-end that the client does not directly request.

• The first test set, test set A, consisted of 74 failed tests which originated from four different defects, as see in Figure 6.5. The four defects originated from sabotage which targeted the following methods: DestroyBookmark, GetScriptBreakpoint, CheckScriptSyntax and GetHyperCubeReducedData.

• The second test set, test set B, consisted of 153 failed tests which originated from five different defects, consisting of the tests in test set A, plus an additional defect, as see in Figure 6.6. The additional defect originated from sabotage targeting the method Lock.

• The third test set, test set C, consisted of 202 failed tests which originated from five defects, consisting of the test in test set A, plus an additional defect, as see in Figure 6.7. The additional defect originated from sabotage targeting the method CreateDimension.

![Test Set A - Correct Clustering](image)

**Figure 6.5:** A graphic representation of the correct clustering of test set A. The 74 failures are clustered into four clusters. Cluster 1 contains 55 failures, cluster 2 contains 15 failures and cluster 3 and 4 contain 2 failures each.
6. Building and Evaluating the Prototype

![Test Set B - Correct Clustering](image)

**Figure 6.6:** A graphic representation of the correct clustering of test set B. The 153 failures are clustered into five clusters. Cluster 1 contains 55 failures, cluster 2 contains 16 failures, cluster 3 and 4 contain 2 failures each and cluster 5 contains 78 failures.

![Test Set C - Correct Clustering](image)

**Figure 6.7:** A graphic representation of the correct clustering of test set C. The 202 failures are clustered into five clusters. Cluster 1 contains 55 failures, cluster 2 contains 16 failures, cluster 3 and 4 contain 2 failures each and cluster 5 contains 127 failures.

In test set B and C we had 16 *DestroyBookmark* failures while only 15 in test set A. This is due to us merging in code from the Qlik Sense master branch to our branch after creating...
test set A. In the newly added code an additional test was added that happened to fail due to our *DestroyBookmark* code defect.

**Human Factors for Evaluating Clusterings**

To be able to evaluate the clusterings with the the prototype, we talked to our supervisor who is a principal software engineer at Qlik. He informed us that getting as close as possible to the correct amount of clusters is one of the most important things for the developers and testers since every additional cluster increases the analyze time.

In addition, an approach allowing for a higher threshold was more or less always preferred to a lower one. Since a clustering approach using a higher clustering threshold is more probable to create clusters that only contains test case failures originating from the same problem, i.e. a "pure" cluster which makes it easier to analyze.

In addition to adjusted Rand index, we used the information from our supervisor as metrics in order to evaluate the clusterings produced by the prototype. As such the metrics we used to evaluate the approaches were the following:

- The similarity between the correct clustering and the one produced by the evaluated approach calculated using the adjusted Rand index.

- How close to the correct number of clusters the clustering produced by the approach were (closer is better).

- The purity of the clusters (each cluster should optimally only contain failures from one source).

**Clustering using the Prototype**

We chose to distinguish between the different approaches in order to more easily evaluate them. As such we ran the prototype on the three different configurations. The configurations were:

- **Iteration One (Augmented TF with Camel Case Splitting)** - Weighting each term based on how many times it occurs in the test execution. The terms were generated by camel case splitting.

- **TF-IDF** - Weighting each method based on how many times is is requested in the test execution as well as how unusual the method is.

- **Suspicious Methods** - Weighting each method based on how often it is requested by tests that have failed.

For each of the three approaches, we let the prototype cluster the three test sets using a large number of different configurations. We then compared each clustering to the correct clustering for that test set by calculating the adjusted Rand index between the two. In addition to the configurations above, we also let Qliks current clustering tool NIOCAT cluster the test sets. This was so we could compare the results to how NIOCAT performed.
After letting the prototype run the different approaches. We found some patterns regarding the different parameters. We initially focused on analyzing the clusterings which had been achieved with a threshold of 0.8 since this is the threshold currently used by NIOCAT at Qlik.

The parameters which produces the best clusterings for a threshold set to 0.8 for each approach can be seen in Table 6.4, Table 6.5 and Table 6.6. In addition, NIOCAT, with its original settings can be seen in Table 6.7.

<table>
<thead>
<tr>
<th>Iteration one - Optimal Parameters (threshold:0.8)</th>
</tr>
</thead>
<tbody>
<tr>
<td># methods</td>
</tr>
<tr>
<td><strong>Test Set A</strong></td>
</tr>
<tr>
<td><strong>Test Set B</strong></td>
</tr>
<tr>
<td><strong>Test Set C</strong></td>
</tr>
</tbody>
</table>

Table 6.4: The parameter which produces the best clusterings for the Augmented TF approach for a threshold set to 0.8. A hyphen (-) between two values indicates that the same result was reached for any value within the span. The value in parenthesis besides the number of clusters represents the correct number of clusters for that test set.

<table>
<thead>
<tr>
<th>TF-IDF - Optimal Parameters (threshold:0.8)</th>
</tr>
</thead>
<tbody>
<tr>
<td># methods</td>
</tr>
<tr>
<td><strong>Test Set A</strong></td>
</tr>
<tr>
<td><strong>Test Set B</strong></td>
</tr>
<tr>
<td><strong>Test Set C</strong></td>
</tr>
</tbody>
</table>

Table 6.5: The parameter which produces the best clusterings for the TF-IDF approach for a threshold set to 0.8. A hyphen (-) between two values indicates that the same result was reached for any value within the span. The value in parenthesis besides the number of clusters represents the correct number of clusters for that test set.
6.2 Iteration #2 TF-IDF and Suspicious Statements

<table>
<thead>
<tr>
<th>Suspicious Methods - Optimal Parameters (threshold:0.8)</th>
</tr>
</thead>
<tbody>
<tr>
<td># methods</td>
</tr>
<tr>
<td>Test Set A</td>
</tr>
<tr>
<td>Test Set B</td>
</tr>
<tr>
<td>Test Set C</td>
</tr>
</tbody>
</table>

Table 6.6: The parameter which produces the best clusterings for the Suspicious Methods approach for a threshold set to 0.8. A hyphen (-) between two values indicates that the same result was reached for any value within the span. The value in parenthesis besides the number of clusters represents the correct number of clusters for that test set.

<table>
<thead>
<tr>
<th>NIOCAT - (threshold:0.8)</th>
</tr>
</thead>
<tbody>
<tr>
<td>adjusted Rand index</td>
</tr>
<tr>
<td>Test Set A</td>
</tr>
<tr>
<td>Test Set B</td>
</tr>
<tr>
<td>Test Set C</td>
</tr>
</tbody>
</table>

Table 6.7: NIOCAT when clustering the three test sets using its original threshold setting of 0.8. The value in parenthesis besides the number of clusters represents the correct number of clusters for that test set.

- **Iteration one - threshold:0.8** - When looking at the clusterings generated by the augmented TF approach with a threshold of 0.8, the highest adjusted Rand index scores for test set A and B was achieved when using 20 to 40 method names. For test set C, the best results were generated even when using a lower amount of methods. Still, the maximum number of methods seemed to work well for all three test sets. Overall on this threshold, the augmented TF approach produces the worst adjusted Rand index of the three, with an average rand index of 0.540 on the three test sets. The number of clusters produced was on average about 3.5 times as many as the correct number of clusters.

- **TF-IDF - threshold:0.8** - On test set A and B, the TF-IDF approach for the threshold 0.8 achieved its best adjusted Rand index score when using a very low amount of method names, only 2. In addition, the augmented term frequency seemed to be the best option for those test sets. For test set C, neither the number of methods used or the term frequency type seem to produce different results. Overall on this threshold, the TF-IDF approach produces the second best adjusted Rand index of the three, with an average rand index of 0.644 on the three test sets. The number of clusters produced was on average about 2.3 times as many as the correct number of clusters.

- **Suspicious Methods - threshold:0.8** - The Suspicious Methods approach produced overall the best results regarding adjusted Rand index of the three. The approach produced the best clusterings for all three test sets when using 20 to 40 methods. For the three test sets, the approach produced an average adjusted Rand index of
6. Building and Evaluating the Prototype

0.910. The number of clusters produced was on average about 1.4 times as many as the correct number of clusters.

- **NIOCAT - threshold: 0.8** - The original NIOCAT approach on average produced clusterings with an adjusted Rand index of 0.296 and with a number of clusters equal to 5.9 times that produced by the correct clustering.

For each of the three approaches, we also let the threshold vary in order to see when the best clusterings were produced when any threshold was allowed. The best results for each approach and test set can be seen in Table 6.8, Table 6.9 and Table 6.10. In addition, plots displaying the average adjusted Rand index for the three test sets for each approach with different parameter settings can be seen in Figure 6.8, Figure 6.9 and Figure 6.10.

<table>
<thead>
<tr>
<th>Iteration one - Optimal Parameters per Test Set (any threshold)</th>
<th># methods</th>
<th>threshold</th>
<th>adjusted Rand index</th>
<th># clusters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Set A</td>
<td>20-40</td>
<td>0.6</td>
<td>0.926</td>
<td>6(4)</td>
</tr>
<tr>
<td>Test Set B</td>
<td>15-25, 40</td>
<td>0.5</td>
<td>0.755</td>
<td>5(5)</td>
</tr>
<tr>
<td>Test Set C</td>
<td>2</td>
<td>0.3-0.5</td>
<td>0.804</td>
<td>7(5)</td>
</tr>
</tbody>
</table>

**Table 6.8:** The parameter which produces the best clusterings for the Augmented TF approach without any threshold restrictions. A hyphen (-) between two values indicates that the same result was reached for any value within the span. The value in parenthesis besides the number of clusters represents the correct number of clusters for that test set.

<table>
<thead>
<tr>
<th>TF-IDF - Optimal Parameters per Test Set (any threshold)</th>
<th># methods</th>
<th>TF-type</th>
<th>threshold</th>
<th>adjusted Rand index</th>
<th># clusters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Set A</td>
<td>20-35</td>
<td>Boolean, augmented</td>
<td>0.1-0.2</td>
<td>1.0</td>
<td>4(4)</td>
</tr>
<tr>
<td>Test Set B</td>
<td>20-40</td>
<td>Boolean, augmented</td>
<td>0.1-0.4</td>
<td>0.875</td>
<td>6(5)</td>
</tr>
<tr>
<td>Test Set C</td>
<td>20-40</td>
<td>Boolean, augmented</td>
<td>0.1-0.4</td>
<td>0.883</td>
<td>6(5)</td>
</tr>
</tbody>
</table>

**Table 6.9:** The parameter which produces the best clusterings for the TF-IDF approach without any threshold restrictions. A hyphen (-) between two values indicates that the same result was reached for any value within the span. The value in parenthesis besides the number of clusters represents the correct number of clusters for that test set.
6.2 Iteration #2 TF-IDF and Suspicious Statements

### Suspicious Methods - Optimal Parameters per Test Set (any threshold)

<table>
<thead>
<tr>
<th>Test Set</th>
<th># methods</th>
<th>threshold</th>
<th>adjusted Rand index</th>
<th># clusters</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>20-40</td>
<td>0.1-0.8</td>
<td>0.98-1.0</td>
<td>4-5(4)</td>
</tr>
<tr>
<td>B</td>
<td>20-40</td>
<td>0.1-0.8</td>
<td>0.869-0.874</td>
<td>6-7(5)</td>
</tr>
<tr>
<td>C</td>
<td>20-40</td>
<td>0.1-0.8</td>
<td>0.880-0.883</td>
<td>5-7(5)</td>
</tr>
</tbody>
</table>

**Table 6.10:** The parameter which produces the best clusterings for the Suspicious Methods approach without any threshold restrictions. A hyphen (-) between two values indicates that the same result was reached for any value within the span. The value in parenthesis besides the number of clusters represents the correct number of clusters for that test set.

### NIOCAT - (best threshold)

<table>
<thead>
<tr>
<th>Test Set</th>
<th>threshold</th>
<th>adjusted Rand index</th>
<th># clusters</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.4</td>
<td>0.582</td>
<td>6(4)</td>
</tr>
<tr>
<td>B</td>
<td>0.4</td>
<td>0.721</td>
<td>11(5)</td>
</tr>
<tr>
<td>C</td>
<td>0.5</td>
<td>0.891</td>
<td>12(5)</td>
</tr>
</tbody>
</table>

**Table 6.11:** NIOCAT when clustering the three test sets without any threshold restrictions. The value in parenthesis besides the number of clusters represents the correct number of clusters for that test set.

### Adjusted Rand Index - Iteration One

**Figure 6.8:** The average adjusted Rand index for the three test sets for the approach developed in iteration one, augmented TF with camel case splitting.
6. Building and Evaluating the Prototype

Figure 6.9: The average adjusted Rand index for the three test sets for the TF-IDF approach, where either augmented or Boolean TF is used.

<table>
<thead>
<tr>
<th># methods</th>
<th>TF-type</th>
<th>threshold</th>
<th>adjusted Rand index</th>
<th>clusters ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>20-40</td>
<td>augmented</td>
<td>0.2</td>
<td>0.912</td>
<td>1.2</td>
</tr>
</tbody>
</table>

TF-IDF - Optimal Parameters average Test Set (any threshold)
6.2 Iteration #2 TF-IDF and Suspicious Statements

**Figure 6.10:** The average adjusted Rand index for the three test sets for the suspicious method approach.

<table>
<thead>
<tr>
<th># methods</th>
<th>threshold</th>
<th>adjusted Rand index</th>
<th>clusters ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>20-40</td>
<td>0.8</td>
<td>0.910</td>
<td>1.3</td>
</tr>
</tbody>
</table>

**Figure 6.11:** The average adjusted Rand index for the three test sets for the Qlik tool NIOCAT.

<table>
<thead>
<tr>
<th>threshold</th>
<th>adjusted Rand index</th>
<th>clusters ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.4</td>
<td>0.702</td>
<td>1.8</td>
</tr>
</tbody>
</table>

- **Iteration one - flexible threshold** - When looking at the clusterings generated by the augmented TF approach without any threshold limitations, the highest adjusted Rand index scores for test set A was very high, 0.926. This was achieved when using anywhere from 20 to the maximum number of method names and a threshold of 0.6.
For the other second test set, a relatively high adjusted Rand index was also achieved, 0.755. This was achieved by using a threshold of 0.6 and anywhere between 15 and 40 method names, excluding the span of 26-39. For test set C, the best results were achieved by only using two method names and a threshold anywhere between 0.3 and 0.5. The best parameter settings generated the average adjusted Rand index of 0.78 and the number of clusters produced was on average 1.5 times the correct amount.

- **TF-IDF - flexible threshold** - Evaluating the TF-IDF approach, we found that the best clusterings for test set B and C, were achieved when the number of methods used were in between 20 and 40 and with a threshold between 0.1 and 0.4. Similar parameter values were optimal for test set A, but here the number of methods used were between 20 and 35 and the threshold were set between 0.1 and 0.2. The best parameter for the average test set resulted in an average adjusted Rand index of 0.91 and the number of clusters produced was on average 1.2 times the correct amount.

- **Suspicious Methods - flexible threshold** - When using the suspicious methods approach, a threshold between 0.1 and 0.8 always seemed to produce the best clustering for all three test sets, also all three test sets, the number of methods could be set anywhere between 20 and 40. On average, the resulting adjusted Rand index was 0.91 and the number of clusters produced was on average 1.3 times the correct amount.

- **NIOCAT - flexible threshold** - When we searched for the optimal threshold for the NIOCAT approach, we found that the best parameters were a threshold of 0.4 for test set A and B and 0.5 for test set C. For the threshold 0.4 on average between the three test sets, NIOCAT produced clusterings with an adjusted Rand index of 0.70 and with a number of clusters equal to 1.8 times that produced by the correct clustering.

### 6.2.5 Reflection

**Iteration one: augmented term frequency and camel case splitting**

The approach from iteration one on average seemed to be the least reliable of the three approaches. This due to a number of reasons such that it for high thresholds produced poorly in terms of adjusted Rand index as well as the number of clusters produced in comparison to the other two approaches. And when looking at the plot in Figure 6.8, the approach seem to drastically worsen in performance when the parameters are slightly changed from the values which produce the best results. The best value of the parameter determining the number of methods used seem to be less consistent between the best clustering for the three test sets. As seen in Table 6.8, the best clustering for test set C is produced with only two methods used, and a slightly lower threshold, while the approach performs better using more methods on the two other test sets. Provided a larger scaled evaluation, using larger and more diverse test sets, we hypothesize that due to the slim margin for the optimal parameters, the parameters for numbers of methods used and the clustering threshold
might not fit all test sets. The upside to this approach is that the calculations are lighter than those required by the TF-IDF and suspicious methods approaches as this approach only requires information from the actual failed test cases.

When investigating the reason for why this approach performs so well using only two methods for test set C we found that is could be attributed to the fact that many of the test case failures in test set C has only requested two methods before a test failure was detected. Thus for a large quantity of test cases in the set, setting the number of methods used to two results in the inclusion of all methods requested.

The TF approach did outperform the original configuration of NIOCATE (when a threshold of 0.8 was used) which is what has been determined to work best overall for NIOCATE on Qliks test cases. Still when NIOCATE was used with a threshold of 0.4, NOICAT was able to produce comparable results as our iteration one approach. In general, the iteration one approach seemed to perform best with low thresholds close to 0.6, when looking at the graph in Figure 6.8 the best adjusted Rand index values were achieved with a larger amount of methods and when the threshold is between 0.5 and 0.6.

TF-IDF

TF-IDF seems to outperform the approach from iteration one concerning adjusted Rand index and the number of clusters produced for higher thresholds. Patterns in the plots in Figure 6.9 seem to show that when using lower thresholds, the approach allows for a less narrow span regarding parameter values in order to perform the best clustering compared to the iteration one approach.

In general, the approach seem to perform best using low thresholds and the clustering performance is reduced by a lot when raising the threshold. A problem with running on lower thresholds is that the clustering becomes more susceptible to adding a test case to a cluster where does not belong. This is because the test case does not have to have as high similarity score to be included in a cluster that it would need to have when running on a higher threshold. Running on a lower threshold does therefore increase the risk of creating impure clusters, that is clusters containing test cases that have failed due to different code defects. This is something our supervisor at Qlik said was highly undesired. This did not happen in our evaluation, but that might be because the failures in our test sets were very different from each other, and thus two test cases failing due to two different code defects would get a very low similarity score and not be clustered together even on a low threshold.

For the TF-IDF approach, when reviewing the graphs in Figure 6.9 it seemed that the type of term frequency used did not effect the result to a very large extent, this is quite interesting and seems to suggest that the number of times a method is used does not matter very much for the TF-IDF clustering approach. Still, this might not be completely accurate as the augmented TF seem to outperform the Boolean TF with a slight margin when the number of methods is increased.

The TF-IDF approach overall seem to perform best when the threshold is below 0.4 and the number of methods is large. Using 20-40 methods can somewhat simplified be seen
as the same as using all methods available. As most test cases do not perform more than 20 method requests before failing.

Even though the TF-IDF approach seem to outperform the TF approach from iteration one, a downside to TF-IDF is that a lot more information is requited in order to calculate the IDF value of each method. As data is not only extracted from the failed test cases but also the successful ones.

**Suspicious Methods**

The suspicious methods approach seemed to perform best of the three approaches. For high thresholds it outperformed the two other approaches in terms of both adjusted Rand index and the number of clusters produced and the purity of the clusters. In addition, it seemed to produce clusterings of similar quality for a wide spectrum of threshold values between 0.1 and 0.8. With even more large scale testing, the optimal threshold spectrum might be able to be narrowed down somewhat further but we believe this approach to be very promising due to the fact that the algorithm, for our test sets, is stable for such a variety of configurations.

This approach did perform better when a larger number of methods was included. In addition it achieved very high adjusted Rand index values almost independent of the threshold. As long as the number of methods used was larger than 15, a threshold between 0.1 and 0.8 seemed to produce clusterings which on average achieved an adjusted Rand index value of over 0.8, this can be seen in Figure 6.10. As mentioned in the TF-IDF section above, using between 20 and 40 method can be regarded as somewhat the same as using all methods available.

The fact that the suspicious approach was the only approach that managed to get a high adjusted Rand index for all test cases at a high threshold is the main reason that we think that this approach is the most promising one. Having a high adjusted Rand index at a high threshold means that even if two defects produced quite similar failures they would most probably not be clustered together. Something that is much more probable to happen with the other approaches since they needed to be run at a lower threshold to get as good adjusted Rand index scores as the suspicious approach.

A downside to the suspicious methods approach is that as with the TF-IDF approach, information from each successful test case is required, which in the approach from iteration one, as well as the NIOCAT tool does not require.

**General Reflections**

NIOCAT was able to produce a high adjusted Rand index, about 1% higher than the suspicious methods approach, when using a low threshold for test set C. This result is not very significant since it is only 1% better and also seems to be coincidental as it is not reflected for the two other test sets. The result can probably be attributed to the fact that a large number of the test cases which fail due to the same defect in test set C have similar
6.2 Iteration #2 TF-IDF and Suspicious Statements

Figure 6.12: The clustering of test set C produced by NIOCAT using a threshold of 0.5. The clustering received a high adjusted Rand index but a large number of clusters were produced and multiple clusters were impure and thus contained failures originating from more than one defect.

Another big benefit of our prototype independent on what of the approaches that are used compared to the current NIOCAT tool is the ability to cluster test cases that have been run by different test frameworks. The original NIOCAT tool can not create a cluster with test cases from two different frameworks even if they all have failed due to the same code defect.
6. Building and Evaluating the Prototype

Figure 6.13: The clustering of test set C produced by the suspicious methods approach using a threshold of 0.8. The clustering received a high adjusted Rand index and produced seven clusters which was close to the correct number (five). All seven clusters were pure in the sense that no cluster contained failures which had occurred due to different defects.
Chapter 7

Conclusions

In this chapter we give our answers to the three research questions we had going in to this thesis. We also point out threats to the validity of our results as well as possible future work.

7.1 Which back-end information can be used to improve precision when clustering test case executions?

During the literature study phases of each iteration we found multiple types of back-end information which could be used to cluster test case executions.

- **Call Stacks** - In the case where a test case has caused a software crash, we found that analyzing the state of the call stack for when the software crashes is useful for clustering together test cases. This can be seen in several of the papers we discuss in the related work chapter [23] [10].

- **Code Coverage** - We found that code coverage, meaning essentially the statements which a test case resulted in the execution of, is a useful type of information to utilize when clustering together test cases [11] [40].

- **Exceptions** - If a test case execution triggered an exception in the software, using the exception message and the exception type, test case failures may be clustered together. This approach was by itself not very successful according to the literature and might be best used as a complement to another type of information [23].
7. Conclusions

- **Client-server Traffic** - From our supervisor at Qlik, we were introduced to the concept of using client-server traffic to cluster failed test case executions. This is the type of information we have used in this thesis which has shown promising results. The closest thing to using client-server traffic that we found in literature was the use of database operations as clustering metrics [34].

7.2 Which types of back-end information can be made accessible after a test execution?

We found that in the case of Qlik Sense, we were able to access client-server traffic. We did this through logging the requests and responses made by and to the client and later on parsing the log files so that the information could be used for clustering. Exceptions in the case of Qlik Sense could potentially also be made available after a test case execution. In addition, for test cases testing Qlik Sense, it would be a more difficult task to extract Call Stack and Code Coverage information without spending large resources to implement these features. As such, we did find that the easiest types of information which could be made available after a test execution was client-server traffic.

7.3 How can this back-end information be used to improve clustering precision?

We used client-server traffic in order to create clusterings with high precision. More precisely the requests which were made from the client to the server. We used the method names in these requests and defined three different approaches for clustering using these.

All three approaches were based on using the vector space model in order to represent each test case as a vector created by using the methods requested during the execution of that test case. Clusters were then created by comparing the cosine similarity measure between pairs of test case vectors. The vectors were created differently based on the three approaches. The approach which we found to be best, was to calculate the suspiciousness of each method, essentially as a fraction of the number of failed test cases in which it was requested divided by the total number of test cases in which it was requested. This approach was for high thresholds able to create the best clusterings among the approaches evaluated and was tied with the TF-IDF approach when any threshold was allowed. When clustering using the suspiciousness approach the method names for each test case were used as terms which were assigned a weight based on how suspicious the method was. We did investigate how the number of method requests used in the calculations, starting from the most recent, effected the clustering. We found that the best clusterings for the suspiciousness approach were made when all method calls made by each test case were used in the calculations.
7.4 Threats to Validity

On average the approach from iteration one, TF and camel case splitting, with a low threshold of 0.6, created 1.5 times the correct number of clusters.

On average the TF-IDF approach, with a low threshold of 0.2 and using augmented term TF, created 1.2 times the correct number of clusters.

The suspicious methods approach did not differ very much in performance when the threshold was varied between 0.1 and 0.8 and produced, for thresholds in that range, between 1.2 and 1.4 times the correct number of clusters.

7.4 Threats to Validity

All results which we have found may be inapplicable in other contexts than for Qlik Sense and the Horsie and ProtocolTester4Net testing frameworks. Client-server traffic may not be available in the same sense and the way API:s are used may vary for other software products. As such, we can not state if the findings in this thesis can be generalized or if they are subjectively tailored to the clustering of Qlik Sense test case execution failures.

Due to reasons stated in the previous chapter, we did not manage to produce or access the diverse types of test information that we would have liked. The three test sets which we created did build upon each other, test set B and C are essentially test set A but with an additional unique code defect. Thus we did not have three test sets which were independent of each other and this might have effected our results or made the optimal parameter settings more tailored towards clustering focused on the four defects available in test set A. In addition, the test sets were created by introducing defects which we knew would target specific requested methods. Thus, a valid threat to the validity of the evaluation is that the test sets does not reflect the way that actual bugs occurs in the Qlik Sense code base.

Another potential threat to the validity of this thesis is that we manually for each test set annotated the failed test cases with the likely defect which caused it to fail. We did this very carefully and lots of effort was spent identifying which failures were originating from which defect. Still the possibility exists that we made a human error and annotated some parts of the test sets wrong. This would, if such was the case, effect the results of the evaluation.

Adjusted Rand index might not be the best metric to use when evaluating the clusterings which should later be analyzed by developers and testers at Qlik. When evaluating our approaches, we initially mainly used the adjusted Rand index value in order to determine how good a clustering was. After inspecting clusterings, such as that produced by NIOCAT on test set C where NIOCAT achieved an adjusted Rand index higher than any of our three approaches did, we found that the clustering by NIOCAT was actually not that useful compared to those produced by for example the suspicious methods approach. The reason was that too many clusters were produced and also the clusters were not pure as many of them contained failures from more than one defect.
7. Conclusions

7.5 Future Work

When analyzing the Qlik Sense Engine API, we found that the API included a method call named *GetInternalTest* which actually triggered internal tests in the engine, resulting in nothing further being logged. Two test cases using this method might in turn test completely different things even though the logs we use to extract method requests does not reflect it. Still, these methods might be deemed very suspicious by our suspicious methods approach and result in multiple test cases using this method being clustered together even though they actually fail due to completely different code defects. In order to solve this, one might be able to extract what method requests are executed by the internal test execution or these tests may simple be excluded from the clustering.

Another thing which might improve the clustering is to filter out method calls which are deemed immune to bugs in some sense. If Qlik has methods which they are very confident in are not defective, it may be worth ignoring these methods when clustering. This type of filtering has been used in an approach presented in a report named *ReBucket: A Method for Clustering Duplicate Crash Reports Based on Call Stack Similarity* [10].

As future work, we think that the suspicious methods approach should be further evaluated. We would like to see the approach being evaluated on actual non fabricated defects. This might however be a larger task, but would nonetheless be valuable when it comes to evaluating the approach. We would also like to test the TF-IDF approach in the same way.

In addition to having more test sets, we also think that it might be worth investigating if there are other evaluation measures than adjusted Rand index which better reflect how a developer or tester would evaluate a test failure clustering. Potential measures to evaluate could be precision, recall or combining them into something called F1-score [28].

Currently, the suspicious methods approach does only cluster test execution failures from one certain test execution. As additional future works, the suspicious methods approach should be able to cluster failed test cases from multiple test executions. This could potentially be done by creating a suspiciousness map for each test execution, and then weighting the test cases by using the suspiciousness rating generated from their individual execution. This approach would allow for test executions from different branches to be clustered together.

The clusterings in which the approaches result are currently only available in databases. Having a presentable way in which these clusterings can be viewed would be a natural next step in the development of this prototype. One way to do it would be to use Qlik Sense in a similar manner as to how the clusterings which NIOCAT produces are presented. Also, using the suspicious methods approach, and due to the fact that the suspiciousness of methods are already calculated. It would be a logical next step to alongside the clusters produced, present which methods that are the most suspicious, such that this information might also be used for debugging purposes.

During this thesis, we did not manage to extract code coverage information from executed test cases. We would however still like to find out how code coverage could be used in
order to cluster test cases originating from back-end defects. We believe this to be an approach which could be very successful for clustering and a path worth exploring in the future, when such information is available.

As a potential expansion of the prototype, it may be so that the order in which the methods are called is what triggers a failure. In future works, this approach could be explored by for example clustering using Markov chains to describe the execution trace of test cases.

In the suspicious methods approach, the suspiciousness values for each method might be better calculated using the logarithm of the fraction. In contrast to the IDF weighting part, used in the TF-IDF weighting approach, which is calculated as the logarithm of the fraction between the number of times the method is used in a test and the number of available tests, the suspiciousness value for the methods is not calculated using a logarithmic weighting scheme. But the suspiciousness of a method might not be proportional to the number of times it occurs in failed test cases. A logarithmic approach might have made for even better clusterings.
7. Conclusions
Bibliography


Suspekta Metodanrop Leder Till Smidigare Testning av Klient-Server System

**POPULÄRVETENSKAPLIG SAMMANFATTNING** Fredrik Folkesson, Johan Nyholm

Programvarutestning resulterar lätt i svårhanterligt stora mängder misslyckade tester, trots få faktiska fel i koden. Genom att identifiera misstänkta metodanrop mellan klient och server kan tester grupperas så att utvecklare kan fokusera på buggrättning istället för att gå vilse bland tester.


Ett typiskt exempel på hur kommunikationen mellan en klient och en server ser ut kan vara att klienten begär att en viss operation ska utföras i form av ett metodanrop, varpå servern svarar med resultatet av anropet. Under ett typiskt systemtest, det vill säga ett test som testar systemet i sin helhet, görs ofta ett flertal metodanrop.

Metodanrop som ofta gjorts av de tester som misslyckats kan klassas som suspekta. Genom att gruppera de tester som har misslyckats baserat på de gemensamma metodanrop som gjorts, med mer vikt på de suspekta metodanropen, kan testerna delas upp i grupper som speglar de faktiska felorsakarna. Varje grupp av testfall består då av tester som har anropat samma suspekta metodanrop och varje suspekt metodanrop beror i sin tur troligen på en specifik bugg i programmet, och på så sätt har testerna grupperats så att alla testerna i en grupp beror på samma faktiska bugg. Detta gör att mindre tid kan läggas på att hitta vilken bugg som har gjort att ett test har misslyckats och mer tid på att fixa buggarna.

Vi har på mjukvaruföretaget Qlik utvecklat en prototyp som gör just detta, i syfte att underlätta vid testning av Qlik produkten Qlik Sense. Vi demonstrerade lyckade resultat och reducerade till exempel 202 misslyckade tester, orsakade av fem olika serverfel, till sju grupper där varje grupp Endast innehöll tester som misslyckats på grund av samma bugg. Denna gruppering gör det väldigt mycket enklare att överskåda antalet buggar och att sedan rätta dem.