Master's thesis
Master's Programme in Data Science

Tool for grouping test log failures using string similarity algorithms

Vladimir Topias Kramar

September 25, 2022

Supervisor(s): Professor Jukka K. Nurminen

Examiner(s): Dr. Antti-Pekka Tuovinen

test logs, log parsing, string similarity algorithms

University of Helsinki
Faculty of Science

P. O. Box 68 (Pietari Kalmin katu 5)

00014 University of Helsinki

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Degree programme
Faculty of Science	Master's Programme in Data Science

Tekijä — Författare — Author

Vladimir Topias Kramar

Työn nimi — Arbetets titel — Title

Tool for grouping test log failures using string similarity algorithms

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidantal — Number of pages
Master's thesis	September 25, 2022	63

Tiivistelmä — Referat — Abstract

This work presents a novel concept of categorising failures within test logs using string similarity algorithms. The concept was implemented in the form of a tool that went through three major iterations to its final version. These iterations are the following: 1) utilising two state-of-the-art log parsing algorithms, 2) manual log parsing of the Pytest testing framework, and 3) parsing of .xml files produced by the Pytest testing framework. The unstructured test logs were automatically converted into a structured format using the three approaches. Then, structured data was compared using five different string similarity algorithms, Sequence Matcher, Jaccard index, Jaro-Winkler distance, cosine similarity and Levenshtein ratio, to form the clusters. The results from each approach were implemented and validated across three different data sets. The concept was validated by implementing an open-sourced Test Failure Analysis (TFA) tool. The validation phase revealed the best implementation approach (approach 3) and the best string similarity algorithm for this task (cosine similarity). Lastly, the tool was deployed into an open-source project's CI pipeline. Results of this integration, application and usage are reported. The achieved tool significantly reduces software engineers' manual work and error-prone work by utilising cosine similarity as a similarity score to form clusters of failures.

ACM Computing Classification System (CCS):
Software verification and validation → Software creation and management → Software verification and validation → Software defect analysis → Software testing and debugging

Avainsanat — Nyckelord — Keywords

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — Övriga uppgifter — Additional information

1. Terms and terminology

- **AWS** - Amazon Web Services
- **CAGR** - Compound Annual Growth Rate, a formula that calculates the rate of return needed for an investment to grow from its beginning balance to its ending balance [7].
- **CI** - Continuous Integration is a software development practice that involves regular commits and automated tests like regression testing to ensure the stability of the software after each commit [8].
- **CSV** - Comma Separated Values.
- **DSR** - Design Science Research, a set of principles that guide research conduct [47].
- **HDFS** - Hadoop Distributed File System.
- **IVVES** - A EU-backed project that focuses on applying AI to evolving systems like software [14].
- **IRP** - Intellectual Property Rights
- **Log parsing algorithms** - Algorithms that parse software logs or test logs using different approaches, including traditional regular expression extraction and more contemporary like deep learning [36, 45].
- **LCS** - Longest Common Sub-sequence.
- **OSS** - Open Source Software, software released open-sourced license, usually available and freely editable by anyone.
- **Software log** - Software log is an artefact produced by software during it is run time. Software logs contain various information regarding the software execution and can be utilised for different tasks, such as fault detection and debugging [75].

- **String Similarity Algorithms** - Algorithms that compare two strings using different methods. Often string similarity algorithms return a score between 0 and 1, denoting how similar the strings are [57, 51, 58].
- **SLCT** - Simple Logfile Clustering Tool.
- **TAP** - Test Automation Platform.
- **Test log** - Much like software, test logs are often an artefact of a testing framework. Test logs contain information on the execution of the tests. Additionally, some frameworks log a stack trace of a failure in case of failure in a test [25].
- **Test suite** - The test suite contains multiple tests within. Tests can be grouped in test suites, for example, by a common theme [53].
- **Test** - In the context of this master's thesis, a test is referenced to a singular automated unit test written using testing automation frameworks. These tests ensure that a smaller part of a software, for example, a function works according to the design [4].
- **Testing Framework / Test Automation Framework** - A framework used to write and execute tests within, for example, a Continuous Integration pipeline. Frameworks like Pytest or XUnit support the export of test logs in various formats [24].
- **TFA** - Test Failure Analysis, the main artefact produced by this master's thesis.
- **WithSecure** - A cyber security company based in Finland that was a key partner in solving the problems tackled in this master's thesis work [15].

Contents

2. Introduction

The software development market is valued at $474.61 billion and is projected to grow to $1,153.7 billion by 2030 [6]. The global big data and business analytics market size was valued at $198.08 billion in 2020 and is projected to reach $684.12 billion by 2030, growing at a CAGR of 13.5% from 2021 to 2030 [5]. One of the ways to improve business efficiency, software application and infrastructure performance, and up-time is log analysis [29]. O'Reilly's "Understanding Log Analytics at Scale" report covers the common best practices and considerations that can guide architects during the planning process [27]. This master's thesis was done as part of the IVVES project [14] and in collaboration with WithSecure cyber security software company [15]. IVVES aims to find ways of applying AI and machine learning to evolving systems, like software. WithSecure is a respected software company headquartered in Helsinki, Finland that focuses on delivering cyber security solutions to individuals and big corporations. WithSecure has over a thousand employees across the globe and quarterly revenue of hundreds of millions of dollars, making this company a respectable and experienced partner for the work done in the IVVES project and this master's thesis [30].

Because of the goal of the IVVES project, the work in this master's thesis started from a question: "How can modern data science applications help software engineers save time?". To answer that question, this work introduced a novel concept of comparing failures found within test logs using string similarity algorithms to achieve clusters of failures that indicate similar failures. To analyse the concept, a tool named "Test Failure Analysis" (TFA), published as a Pip package and as open-source software (OSS) on GitHub, was developed [11, 10]. Validation proceeded using three distinct data sets across three implementation approaches using five different string similarity algorithms. Later, TFA was deployed into a Continuous Integration (CI) pipeline of an open-source project [19]. These approaches and algorithms are introduced later in the work.

Logging is an essential and highly valued practice in software development [41]. Software engineers can deem some information vital during software development and log it for various reasons [61]. For example, logs can depict a problem in the software run time, record states of the software or assist in debugging procedures [64].

In software development, using software logs to improve the software or software

development process is an old concept and has been researched extensively. Utilising information concealed inside the software logs can be useful and sometimes important for many various tasks. Among the typical usage of logs are anomaly detection, performance issues, debugging, and fault detection [32, 39, 48, 59, 71, 78, 54].

Continuous integration (CI) is a modern software development practice used by many companies, e.g. Google [8, 68]. Tests play an essential role in the CI pipeline [38, 76]. Essential to CI are automated tests that ensure system stability [28]. Automated tests are often executed by testing frameworks such as Pytest or RobotFramework [20, 22]. Good frameworks produce detailed logs of test execution. Automated tests can often fail with high counts and for various reasons [55]. Software engineers face analysing hundreds and thousands of failed tests daily as a real-world challenge. The overwhelming number of logs due to the executed tests and the laborious and error-prone task can prove hard to tackle for an engineer [84]. Rerunning a test after every fix is not always an option as it can be a costly endeavour [63].

During discussions with the software engineers from WithSecure, these problems became apparent. It is up to software engineers to identify the failures causing the test to fail and fix them. Before a fix can be done, a software engineer interprets which failures are the same so that the fix has the most impact. The mentioned undertaking is cumbersome, time-consuming, error-prone, and is rarely done so that every log file is analysed as the sheer amount of logs can be overwhelming for a human and take up days worth of work.

The TFA tool developed in this work aimed to automate the work of manual investigation work by comparing test logs with each other and finding and presenting clusters of similarly failed tests. These clusters guide software engineers in selecting failures for fixing. Using clusters to identify failures within the test logs leads to more impactful fixes of an underlying issue, leading to a more reliable CI pipeline and, eventually, a more reliable software. Therefore the first research question is:

- **Research Question 1 (RQ1)** - To what extent is the concept, in the form of the software tool, applicable to address the log analysing challenges?

The efficiency of software development has always been important. Particularly a CI needs to match a flexibility level and a high demand for fast cycles. Therefore the second research question is:

- **Research Question 2 (RQ2)** - How does the concept, in the form of the software tool, improve the performance of log analyses?

The work has been conducted through a series of phases which may be described as follows:

- Development of the concept that utilises string similarity algorithms to cluster failures within test logs.
- Validation of the tool by exposing the tool to three different data sets with unique characteristics and scoring the data sets with five string similarity algorithms across three alternative approaches.
- Implementation and analysis of the concept as an open-sourced software project and a pip package called "Test Failure Analysis".
- Deployment of the tool into an open-source projects CI pipeline and results from that experiment. Results show that applying the developed in this work TFA tool increases the performance of the log analyses by 99.5%

Part of this work is to be published and presented in a peer-reviewed publication in Springer Book Series: Advances in Intelligent Systems and Computing [56]. That publication focuses on one of the approaches described in this work. The TFA has been received within WithSecure and is planned to be implemented in various in-house projects.

The rest of the thesis is structured as follows. The methodology chapter presents principles according to which the research of this work has been done. The related work chapter gives an overview of studies relevant to the work and opens more deeply to illustrate the scope of the problem. The Concept introduces the concept in more detail and ways to solve the problem. The implementation chapter dives deeper into the implementation of the concept in the form of the TFA tool. That chapter presents the implementation of all three approaches. The "Quantified results of the three approaches" chapter presents quantitative results from applying five different string similarity algorithms on three different data sets between the three approaches. Results from applying the TFA to an open-source project" chapter provides results of the tool's integration into an open-sourced project's CI pipeline. The "Discussion" chapter brings an overview of the issues associated with every approach, strengths and shortcomings and explains why specific approaches are picked over others. In that chapter, the discussion about the experiment with the open-sourced project continues. By the end of the chapter, answers to both research questions are given. The concluding remarks are given in the Conclusion chapter.

3. Methodology

This work has been implemented according to the Design Science Research principles. Design science research (DSR) is a research paradigm in which a designer develops novel products to address issues about human problems and adds new information to the body of scientific data; the developed artefacts are essential to comprehending that issue as well as being helpful [46]. A design challenge and its solution are learned and understood through the creation and use of an artefact, according to the fundamental principle of DSR.

To acknowledge and understand the design problem, weekly peer communication with software engineers from WithSecure has been conducted from the beginning of the work. The software engineers involved in the communication were of different levels of seniority, from developers up to a team lead. All the engineers mentioned above are involved in the software development processes, which is the primary business of WithSecure company. All software engineers have more than ten years of experience in software development. As has been stated in Chapter2, analysis of software logs is essential to developing quality software and therefore appeared to be the primary challenge of this work.

During the early stages of communications, it has been discovered that the problem of log analysis cannot be solved by business organisation changes, strategies or processes; instead, a software tool is required to address this problem. The performance of the software is the primary consideration of the tool. The primary measure for the version of the software is time used to analyse and identify clusters of failures under the acceptable level of errors. All these requirements were wrapped up the problem relevance and therefore were reflected in the **Research Questions** of this master's thesis:

- **RQ1:** To what extent is the concept, in the form of the software tool, applicable to address the log analysing challenges?
- **RQ2:** How does the concept, in the form of the software tool, improve the performance of log analyses?

The ultimate objective of this work is the software artefact that implements the

concept of answering both of the research questions, satisfying the requirements of the WithSecure company while also being generic enough to be applicable in the broad range of use cases where similar problems are encountered.

The new knowledge has been generated based on the experience of the software engineers and information obtained from the literature review processes. Literature review processes have not been limited to scientific literature only but professional and specialised software development forums, blogs and other sources of information typical used by software engineers. Peer communication sessions and the literature review results were documented as research notes and discussed with the project supervisor in the following peer communication sessions.

Design evaluation methods were not known before the start of the work. None of the information sources brought any appropriate methodology for evaluation. Therefore the evaluation method had to be designed as part of the work, and the efficiency of the design had to be confirmed with WithSecure. One requirement is to achieve the artefact's ability to form cluster failures even if they are not in the desired order. The other requirement is to measure the artefact's performance by processing the set of logs compared to the time needed to process the same set of logs by a human. The artefact performance was measured automatically by measuring the time difference between two timestamps, one at the beginning and the other at the end of the execution process. The human performance numbers were derived from statistical data collected by WithSecure over the past years from exactly the software development process intended to be improved.

The dissemination of work has been planned through scientific publication and publishing the source code of the artefact on Github. Dissemination activities were intended to make the research artefact available to scientific and practitioner communities.

Table 3.1 shows seven guidelines for the DSR and the reflection of methodological particularities of this work.

The entire process of the work has been rather pragmatic with a strong emphasis on practicality and utility [72]. That is, both the concept and the implementation have been designed to fulfil the requirements and achieve a high level of utilisation.

Table 3.1: Reflection on the DSR guidelines

Number	Guideline	Reflection
1	Design as an Artifact	The artefact is the TFA which is a software tool implementing the concept of log clustering using the string similarity algorithms.
2	Problem Relevance	The artefact is built based on direct requirements set up by software development processes and problems encountered.
3	Design Evaluation	The evaluation is performed based on the requirements obtained through the design process.
4	Research Contributions	The clear and verifiable contributions are ensured by peer-review processes of the scientific publication and the availability of the code as OSS available for verification.
5	Research Rigor	Research Rigor has been reached by the variety of tests presented in this thesis and confirmed by the peer-review processes of the scientific publication.
6	Design as a Search Process	The available information sources, including peer communication and extensive literature review, have been utilised.
7	Communication of Research	The research process and results have been communicated not only to the development-oriented audience of software engineers but also to team leads and scientific communities.

4. Related work

Due to the massive increase in the number of logs, manual analysis of logs has become almost impossible [83]. A wide variety of automatic analysis methods have been researched. Available studies present different approaches applied to the logs to achieve clusters. One study proposes a data clustering algorithm for mining patterns from event logs [82]. The other study paper presents a tool called Simple Logfile Clustering Tool (SLCT) that uses a novel algorithm that uses a threshold value to display a summary of frequently used words in the logs to form clusters [23]. One more study proposes a clustering of logs using iterative partitioning [62]. In another study, the clustering is achieved using a novel algorithm called Iterative Partitioning Log Mining (IPLoM) to achieve clusters of event logs. Another work relied on SLCT to take further its achievements [73]. That approach also uses frequently used words in the logs but uses that part to learn failure symptoms by building a decision tree. Another algorithm, LogCluster, expands on the algorithm used by SLCT [84]. SLCT algorithm relies on finding patterns that occur at least n times before displaying them. This is done by considering the frequency of each word but disregarding the order they are in. After selecting the frequent words above the n threshold, LogCluster creates a cluster candidate list by iterating over the frequent words once more and increasing the count of event templates if the frequent words are common enough. The candidate list is then enhanced with heuristic models to solve a common problem of SLCT of overfitting. Clustering logs by transforming log lines into event templates or features is done by many papers that offer solutions that try to tackle clustering problems [77, 61, 69].

Test Failure Analysis tool differs from these approaches in many ways. First, TFA does not try to solve the general problem of analysing all types of logs but focuses on the specific problem of failure analysis within test logs provided by testing frameworks. Second, TFA does not extract event templates from the log line but instead treats the strings as such and compares them instead. Thirdly, TFA does not try to detect failures or anomalies in the logs but assists developers by pointing out similar failures. Fourth, log parsing presented in the above research papers relies on patterns, whereas failures can have very complex stack traces and appear only once. It is worth mentioning that SLCT and IPLoM were part of a survey study where popular algorithms that parse

logs into event templates were evaluated [88]. Two best-performing algorithms (Drain and Spell) were picked as an approach to trial in TFA.

It is seen from the above-presented review of the studies topics such as log analysis and data transformation of unstructured log files into structured data are inseparable. Therefore, it is worth looking into other log analysis applications and data transformation methods than clustering software logs.

Anomaly detection during software run-time using extraction of numerical data like counts of different states that the software visits during the run-time and pattern recognition of these system states. Changes in the patterns software states and counts would indicate a possible anomaly [40]. Another approach is training Generative Adversarial Networks to fabricate logs with anomalous data using log keys, after which neural networks can differentiate between fabricated and real anomalous logs in software logs. However, another approach is a DeepLog - an approach that utilises Long Short-Term Memory [37]. Log clustering can also be achieved via unsupervised learning methods [87]. Many studies that propose log parsing algorithms or approaches to anomaly detection talk about online anomaly detection [45], [61], [44], [52], [85], [36], [86]. Such online anomaly detection algorithms can parse logs into events and highlight an anomaly during the software's run time. As an alternative to log parsing algorithms, existing algorithms have been enhanced to parse log data. For example, an improved string edit distance-based algorithm and utilisation of support vector machines for training [50], [33]. In addition to anomaly detection, log analysis can also be applied to topics such as health monitoring of data storage systems [81] or identification of previously unseen performance issues [60] [71]. Even in techniques such as these, data transformation is a crucial step. Log parsing has also been bolstered with, for example, statistical analysis tools like Conformal Prediction [74]. That study proposes conformal prediction with log parsing algorithms to make event template extraction more reliable. In addition to software logs, test logs analysis has been utilised to some extent. For example, a study suggests improving regression testing by slicing big test suites into more minor test cases using test logs [66], predicting future bugs and failures by mining historical test logs [31] [80]. A whole testing framework revolved around log-based testing [49]. However, another study suggests a way to improve code coverage so that testing suites can be improved [34].

All the works presented in this chapter share a common challenge: "How to analyse huge amounts of data reliably and with added value?". That is precisely the same challenge that was also a problem that the TFA set out to solve.

5. Concept

Figure 5.1 is from Amazon's AWS website explaining the continuous integration development cycle [28]. There are variations to the cycle, but the main idea applies across. Software is developed, and code is written and merged into a central repository from which automated tests like regression tests are run [8]. As shown in Figure 5.1, tests play a significant role in two parts of the cycle: build and staging. In the build stage, tests related to new features are run in addition to unit tests. In the staging phase, load tests and tests related to integration are conducted. Both stages may utilise a testing framework to plan, run and execute their tests. These tests are run as part of software developed by a continuous integration practice. The tests executed in both of these stages produce test logs. Depending on the testing framework, the logs may differ.

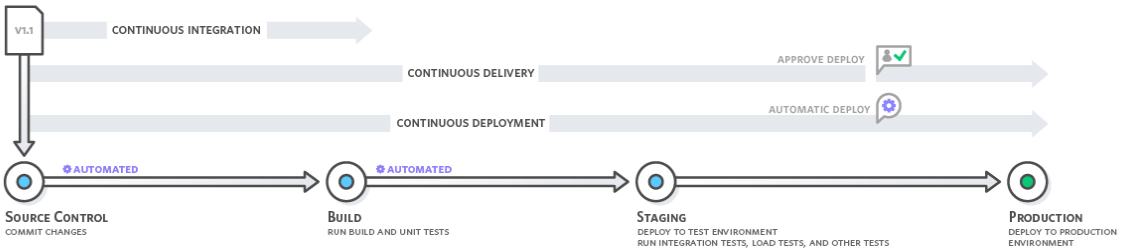

Figure 5.1: Amazon AWS example of continuous integration development cycle

Projects with CI pipelines that run many tests can produce vast amounts of log data. For example, an Aliyun Mail (Alibabas's production e-mail system that provides free e-mail service to the public) generates thirty to fifty gigabytes of logs per hour (120-200 million lines) [2, 1, 67]. The overwhelming number of logs and the laborious and error-prone task can prove hard to tackle for an engineer. A regular expression can alleviate this problem by filtering out some information but require prior domain knowledge by a developer of the system [45]. For example, sophisticated tools used in-house at WithSecure can aggregate the failing tests by the count and visualise information conveniently for the software engineers. Unfortunately, even a single change of character in the failure can throw off these tools as these morphed failures are considered unique by these tools. For example, this sort of tool would

mark a failure with the following text "Process ID: 123" different from a failure with "Process ID: 124". These two log lines are different, but in the context of the failure, they maybe are an output of the same underlying issue. While a software engineer can fix the failing test with the most failures, it would leave other tests caused by the same failure potentially undetected and left there to be discovered after the tests are rerun. Reruns can cost money and time for companies. For example, Google has developed their Test Automation Platform (TAP) [26]. The goal of TAP is to run automated tests smartly. Tests are run after every commit, and commits at Google are frequent (almost one commit every second) [63]. As it stands right now, TAP runs automated tests every 45 minutes. These 45-minute cycles mean the engineer must wait 45 minutes to see their commit pass or fail the tests in the best-case scenario.

In order to avoid unnecessary reruns, a software engineer is left with analysing and grouping the failures manually. As explained by software engineers at Withsecure, grouping is a task in which a software engineer tries to identify tests that fail similarly from hundreds of test logs. This task takes time and is easily error-prone, especially since the software logs and test logs come in all shapes and forms, require domain knowledge and can be daunting due to the enormous amounts of logs.

At its core, the concept proposed by this work is the following: failures that fail similarly have a higher similarity score than those that fail differently. Using these scores, clusters of failures start to appear. These clusters guide software engineers in the correct direction to choose to fix failures that have the most impact.

TFA works by applying a string similarity algorithm on two strings. In this context, these two strings are failures. String similarity algorithms output a similarity score between zero and one, where zero denotes a nonexistent similarity between strings and one, meaning that the strings are identical. Pairs of failures with high similarity scores mean that the failures are similar. This, in return, means that they most likely share a common underlying issue and can be resolved with one fix.

To validate the concept, three distinct data sets were artificially created from a tool currently running in the production environment at WithSecure. The following subsection describes the data sets in question.

5.0.1 Data set description

Data sets used to validate the concept originate from another software that is being used daily at WithSecure in the production environment. The validation phase of the tool consisted of evaluating the TFA tool on the three data sets. Data sets in the context of this master's thesis are test logs. Such a test log can contain multiple test runs, and within a test run can be many individual tests. Each test is placed in an

Table 5.1: Data set 1

Log name	Failure in
pass_1	-
pass_2	-
pass_3	-
pass_4	-
fail_same_error_1	test_02
fail_same_error_2	test_02
fail_same_error_3	test_02
fail_same_error_4	test_02
fail_diff_test_1	test_03
fail_diff_test_2	test_03
fail_diff_test_3	test_03
fail_diff_test_4	test_03
fail_diff_error_1	test_02
fail_diff_error_2	test_02
fail_diff_error_3	test_02
fail_diff_error_4	test_02

individual log file, with this being said, tests and log files can be used interchangeably in this context.

Each test contains only one failure. Cases where multiple failures are present within one test are not impossible but rare enough that it was deemed unnecessary to account for. Each data set consisted of 150 000 individual test log lines. In total, resulting in about 450 000 log lines for concept validation.

Table 6.2 shows an example of what some of the lines from a random data set may look like. Due to Intellectual Property Rights (IPRs)s, some of the content has been redacted.

The first data set contained sixteen different tests or log files. Four of those test files were the result of the passed tests. Twelve of those were the results of the failed test runs. The second data set contained eleven different tests. Three of those log files were the result of the passed tests. Eight of those were the results of the failed test runs. The third data set contained twelve different tests. All of the files had results of failed test runs. Tables 5.1, 5.2 and 5.3 show log names and the name of the failing test.

Table 5.2: Dataset 2

Log name	Failure in
pass_1	-
pass_2	-
pass_3	-
fail_same_error_1	test_01
fail_same_error_2	test_01
fail_same_error_3	test_01
fail_diff_test_1	test_02
fail_diff_test_2	test_02
fail_diff_error_1	test_01
fail_diff_error_2	test_01
fail_diff_error_3	test_01

5.0.2 Type of failures

Every data set had a slightly different failure setup. The purpose is to mimic the possible failure types in the production environment. The goal was to expose the tool to various failures before proceeding further. Due to IRPs, failure stack traces cannot be showcased as such, nor can they be shown as redacting the failure stack traces, as they would lose much of their meaning.

First data set

The first data set ensures the tool can differentiate between two similar failures. Failures in all the categories are almost identical and have only minor differences. Ideally, the TFA would notice the discrepancies and score them accordingly. The similarity score is expected to be close between the failures.

Second data set

The second data has failures that are significantly different from each other. The tool should have an easier time scoring these failures.

Third data set

The objective of the third data set is to expose the tool to possible complex failures that might emerge during tests. These failures have lengthy stack traces and innumerable amounts of special characters.

Table 5.3: Dataset 3

Log name	Failure in
fail_same_error_1	test_01
fail_same_error_2	test_01
fail_same_error_2	test_01
fail_diff_test_1	test_02
fail_diff_test_2	test_02
fail_diff_test_3	test_02
fail_diff_error_1	test_01
fail_diff_error_2	test_01
fail_diff_error_3	test_01

5.0.3 Data transformation

The related work chapter showed the importance of the data transformation topic. Three different approaches were trialled during the validation phase of TFA before settling on the final one.

In the first approach (from now on, Approach 1), the tool utilised state-of-the-art log parsing algorithms (Drain and Spell) on test logs produced by the Pytest testing framework. This action is done to parse unstructured data into a structured format such as pandas data frames for further action [36], [45]. The Pytest testing framework was picked as the framework to use because of two reasons. First, the Pytest testing framework is the main testing framework used by the WithSecure projects. Second, Pytest is a popular testing framework. Pytest is at 10 million weekly downloads between June 2020 and April 2022 and has an average of 47 commits per week [21]. These numbers make Pytest a worthwhile package to augment and develop further.

In the second approach (from now on, Approach 2), TFA uses manual parsing of test logs produced by the Pytest testing framework to transform unstructured test logs into a structured format such as pandas data frames. Manual parsing is an attempt to utilise clear patterns in the Pytest testing framework and create a parse from scratch.

In the third approach (from now on, Approach 3), TFA uses .xml files produced by the Pytest testing framework to achieve structured data frames. The tool ended up using a third approach in the final version.

5.0.4 Type of clusters

Using the data sets and the approaches described above following categories were identified with the help of software engineers at WithSecure. These categories aim to serve

as the benchmarks of string similarity algorithms during the validation phase of the tool. In other words, after applying string similarity algorithms to the data sets between the three different approaches, the results similarity scores should fall in the categories specified below and, by doing so, validate the concept's validity and the tool's functionality.

For Approaches 1 and 2, similarity scores consist of two values. The first is an overall similarity score between the two tests, corresponding to the category numbers. The second is the failure similarity if the failure exists in the test run where two failures are scored against each other. For Approaches 1 and 2, the categories are the following:

Category 1 - C1

Category 1 (C1) - Same failure, same test: Similarity score is high between a failing test in a test run and the same failing test in a different test run if the failure is similar.

Category 2 - C2

Category 2 (C2) - Same failure, different test: Similarity score is high between two tests in different test runs that fail similarly. Additionally, it is lower than Category 1.

Category 3 - C3

Category 3 (C3) - Different failure, same test: Similarity score is low between two tests in different test runs if the failure is different. Additionally, it is lower than in Category 2 and Category 3

For Approach 3, the similarity score consists of only one value; failure similarity. In Approach 3, the only calculable score is the similarity between two failures if they are present. For Approach 3, the subcategories are the following:

Category 1.1 - C1.1

Category 1.1 (C1.1) - Same failure: Similarity score is high between the same failures in different test runs.

Category 2.1 - C2.1

Category 2.1 (C2.1) - Same failure: Similarity score is high between the same failures in different test runs.

Category 3.1 - C3.1

Category 3.1 (C3.1) - Different failure: Similarity score is low between two failures if the failure is different within the same test in different test runs.

Relationship between categories

Let us denote Categories 1,2, and 3 as $C1$, $C2$ and $C3$ and subcategories 1.1, 2.1, and 3.1 as $C1.1$, $C2.1$ and $C3.1$. The relationship between the similarity scores of those categories can be summarised in equation 5.1 as such:

$$(C1 > C2 > C3) \wedge (C1.1 > C2.1 > C3.1) \tag{5.1}$$

The relationship between subcategories for Approach 3 can be summarised in equation 5.2 as such:

$$C1.1 > C2.2 > C3.1 \tag{5.2}$$

5.0.5 String similarity algorithms

Table 5.4: Algorithms

Name	Type
Sequence Matcher	Sequence-based algorithm
Jaro-Winkler distance	Edit distance-based algorithm
Levenshtein Ratio	Edit distance-based algorithm
Jaccard Index	Token-based algorithm
Cosine Similarity	Fuzzy matching algorithm

Algorithms that are used and analysed in the validation phase are Sequence Matcher from Pythons diff library [9]. Jaro-Winkler distance [35], Jaccard score [51], cosine similarity [43] and Levenshtein ratio [58].

String similarity algorithms accept two strings and return a similarity score value. A similarity score is a number between zero and one on how similar these two strings are. Zero means that strings are not similar, and one means that strings are identical.

Table 5.4 describes the algorithms and their type. The algorithms were chosen based on the type they belong to, their ease of access and the frequency they appeared in popular research papers and popular professional forums. Extensive comparison, shortcomings and strengths of each algorithm fall outside the scope of this work. Nevertheless, a small experiment on the algorithms was conducted as part of this work.

The experiment compares four different strings using these five similarity algorithms. Results from the experiment are visible in Table 5.5.

Sequence Matcher uses the longest common subsequence (LCS) algorithm as its base. This means that if the failure is present in two logs and they are similar, they share a long common sequence of characters, meaning the Sequence Matcher will score two failures highly. For example, an S1 and S2 from 5.5 have an LCS of 55. Jaro-Winkler and Levenshtein are edit distance-based algorithms, meaning they calculate how many edits from S1 need to be made to achieve S2. In this example, six edits are needed to achieve S2 from S1. Jaccard Index works by splitting a string into tokens and calculating how many similar tokens are between two strings. If the test logs have similar failures, the tokes will also be similar, meaning the score will also be high. Lastly, cosine similarity first turns strings into vectors and then calculating a cosine angle between vectors outputs a score of how similar the two strings are. The advantage is that while the strings can be very far apart on euclidean distance, they might be very close in cosine similarity.

Table 5.5 shows scores of five different algorithms when applied to the combinations of these four strings extracted as an example from HADOOP log [3]s.

- String 1 (S1) - "Recalculating schedule, headroom=<memory:10240, vCores:-17>"
- String 2 (S2) - "Recalculating schedule, headroom=<memory:8192, vCores:-19>"
- String 3 (S3) - "Assigned container container_1445144423722_0020_01_000006 to attempt_1445144423722_0020_m_000004_0"
- String 4 (S4) - "Auth successful for job_1445144423722_0020 (auth:SIMPLE)"

Table 5.5: Results from applying string similarity algorithms applied on example strings

	SM ratio	**Cosine**	**Jaccard**	**Jaro-Winkler**	**Levenshtein**
S1 & S2	0.940	0.714	0.848	0.960	0.940
S1 & S3	0.165	0.00	0.486	0.463	0.242
S1 & S4	0.208	0.00	0.369	0.490	0.295
S2 & S3	0.128	0.00	0.384	0.432	0.205
S2 & S4	0.122	0.00	0.291	0.436	0.263
S3 & S4	0.376	0.00	0.435	0.536	0.389

In Table 5.5, strings one and two are the most similar, and the algorithms show that with high similarity scores. Results are in line with what can be observed.

Interestingly, cosine similarity reports a 0.00 similarity score across every string apart from the first and second. At the same time, the other string similarity scores, like Jaro-Winkler, show a 0.536 similarity score between strings three and four. What the cosine similarity algorithm is showing is in line with what can be observed. No other string apart from the strings one and two are the same. Another two well-performing algorithms seem to be pythons diff library and, more precisely, the similarity ratio of a sequence matcher and Levenshtein ratio. They seem to report low enough scores for every other string except strings one and two.

5.0.6 Summary

The benefits of the presented concept are the following:

- This concept removes the problem of having many unique errors with only one character difference as the similarity will be represented on a continuous scale using numbers between zero and one.
- This concept iterates over all the logs and failures, thus always covering all the files and removing the need to rely on software engineers to pick correct log files.
- This concept does not require a software engineer's prior domain knowledge of the whole system; instead, even junior software engineers can focus on small groups of failures that can be addressed.
- The focus of the concept is only limited to a small subset of log types instead of trying to parse and analyse all sorts of logs.
- Even if the categories are not in the specified order, clusters still provide value to software engineers.

6. Implementation

This chapter contains the implementation details of the three approaches described in the chapter 5 - Concept. Each approach described below aims to turn the unstructured log files into a structured format. The desired format for TFA is the Pandas data frame.

6.0.1 Approach 1: Log parsing algorithms on Pytest frameworks test logs

A thorough study of the most popular log parsing algorithms has been conducted, in which thirteen different log parsing algorithms were evaluated and ranked [88]. An outcome of the study is an open-source tool that gives access to the implementation of the open-source algorithms for researchers and developers [18]. Log parsing algorithms were trialled on the test logs in this work to transform unstructured data into a structured format.

The following Table 6.1 is adapted and redacted from [88]. The table describes which algorithms were part of the survey study and how they turned unstructured log data into a structured format by extracting event templates.

According to the study mentioned above, the best performing algorithms were Drain, and Spell [88]. Based on this, an attempt was made to utilise both of these algorithms on the test logs. These two algorithms turn unstructured data into structured one by first parsing raw log messages using regular expression and then applying data mining models to extract variables from the log message and turn the log message into log event templates. Variables in the log messages are values that change based on the software execution, for example, a date and a timestamp, job ids or printable variables in the code. The event template part is considered a "hard-coded" part. The hard-coded part is observable in the logs due to software executing a line of code similar to "System.debug" or "print". For example. Figure 6.1 shows a line from the HDFS Hadoop system log and how a log parsing algorithm could extract an event template. The hard-coded part, or in other words, an event template part, is left unchanged. The variables part is replaced with "<*>".

According to the study, Drain & Spell can reach a very high parsing accuracy

Table 6.1: Popular log parsing algorithms in [88]

Algorithm	Technique	Reference
SLCT	Frequent pattern mining	[82]
AEL	Heuristics	[87]
IPLom	Iterative partitioning	[62]
LKE	Clustering	[40]
LFA	Frequent pattern mining	[70]
LogSig	Clustering	[79]
SHISO	Clustering	[69]
LogCluster	Frequent pattern mining	[84]
LenMa	Clustering	[77]
LogMine	Clustering	[42]
Spell	Longest common sub-sequence	[36]
Drain	Parsing Tree	[45]
MoLFI	Evolutionary algorithms	[65]

rating in log parsing on a certain type of log lines [88]. Parsing accuracy is defined as the "parsing accuracy (PA) metric as the ratio of correctly parsed log messages over the total number of log messages". [88]. After parsing, each log message has an event template corresponding to a group of messages of the same template. A log message is considered correctly parsed if and only if its event to the same group of log messages as the ground truth does" [88]. Due to simple template formats, the 100% log parsing accuracy can be reached on Apache and HDFS data sets. Unfortunately, the study also mentions that log parsing algorithms, including Drain & Spell, fall short on more complex data sets and would require improvements.

Both algorithms produce a .csv file containing columns extracted by the regular expression parsing and applying the algorithm. The columns in the output CSV file are as follows: date, time, debugging level, component, unparsed log message, the unique id of an event template, event template and extracted variables from the log message. The CSV can then be directly injected into the desired structure, the pandas data frame. From there, items from the log message column could be used as input to string similarity algorithms. Due to the poor performance of the log parsing algorithms, it was decided to focus on another approach instead of continuing with Approach 1. The next chapter describes the detailed results of applying the log parsing algorithms Drain and Spell on the test logs from the Pytest testing framework.

The goal of Approach 1 was to reduce individual log lines within the test logs to event templates by using patterns found. After that has been done, group them

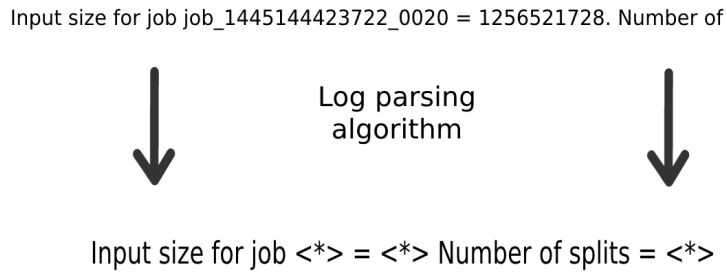

Figure 6.1: Log parsing algorithms functionality simplified

together and compare the only similarity of event templates against each other.

6.0.2 Approach 2: Manual log parsing of Pytest frameworks test logs

Alternative to the log parsing algorithm is the manual log parsing of Pytest framework test logs. Manual log parsing refers to an action where an attempt to implement a parser was trialled instead of using predefined methods of transforming the unstructured logs into a structured format. Table 6.2 depicts an example of logs found within a test log from the Pytest testing framework. Each line within the log file follows a pattern. The pattern consists of a timestamp in a year-month-day hour-minute-second, millisecond format. After that, a level of debugging. This part can take values such as" [INFO]"," [DEBUG]", or in a case of a failure present within a test -" [CRITICAL]". Lastly, the actual log message can contain keywords like" setup" and" teardown". Table 6.2 shows examples from multiple test logs used in this work. These logs are not in order and are there to illustrate better the point above. Due to the sensitive nature of text and privacy concerns, some parts of the log message have been transformed into *log message*s.

Given the reasons above, one can transform each line in the unstructured log files into a structured format such as a pandas data frame for deeper analysis. Line by line, the tool parses loglines using a built-in function of python ".split()" utilising "["and"]" characters into the structured data frame consisting of three columns. Those columns are the date and a timestamp, the logging level, and the message itself.

The second step in Approach 2 is to break test logs into individual, more granular test cases. Test log files can contain multiple tests, and the goal of TFA is to compare

Table 6.2: Example of Pytest logs from one of the data sets

testlog
2022-01-27 11:31:38,285 [DEBUG] *log message*
2022-01-27 11:31:38,285 [DEBUG] setup: *log message*::test_01
2022-01-27 11:31:38,300 [DEBUG] call: *log message*::test_01
2022-02-11 15:45:01,618 [CRITICAL] *failure stack trace*
2022-01-27 11:31:41,941 [DEBUG] teardown: *log message*::test_01

two individual tests against each other. The majority of the individual tests start with the" setup" keyword present in the log message, and the majority of the test ends with a" teardown" keyword present in the log message. TFA splits test logs into individual test cases by selecting all rows between log lines containing the abovementioned keywords and extracting the test name from the line with" setup" keywords. Additionally, TFA extracts the failure message. Failure is present if the logging level contains the "CRITICAL" keyword. All the lines are extracted from that row if such a keyword is present. As the TFA splits big test logs into an individual test case, it places them into a separate array; it also records from which logfile the test originated. The described actions lead to an array of data frames (*Arr.1*). The items in the *Arr.1* are considered to be individual tests from a bigger test log.

The next step in Approach 2 is the manual labelling of the log files. This step is done so that the output of TFA can be validated. Descriptive log names assist in manual labelling – Tables 5.1, 5.2, and 5.3 show this naming convention.

Before labelling can commence, *Arr.1* is permutated so that TFA can compare two individual tests. An "iter.tools" library achieves the permutation [16]. This step results in an array of tuples (*Arr.2*). Each item of the tuple is an individual test from a bigger test run log represented by a data frame. Using *Arr.2* manual labelling of the tests can be done.

Manual labelling goes as follows; *Arr.2* is iterated, and each item in the tuple has information from which log file it originated. If the log file name in both items in the tuple contains "same_error", assign a "category1" label. If the log file name of one item in the tuple is "same_error" and another item has "diff_test", set the "category2" label. Lastly, if the log file name of one item in the tuple is "same_error" and another item has "diff_error", assign the "category3" label.

Table 6.3 shows an example of one of the items from a random tuple. Column "log name" tells to which test log a test belongs. Column "timestamp" is the extracted timestamp. Column "level" describes the level of debugging. Column "log message" contains the log message. Column "test name," tells the name of the extracted test.

Lastly, column "category" includes the category to which the test log and test names belong.

To summarise, the Approach 2 works as follows:

- **Step 1:** Approach 2 takes N amount of log files in as an input.
- **Step 2:** Using clear pattern in the logs, for each log file: transform logs into structured format like pandas data frame using .split() function
- **Step 3:** Break down the newly made data frame into smaller data frames that represent individual tests within it using keywords like "setup" and "breakdown".
- **Step 4:** Newly created data frames are added to the array "Arr.1".
- **Step 5:** "Arr.2" is created. "Arr.2" contains permutations of two of "Arr.1" items. This array allows easy comparison and labelling. "Arr.2" is an array of tuples, each item in the tuple is a data frame.
- **Step 6:** Only in the validation phase: Label each item in the tuple based on the log name.
- **Step 7:** Score each tuple using the string similarity algorithm.

The goal of Approach 2 is to transform unstructured test log files into a structured format using patterns that are inherent to the way testing frameworks log information into test logs. From there, split the test logs that are in a structured format into individual tests, extract the failure from within the test logs if present, and lastly, compare these values against other test logs and tests within a test suite using string similarity algorithms.

6.0.3 Approach 3: The utilisation of XMLs files from the Pytest framework

A third approach was slightly different compared to the other ones. Instead of traditional test logs, TFA used XML files originating from the Pytest framework. The upside of these XML files is that they are already structured and thus do not need to require solving the problem of turning them into a structured format.

The test log files used in Approach 1 and 2 contained every logline that the software logs, whereas the XML captures only the log lines of a failure. XML files have the following structure. The root called "testsuites" contains many nodes called "testsuite". These individual test suite nodes contain "testcase" nodes that denote individual test cases which are run within a more extensive test suite. The test case

node can have a failure node in case of a failure. The failure stack trace is located within the failure node or inside the attribute called "message". The location of the failure depends on the testing framework. For example, the robot framework logs the failure message between nodes' start tag and end tag as a text message. On the other hand, Pytest logs the failure message within the attribute called "message". TFA supports both ways of logging the failure.

Both the root and subsequent nodes contain an attribute called "failures". This attribute possesses an integer value. If the value is zero, the tool can safely ignore the file as it does not contain any failures and is considered a pass. If the value is anything other than zero, then the path in the tree will eventually lead to a failure node.

Using the information described above and knowing the name of the critical nodes in the tree, TFA can parse an XML tree into a data frame which contains only relevant information for the tool. Information relevant to the TFA is the failure message, the test's name, and the log file's name. The procedure is repeated for each file, and each file that contains a failure is transformed into a data frame that is then placed into *Arr.1*.

From there, TFA can permutate the *Arr.1*, and the tool can achieve an outcome of an array of tuples like *Arr.2*.

To summarise, the Approach 3 works as follows:

- **Step 1:** Approach 3 takes N amount of .xml files as input.
- **Step 2:** For each of the file: if the file has failures in the root node "testsuites", continue. Otherwise, skip to the next file
- **Step 3:** From the leaf node that contains the failure, fetch the failure message, the test's name and the file's name. Put them into a data frame and stash that data frame into *Arr.1*
- **Step 4:** Create *Arr.2*. Populate *Arr.2* with permutations of two, of items from *Arr.1*.
- **Step 5:** Score each tuple in *Arr.2* using string similarity algorithms

The goal of Approach 3 is to utilise test logs in an XML format that some testing frameworks are capable of outputting. With the use of XML files, the data comes in an already structured format and allows easier access to the failure within the test logs. Once accessed, the failures can be compared in a similar manner to Approach 2 using string similarity algorithms.

Table 6.3: Example data frame after log prepossessing using approach 2 or 3

log_name	time_stamp	level	log_message	test_name	category
pass_1	*timestamp*	debug	*log message*	test_01	-
fail_same_error_1	*timestamp*	critical	*log message*	test_01	C1
fail_diff_error_3	*timestamp*	info	*log message*	test_99	C3
fail_diff_test_1	*timestamp*	info	*log message*	test_02	C2

Next steps

Each tuple in the *Arr.2* consists of two data frames which correspond to individual tests. For Approaches 1 and 2, the tool will compare and return a similarity score for 1) an overall similarity between logs of two tests and 2) similarity between the two failures if present within these two tests.

For Approach 3, the tool calculates only the failure similarity because the framework does not log the test logs entirely.

As an example, with the third approach, the tool iterates over each tuple, picks up the log message from the "log_message" column and passes it to the string similarity algorithms for scoring. The string similarity algorithm function then returns a value between 0 and 1 and assigns it to that particular tuple for further analysis or usage within the tool.

6.0.4 Outcome

The product of this work is an artefact called Test Failure Analysis tool built with Python. The final version of the tool is available as a pip package and can be contributed to at Github project as it is an open-sourced project [11, 10].

The tool can be invoked from the command line by calling the main script "failure_analysis.py". Alternatively, TFA can be integrated into a CI pipeline using tools like Jenkins or Github Actions [17, 13].

TFA accepts only one parameter in order for it to run; a path to the log files. The tool preprocesses and scores the logs using Approach 3 in the manner described in the previous chapters. Finally, the tool formats and presents an output. Figure 6.2 depicts an example output of the TSA tool. The tool outputs results directly into the terminal or command line. This manner of output guarantees eased access for developers and was requested by the software developers at WithSecure.

error	source test	similar test failures	file	Score
Lorem Ipsum is simply dummy text of the printing and	path.to.file.classWhichCanBe.test_name_is_also_long_to	path.to.file.test_01	xunit_1.xml	0.95
typesetting industry. Lorem Ipsum has been the industry's		path.to.file.test_02	xunit_1.xml	0.95
standard dummy text ever since the 1500s, when an unknown		path.to.file.test_03	xunit_1.xml	0.95
printer took a galley of type and scrambled it to make a		path.to.file.test_04	xunit_1.xml	0.95
type specimen book. It has survived not only five centuries,		path.to.file.test_01	xunit_2.xml	0.93
but also the leap into electronic typesetting, remaining		path.to.file.test_02	xunit_2.xml	0.93
essentially unchanged. It was popularised in the 1960s with		path.to.file.test_03	xunit_2.xml	0.93
the release of Letraset sheets containing Lorem Ipsum		path.to.file.test_04	xunit_2.xml	0.93
passages, and more recently with desktop publishing software		path.to.file.test_01	xunit_3.xml	0.92
like Aldus PageMaker including versions of Lorem Ipsum.		path.to.file.test_02	xunit_4.xml	0.91
Etiam nibh mi, eleifend sit amet dapibus et, hendrerit ut	path.to.file.classWhichCanBe.test_another_test_case_at	path.to.file.test_21	xunit_1.xml	0.95
metus. Nulla nec eros eu velit porta venenatis consequat in		path.to.file.test_22	xunit_1.xml	0.95
dolor. Fusce molestie ultricies augue, non congue ligula.		path.to.file.test_23	xunit_1.xml	0.95
Maecenas id nunc et mi rhoncus vulputate. Aliquam erat		path.to.file.test_24	xunit_1.xml	0.95
volutpat. Nunc id lectus auctor, sodales nisl condimentum,		path.to.file.test_21	xunit_2.xml	0.93
convallis diam. Ut at gravida nulla, sit amet ultricies odio.		path.to.file.test_22	xunit_2.xml	0.93
Duis eleifend nisl nulla, sit amet placerat neque		path.to.file.test_23	xunit_2.xml	0.93
sollicitudin eget. Mauris orci dui, dapibus nec enim eu,		path.to.file.test_24	xunit_2.xml	0.93
consequat imperdiet dolor.		path.to.file.test_21	xunit_3.xml	0.92
		path.to.file.test_22	xunit_4.xml	0.91

Figure 6.2: An example of the tools output

7. Quantified results of three approaches

This chapter provides numerical contexts for the research questions identified in the Introduction chapter for all three approaches. The main research questions will be answered in the Discussion chapter utilising results presented in this and the following chapters.

This work aimed to answer the following research questions:

- Research Question 1 (RQ1) - To what extent is the concept, in the form of the software tool, applicable to address the log analysing challenges?
- Research Question 2 (RQ2) - How does the concept, in the form of the software tool, improve the performance of log analyses?

Quantifiable results are an outcome of applying the five-string similarity algorithms in each approach to three different data sets introduced in the chapter 5 - "Concept". For Approach 2, results are analysed by plotting the score similarities calculated by similarity algorithms on an x and y-axis. An overall test similarity is plotted on an x-axis, and failure similarity is plotted on a y-axis. Plotting these values would form three distinct groups where each group represents categories described in the Concept section.

The best performing algorithm in Approach 2 is chosen by answering the following questions:

- Question 1 (Q1) - Are the three categories identifiable?
- Question 2 (Q2) - Does the equation 5.1 hold, e.g., are the categories in the correct order?
- Question 3 (Q3) - What is the mean distance between the centre of each category?

For Q1, if the categories are easily identifiable and can be grouped without any doubt, it is considered positive. For Q2, if the algorithm scored the groups in an order

specified in equation 5.2, it is considered positive. For question 3, the higher the mean distance between clusters, the better it is. Higher distance means that the clusters are further apart in a significant matter.

For Approach 3, results are analysed by plotting the score similarities calculated by similarity algorithms on an x and y-axis. It is established that each .xml file contains only one failure in one of the test cases. It has also been established that *Arr.2* has permutations of two individual tests. With these assumptions, the plot will show scores for each of the logs with each other. The ball's size denotes the failure similarity's high, and the ball's colour denotes the category.

The best performing algorithm in Approach 3 is chosen by answering the following questions:

- Question 1 (Q1) - Does the equation 5.2 hold, e.g., are the categories in the correct order?
- Question 2 (Q2) - What is the mean range between categories 1 and 3?

For Q1, if the algorithm scored the groups in an order specified equation 5.2, it is considered positive in this work for an algorithm in question. For Q2, the bigger the range is between categories one and three, the further apart the categories are, meaning fewer mistakes to confuse the categories.

Unfortunately, Approach 1 did not reach the same stage of the validation phase of applying the string similarity algorithms as did Approach 2 and 3. The reason for this will become apparent in the following sub-section.

7.1 Approach 1: Log parsing algorithm approach

Results presented in the survey research paper required manual labelling of the data in the event templates to measure the parsing accuracy of Drain and Spell. Labelling was achieved with the help of people with domain knowledge of the systems that produced the logs. Unfortunately, it was impossible with the data sets described as the sheer volume of the test logs and the diversity between individual lines was great. With this in mind, an alternative way to evaluate parsing accuracy was devised for the data sets in question.

Manual analysis is done before applying Drain and Spell to the data sets mentioned earlier. The manual analysis aimed to provide insight into the logs to understand better what kind of event templates are expected. Manual analysis was achieved by sifting through the logs by hand. Manual analysis revealed that most of the lines in the test logs have variables, meaning that each parsed string using test log parsing

algorithms should contain extracted variables. Due to IPRs, this thesis cannot show exact log line message lines.

With that in mind, the two following metrics evaluated the results of log parsing algorithms. In the first metric, % of log lines without any extracted variables, a higher percentage means a poor performance of the log parsing algorithm on the data set. Second, the number of event templates higher count of event templates indicates a worse performance of the log parsing algorithm on the data set.

Table 7.1 shows these results for both Drain and Spell algorithms on the data set.

Table 7.1: Drain and Spell log parsing results

Algorithm	% of templates without variables	Event template count
Drain	36%	1525
Spell	26%	290

Spell produced significantly fewer event templates than the Drain. It still left a lot to be desired in terms of log parsing. While both Drain and Spell correctly extracted other features from the log lines, such as a date timestamp and debugging level, both algorithms struggled with complex log messages. Complex messages were long, contained various special characters like ":", "&", "-", "_" and sometimes utilised the camel-case way of writing the name of the function. These types of messages proved challenging to parse for both Drain and Spell. The results observed are in line with results presented in the survey [88]. Additionally, both Drain and Spell algorithms extracted test names that were not the names of the tests but rather something else. Most of the test names that these log parsing algorithms parsed were valid; nonetheless, 10% of all the tests were something other than the test name on one of the test logs. The amount of uncertainty was not acceptable for a TFA. Because of the results presented here, it was decided to drop experimentation on Approach 1 and focus instead on other approaches.

7.2 Approach 2: Manual parsing algorithm approach

In this section, the results of Approach 2 are presented.

7.2.1 Data set 1 results

Results of applying string similarity algorithms on data set 1 with Approach 2 are depicted in Figure 7.1.

Sequence Matcher algorithm

- Q1 - No. While Category 1 is separate from Category 2 and Category 3, Category 2 and Category 3 present instances that can be mistaken for another category.
- Q2 - No. Category 2 is below category 3 in scores, meaning equation 5.1 does not hold.
- Q3 - Mean distance between categories is at 0.5272.

Cosine Similarity algorithm

- Q1 - Yes. Three distinct categories are observed.
- Q2 - No. Category 2 is below category 3 in scores, meaning equation 5.1 does not hold.
- Q3 - Mean distance between categories is at 0.5260.

Jaccard Index algorithm

- Q1 - No. Categories 1 and 2 are mixed up.
- Q2 - No. Category 3 overlaps with categories one and two and is scored at the same level as category one and two instances. Equation 5.1 does not hold.
- Q3 - Mean distance between categories is 0.0807

Jaro-Winkler distance algorithm

- Q1 - Yes. Three distinct categories are easily observed, with one instance of category 3 getting very close to the category 1 cluster.
- Q2 - No. Category 3 instances score above category two, meaning that the equation 5.1 does not hold.
- Q3 - Mean distance between categories is at 0.1433.

Levenshtein distance algorithm

- Q1 - Yes. Three distinct categories can be observed
- Q2 - No. Category 2 is below category 3 in scores, meaning that equation 5.1 does not hold.
- Q3 - Mean distance between categories is at 0.3606.

7.2.2 Data set 2 results

Results of applying string similarity algorithms on data set 2 with Approach 2 are depicted in Figure 7.2.

Sequence Matcher algorithm

- Q1 - Yes. Three distinct categories can be observed.
- Q2 - Yes. While there is no overlap between categories, the instance is quite spread out, which might lead to difficulty in identifying groups without labels. Equation 5.1 holds.
- Q3 - Mean distance between categories is at 0.3761.

Cosine Similarity algorithm

- Q1 - Yes. Three distinct categories are observed.
- Q2 - Yes. Categories are in the correct order, and equation 5.1 holds.
- Q3 - Mean distance between categories is at 0.1135.

Jaccard Index algorithm

- Q1 - No. Some instances of category three overlap with category one, making category three very difficult to group.
- Q2 - No. Category 3 overlaps with category 2.
- Q3 - Mean distance between categories is at 0.0833.

Jaro-Winkler distance algorithm

- Q1 - No. Three distinct categories are easily observed.
- Q2 - No. Categories are in the wrong order, and equation 5.1 does not hold. Category 1 overlaps with category two, and category 2 overlaps with category three on the x-axis.
- Q3 - Mean distance between categories is at 0.1051.

Levenshtein algorithm

- Q1 - Yes. Three distinct categories can be observed
- Q2 - Yes. Categories are in the correct order, and equation 5.1 holds. Some instances of category one overlap with category two on the x-axis
- Q3 - Mean distance between categories is at 0.2014.

7.2.3 Data set 3 results

Results of applying string similarity algorithms on the data set 3 with Approach 2 are depicted in the Figure 7.3.

Sequence Matcher algorithm

- Q1 - No, it is challenging to differentiate categories 1 and 2.
- Q2 - Yes, categories do not overlap and are incorrect order. Equation 5.1 holds.
- Q3 - Mean distance between categories is at 0.8271

Cosine Similarity algorithm

- Q1 - Yes, while categories one and two are close, they form clear enough clusters.
- Q2 - Yes, categories do not overlap and are incorrect order. Equation 5.1 holds.
- Q3 - Mean distance between categories is at 0.4057.

Jaccard Index algorithm

- Q1 - Yes. The categories are identifiable.
- Q2 - Yes, categories do not overlap with each other and are incorrect order. Equation 5.1 holds.

- Q3 - Mean distance between categories is at 0.0658.

Jaro-Winkler distance algorithm

- Q1 - Yes. The categories are identifiable.
- Q2 - Yes, categories do not overlap and are incorrect order. Equation 5.1 holds.
- Q3 - Mean distance between categories is at 0.1991.

Levenshtein algorithm

- Q1 - Yes. The categories are identifiable.
- Q2 - Yes, categories do not overlap and are incorrect order. Equation 5.1 holds.
- Q3 - Mean distance between categories is at 0.5563.

7.3 Approach 3: XML logs from Pytest framework approach

In this section, the results of Approach 3 are presented.

7.3.1 Data set 1 results

Results of applying string similarity algorithms on the data set 1 with Approach 3 are depicted in the Figure 7.4.

Sequence Matcher algorithm

- Q1 - No. Some instances of category 3 comparison (between fail_diff_error_1.xml" and fail_same_error.XML files) have a higher similarity score than category 2 scores. Equation 5.2 does not hold.
- Q2 - Mean range between categories one and three is 0.51.

Cosine Similarity algorithm

- Q1 - Yes. All groups are in the correct order. Equation 5.2 holds.
- Q2 - Mean range between categories one and three is 0.24.

Jaccard Index algorithm

- Q1 - No, category 3 overtakes category 2 in all cases. Equation 5.2 does not hold.
- Q2 - Mean range between categories one and three is 0.133.

Jaro-Winkler distance algorithm

- Q1 - No. Some instances of category 3 have a higher similarity score than category two instances. Equation 5.2 does not hold.
- Q2 - Mean range between categories one and three is 0.13.

Levenshtein algorithm

- Q1 - No. Category three instances have a higher similarity score than category two instances. Equation 5.2 does not hold.
- Q2 - Mean range between categories one and three similarity

7.3.2 Data set 2 results

Results of applying string similarity algorithms on data set 1 with Approach 3 are depicted in Figure 7.5.

Sequence Matcher algorithm

- Q1 - Yes, categories are in the correct and equation 5.2 holds.
- Q2 - Mean range between categories one and three is 0.48.

Cosine Similarity algorithm

- Q1 - Yes. Equation 5.2 holds.
- Q2 - Mean range between categories one and three is 0.31.

Jaccard Index algorithm

- Q1 - No. Some instances of category three have the same similarity score as category two instances. Equation 5.2 does not hold.
- Q2 - Mean range between categories one and three is 0.12.

Jaro-Winkler distance algorithm

- Q1 - Yes. The categories' similarity score is in order, and equation 5.2 holds.
- Q2 - Mean range between categories one and three is 0.145.

Levenshtein algorithm

- Q1 - Yes. Equation 5.2 holds.
- Q2 - Mean range between categories one and three is 0.295.

7.3.3 Data set 3 results

Results of applying string similarity algorithms on the data set 1 with Approach 3 are depicted in the Figure 7.6.

Sequence Matcher algorithm

- Q1 - Yes. Equation 5.2 holds.
- Q2 - Mean range between categories one and three is 0.876.

Cosine Similarity algorithm

- Q1 - Yes. Equation 5.2 holds.
- Q2 - Mean range between categories one and three is 0.52.

Jaccard Index algorithm

- Q1 - Yes. Equation 5.2 holds.
- Q2 - Mean range between categories one and three is 0.145.

Jaro-Winkler distance algorithm

- Q1 - Yes. Equation 5.2 holds.
- Q2 - Mean range between categories one and three is 0.274.

Levenshtein algorithm

- Q1 - Yes. Equation 5.2 holds.
- Q2 - Mean range between categories one and three is 0.652.

Figure 7.1: Approach 2. Data set 1. Algorithms from top to bottom: Sequence Matcher, cosine similarity, Jaccard Index, Jaro-Winkler distance, Levenshtein ratio.

Figure 7.2: Approach 2. Data set 2. Algorithms from top to bottom: Sequence Matcher, cosine similarity, Jaccard Index, Jaro-Winkler distance, Levenshtein ratio

Figure 7.3: Approach 2. Data set 3. Algorithms from top to bottom: Sequence Matcher, cosine similarity, Jaccard Index, Jaro-Winkler distance, Levenshtein ratio.

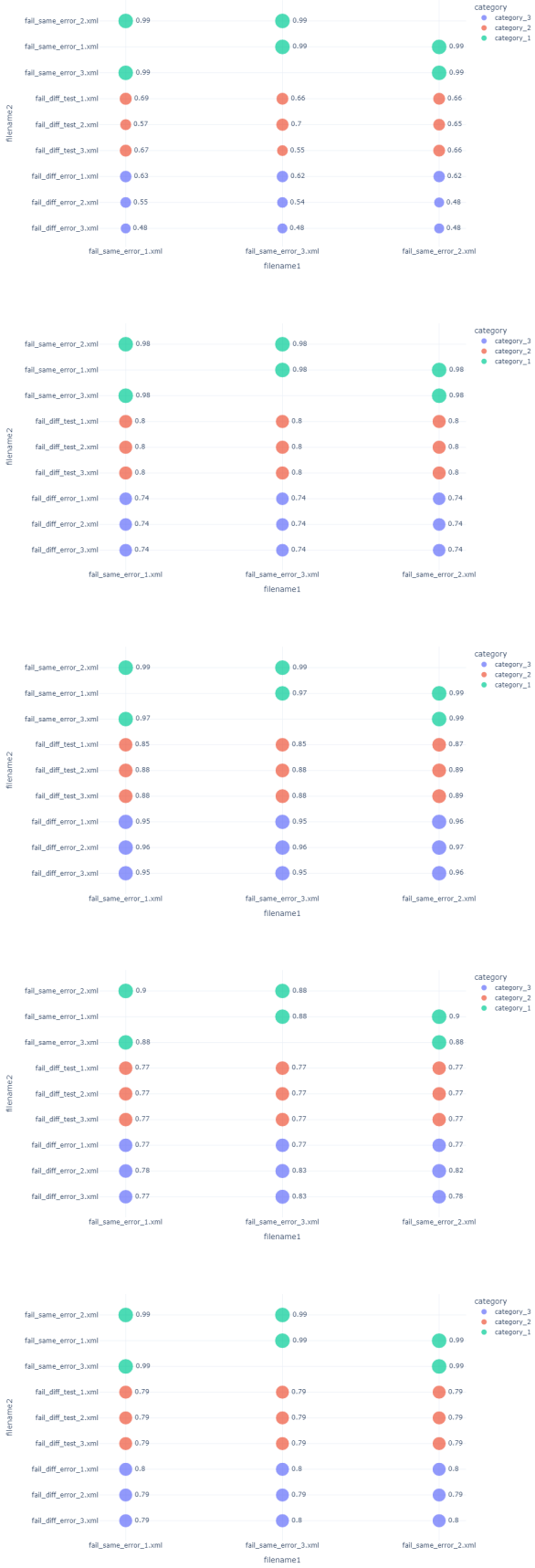

Figure 7.4: Approach 3. Data set 1. Algorithms from top to bottom: Sequence Matcher, cosine similarity, Jaccard Index, Jaro-Winkler distance, Levenshtein ratio.

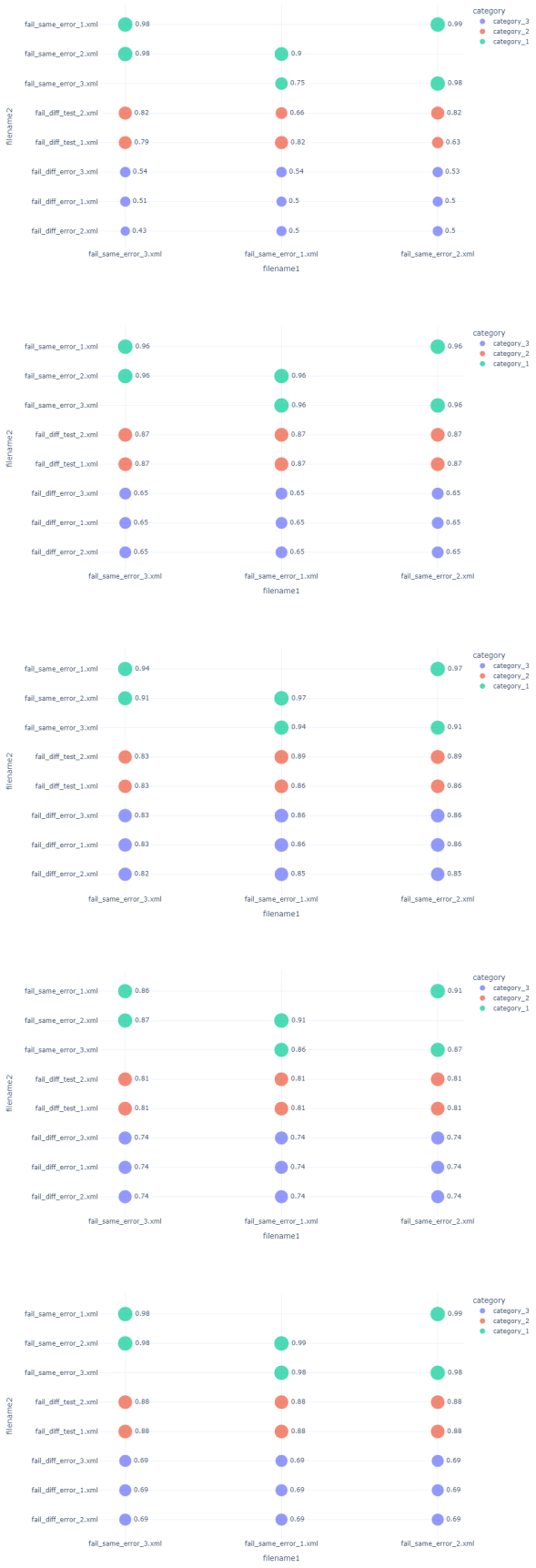

Figure 7.5: Approach 3. Data set 2. Algorithms from top to bottom: Sequence Matcher, cosine similarity, Jaccard Index, Jaro-Winkler distance, Levenshtein ratio.

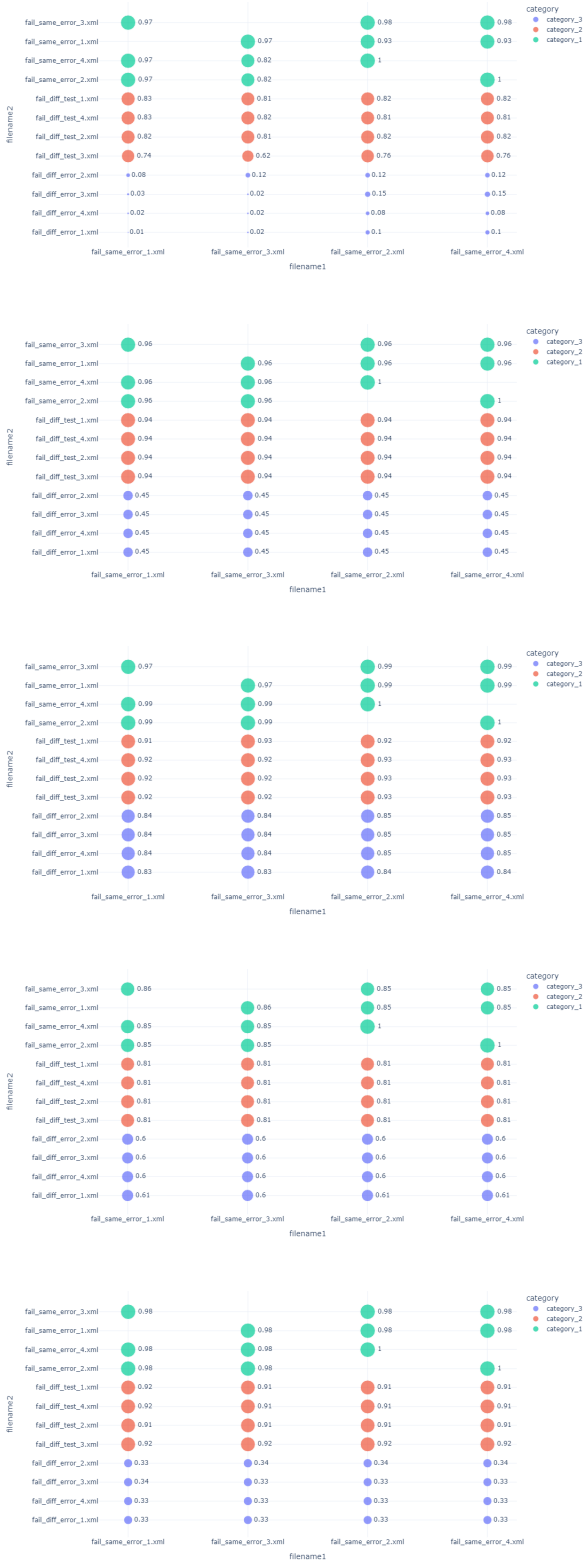

Figure 7.6: Approach 3. Data set 3. Algorithms from top to bottom: Sequence Matcher, cosine similarity, Jaccard Index, Jaro-Winkler distance, Levenshtein ratio.

8. Results from applying the TFA to an open-source project

This section contains the results of deploying the tool into an open-source project. The validation phase relied heavily on manual labelling of the log files in order to measure success with the three approaches. Unlike with the validation data sets, labelling is impossible with the real log data seen in the open-source project. Because of the lack of labels, results from the open-source projects are evaluated based on the estimated time saved by the software engineers and their feedback provided during the process.

The open-source project in question is the Robot Framework Browser library [19]. TFA was integrated into the Robot Frameworks Browser Libraries CI pipeline. After each commit, automated tests are run, and after the tests, TFA is executed. Browser library produces 200 to 300 log files per run. TFA analyses each failure in the previous run tests and presents the results as depicted in Figure 6.2. The results from an exemplary run can be found in the GitHub Action section of the project under the "Failure Analysis" drop-down menu [12]. TFA does not redact any information, nor does the output suffer in quality if compared to the information available if a software engineer would analyse the logs by hand. TFA produces clusters of similarly found failures. From here, a software engineer uses the information presented to evaluate which failures fail because of the potentially same root cause. According to the software engineers, these clusters have great value as they save much time in the daily tasks for a software engineer. For example, a task of downloading, opening and finding a failure in one of the logs takes 5 minutes for a software engineer. The software engineer never evaluates 200 files from one run but randomly picks 10 log files. A software engineer might evaluate all the then-selected files or select only a few depending on the information.

Table 8.1 shows the results of the time needed to analyse the different amounts of log files. Column "# of log files" denotes the number of files to analyse. The "By hand" column tells the time needed for a software engineer to download, open and analyse the failure by hand. The "TFA" column tells the time it takes for a TFA to analyse, score and present the output to a software engineer. "% change" is the percentage change

Table 8.1: Results on how much time it takes to analyse the different amount of logs by hand versus by TFA.

# of log files	**By hand**	**TFA**	**% change**
10	50 min	<10 sec	99.5%
267	22.5 hours	45 sec	99.96%

between values in the "By hand" and "TFA" columns for the selected amount of log files.

9. Discussion

This chapter discusses the approaches taken in this master's thesis and presents thoughts on them. Using the results from chapters "Quantified results of three approaches" and "Results from applying the TFA to an open-source project", this chapter contains answers to the research questions presented in the "Introduction" chapter. In addition, some findings and topics still left unmentioned in the previous chapter of "Results from applying the TFA to an open-source project" are highlighted here regarding the topics of integration of TFA into the Browser library project.

Tables 9.1 and 9.2 summarise the results described in the previous section. Tables can be read in the following manner:

- The X-axis denotes all five different algorithms.
- The left side denotes which questions the cell and row belong to.
- The right side denotes which data set the cell and row belong to

With the first approach, log parsing algorithms work great on simple data test logs that tend to repeat similar types of patterns in themselves. With the increase of complexity in the test logs, log parsing algorithms show their weakness. The log parsing algorithms do not appropriately extract event templates that are clear for a human. As much as 10% of the event templates contained wrong test names. Log event template extraction gets even more challenging with introducing special characters.

Clear logging standards would improve the parsing accuracy of the log parsing algorithms. Software companies can enforce strict logging practices, reducing the logs' complexity. Unfortunately, enforcing logging practices is not common among software companies.

Another shortcoming of the first approach is that validating the efficiency of the log parsing algorithms on custom log data is difficult. Log parsing algorithms lack built-in methods of validation. The lack of such methods leads to the implementation of custom validation metrics that usually require manual labelling of the test log data. Labelling can be difficult or impossible in some cases. Labelling requires domain knowledge, and taking a good sample of logs can be challenging. Labelling also falls

Table 9.1: Summary of results for Approach 2

	SM ratio	Cosine	Jaccard	Jaro-Winkler	Levenshtein	
Q1	No	**Yes**	No	**Yes**	**Yes**	DS1
Q2	No	No	No	No	No	DS1
Q3	0.5272	0.5260	0.0807	0.1433	0.3606	DS1
Q1	**Yes**	**Yes**	No	No	**Yes**	DS2
Q2	**Yes**	**Yes**	No	No	**Yes**	DS2
Q3	0.5772	0.5260	0.083	0.1051	0.2014	DS2
Q1	No	**Yes**	**Yes**	**Yes**	**Yes**	DS3
Q2	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	DS3
Q3	0.8271	0.4057	0.0658	0.1991	0.5563	DS3

Table 9.2: Summary of results for Approach 3

	SM ratio	Cosine	Jaccard	Jaro-Winkler	Levenshtein	
Q1	No	No	No	No	No	DS1
Q2	0.51	0.24	0.133	0.13	0.2	DS1
Q1	**Yes**	**Yes**	No	**Yes**	**Yes**	DS2
Q2	0.48	0.31	0.12	0.145	0.295	DS2
Q1	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	DS3
Q2	0.87	0.52	0.14	0.27	0.65	DS3

short when the system in development gets new updates. New updates to the software lead to system logs and test logs evolving, requiring a new round of log labelling.

Because of the reasons mentioned earlier, it was decided not to pursue Approach 1. Instead, the focus shifted toward Approach 2.

The second approach showed promise. Three out of five algorithms formed three distinct groups in the first-tested data set that contained very similar failures, with the equation 5.1 not holding for any of the algorithms. The best performing algorithms were sequence matcher, cosine similarity and Levenshtein ratio. All three of them had the most considerable mean distance between the groups. Unfortunately, only Levenshtein and cosine answered yes to the first question.

The second data had failures that differed significantly from each other. With clear enough failures in the second data set, categories align correctly, and thus equation 5.1 holds. Similarly to the first data set, the best performing algorithms were the cosine similarity and the Levenshtein index, which displayed a significant average distance between the groups.

The third data set had more complex failures between categories two and three. The expectation is that the distance between categories one and two should be smaller and that the distance between categories two and three should be considerable. As per expectation, four out of the five algorithms showed precisely this. Categories one and two are very close to each other, and category three was a substantial distance away from both of them. As with the first and second data sets, Levenshtein and cosine similarity algorithms were the best-performing ones. They had clear groups displayed and showed an excellent mean distance between them. Additionally, all algorithms answered "Yes" to the second question. The second approach proves that it can be reliable to cluster complex failures instead of just simple ones in the second data set.

The best performing algorithm across all three data sets for Approach 2 was cosine similarity. While the Levenshtein distance came very close to producing the same results, it showed less distance between clusters in some cases.

During the review phase of Approach 2, it was noticed that not all the individual test cases in the Pytest testing framework start with the "setup" keyword and end with the "teardown". Software engineers can sometimes construct test suites to run multiple test cases simultaneously. This can have an appearance of multiple consecutive "setup" keywords appearing in the test logs without the "teardown" keyword in between. Test suite construction is very dependent on the software engineer working on it and is very contextual to the needs of the project or software under testing. The second approach heavily relies on the "setup" and "teardown" keywords to form an accurate structure of the tests and break down the test suites into individual test cases during the log preprocessing step for comparison and scoring. The mix-up caused by consecutive

keywords in the test logs can lead to poor parsing, false test case extraction, and inaccurate data.

Improving the preprocessing step by developing a better way to parse and extract individual test cases from the test suites relies on patterns. Identifying patterns by which the keywords appear seems impossible, and even if it can be done, it would only apply to this test suite. Changing the test suite by adding a new test case or removing such would require another look at the pattern and alterations to the parsing.

The reasons described above were the biggest shortcoming of the second approach. While non-existent for the data sets presented in this work, these problems cannot be ignored as they can lead to unreliable results. Because of this, Approach 2 was dropped at this point, and the focus shifted to Approach 3.

The third approach relied on XML files, another form of test logs and is a product of the Pytest testing framework. The approach aimed to reduce the need for an additional layer of parsing that exposes the tool to complications similar to the ones encountered in the second approach.

The first data set behaved in the same manner as the second approach. Across all the algorithms, categories two and three were in the wrong order. As with the second approach, too similar of a failure can confuse the tool also with this approach. The best performing algorithms were cosine similarity and Levenshtein.

With the second data set, only the Jaccard index did not place categories in the correct order. This only happened because the similarity score of category three test logs scored the same as in category two. The best performing algorithms were sequence matcher ratio, cosine similarity and Levenshtein. All three categories had a good range between categories, meaning that there is a reasonable distance between failures, allowing some room for error between the failures.

The third data set had complex failures. All the algorithms scored the categories in the correct order, meaning that Equation 5.2 holds for all the algorithms. Sequence Matcher showed the biggest range among all the algorithms. Due to this, Sequence Matcher performed the best.

While reviewing the results, no apparent shortcomings were observed with Approach 3 on the data sets available for validation; additionally, the results were satisfactory. Both Levensthein ratio and cosine similarity performed well across the three data sets. However, cosine similarity had a higher range between the similarity scores of failure; cosine similarity was picked over as the best-performing algorithm for Approach 3.

Because of the significant shortcomings of Approach 1 and 2, it was decided to implement Approach 3 into the open-sourced project with cosine similarity as the main algorithm for scoring the failures.

Results of implementing TFA into a CI pipeline of an open-source project were promising. During the evaluation phase of TFA, TFAs execution time has improved drastically. Initially, TFA took up to 30 minutes to analyse 267 test logs. From there, improvements have been made to the tool; these improvements included code optimisation, removing unnecessary calculations, changing the approach from permutations to combinations and introducing a threshold value. The threshold value filters out failures below a particular value, meaning the filtered-out failures were not similar. These improvements reduced execution time from 30 minutes to one minute. Additionally, support for another testing framework - Robot Framework, was added [22].

In the current state of implementation, the tool does much more than a software engineer would do to analyse the failures. First, the tool evaluates all the files quickly compared to the time it takes to do the same task manually. Secondly, it already clusters the failures, which would require additional time for a software engineer.

The results described above are hard to compare with each other as a case where a software engineer analyses all 267 log files would never happen. Instead, comparing the amount of time it would take to analyse ten log files by hand versus TFA analysing all the files available is more reasonable. Even with such a comparison, a change of 99.96% in time spent on analysing failures is significant. It is worth mentioning that while TFAs run time is under a minute, a software engineer needs to analyse the output produced by the TFA. While an experienced software engineer familiar with the project can conclude the root cause in two to three minutes, someone with less experience with the project might require more time. This time, of course, adds to the time needed to conclude what to fix next.

Unfortunately, any additional metrics for evaluating TFA were not available. Metrics like "how many failures got fixed after analysis of TFA" and "how many similar failures get fixed as a byproduct of fixing the original failure" would have helped analyse the tool.

Currently, there is an issue where some failures might be considered duplicates even with a threshold value. Failures that the tool considers different but are the same are caused by, for example, a timestamp or a transaction id. The problem can be addressed using log parsing algorithms tried out in Approach 1 by extracting the parameters such as a date and time stamp before comparison. The approach of extracting the parameters of the failure message before applying the string similarity algorithm could be highly successful as the log messages are not complex.

Nonetheless, after consulting with the software engineers at WithSecure, TFA has been deemed valuable and time-saving, and there are plans to implement it in in-house projects of WithSecure.

9.0.1 Answer to RQ1 - To what extent is the concept, in the form of the software tool, applicable to address the log analysing challenges?

The concept implemented in the form of the software tool is applicable to address the log analysing challenges to a great extent. The concept of utilising string similarity algorithms on the failures found within the test logs to produce clusters of failures that guide the engineers into selecting impactful failures for fixing - works.

Out of the three approaches, the best approach was Approach 3, even though results from Approach 2 were promising. The results chapter showed that the validation phase was most successful using the cosine similarity algorithm across all three data sets. The tool can differentiate between regular failures in the logs and complex stack traces. While the tool had trouble categorising the almost identical failures, it still provided clear value by clustering them.

9.0.2 Answer to RQ2 - How does the concept, in the form of the software tool, improve the performance of log analyses?

The implemented concept in the form of the software tool improves the performance of log analyses significantly. With the integration of TFA into the Robot Frameworks Browser library CI pipeline and the results reported in the previous chapter, it is clear that the concept saves time in a software engineer's daily work. The current implementation of the TFA can save up to 99.5% of time used for failure analysis and possibly more if the amount of logs increases.

10. Conclusions

This work comes up with the concept of grouping failures within test logs using string similarity algorithms. The developed concept tries to combat the problem of the growing amounts of test logs produced by CI pipelines that run automated tests. The concept has been validated by implementing the Test Failure Analysis tool. A pip package that is open-sourced and available for further improvement. The tool assists software developers who currently have to analyse the failures manually. TFA provides a fast and reliable way to find a similar group of failures with the test log produced from the tests run in the CI pipeline. The groups aim to provide insight to software engineers by highlighting failures which fail similarly by scoring the failure stack traces with the cosine similarity algorithm. With this information, a software engineer can prioritise fixing the failures that have more impact on software quality.

During the work, analyses have been performed on related work. Related work topics include log clustering, log analysis, test log analysis and data transformation. These topics are essential for this master's thesis because they are closely tied together. Almost all the log analysis approaches try to answer the question of data transformation. These approaches that turn unstructured log formats into structured data formats vary. Some use mathematical models, machine learning, novel log parsing algorithms, and an enhanced version of existing algorithms. It has been deemed that none of the reviewed research papers tries to tackle test log analysis in the same way as the TFA.

TFA has the implementation of three different approaches. Each of the approaches tried to solve data transformation uniquely. Each approach was validated on the three different data sets with unique traits. Results from each approach were evaluated based on the three questions involving the categories into which the failures should fall.

Results have shown that while novel log parsing algorithms such as Drain and Spell are exciting and can have high parsing accuracy on simple log messages, they struggle with complex log messages (Approach 1). Manual parsing of log files provided by the Pytest framework proved challenging and required custom parsing for each test suit setup and was abandoned because of that reason (Approach 2). The best approach was to utilise .xml files provided by the Pytest framework (Approach 3).

In addition, the results have shown that the best-performing algorithm was cosine similarity. Cosine similarity has scored the failures within the validation data set in the order specified while having the highest range between clusters and failures. Unfortunately, like all other algorithms, cosine similarity has failed to differentiate identical failures. Nonetheless, the clusters created out of scoring the failures were helpful even with the wrong order of the categories.

Based on the results best algorithm and approach have been picked. With Approach 3 and cosine similarity, TFA was deployed to the Robot frameworks Browser library projects CI pipeline. Results from this experiment have not reached the level of quality as those from the validation phase. Nonetheless, the software engineers at WithSecure have highly appreciated the tool. TFA has boosted performance by almost 99.5% of the time that went into analysing failures by hand. Further development of the TFA is to introduce the log parsing algorithms trialled in Approach 1 (Drain and Spell) to reduce the issues presented. WithSecure has reported plans to implement TFA into in-house projects for further use. TFA has also been released as an open-source project allowing researchers and developers to augment the tool to their needs further.

Research questions are answered based on the results from the validation phase and the deployment of the TFA into the open-source project. Based on the results, the answer to both questions is positive and even very promising. Both the concept and the tool implementing the concept are not only applicable to address the log analysing challenges but significantly improve the performance of test log analyses. That is expected to save time from the daily work of software engineers and therefore develop more efficient software engineering processes to improve the data analytics performance in general.

Bibliography

[1] Alibaba.com: Manufacturers, Suppliers, Exporters & Importers from the world's largest online B2B marketplace. https://www.alibaba.com/, Accessed on 12th March 2022.

[2] AliMail Personal Edition. https://mail.aliyun.com/alimail/auth/login, Accessed on 12th March 2022.

[3] Apache Hadoop. https://hadoop.apache.org/, Accessed on 27th April 2022.

[4] Automated software testing for continuous delivery. https://www.atlassian.com/continuous-delivery/software-testing/automated-testing, Accessed on 14th September 2022.

[5] Big Data and Business Analytics Market Size, Share | 2030. https://www.alliedmarketresearch.com/big-data-and-business-analytics-market#:~:text=The%20global%20big%20data%20and,13.5%25%20from%202021%20to%202030, Accessed on 19th June 2022.

[6] Business Software And Services Market Report, 2030. https://www.grandviewresearch.com/industry-analysis/business-software-services-market, Accessed on 19th June 2022.

[7] Compound Annual Growth Rate (CAGR) Formula and Calculation. https://www.investopedia.com/terms/c/cagr.asp, Accessed on 25th September 2022.

[8] Continuous Integration. https://martinfowler.com/articles/continuousIntegration.html, Accessed on 27th February 2022.

[9] difflib Helpers for computing deltas Python 3.10.4 documentation. https://docs.python.org/3/library/difflib.html, Accessed on 14th May 2022.

[10] F-Secure/failures-analysis: Groupping automatically similar failures in the CI/CD pipeline. https://github.com/F-Secure/failures-analysis, Accessed on 2nd June 2022.

[11] failures-analysis PyPI. https://pypi.org/project/failures-analysis/, Accessed on 2nd June 2022.

[12] Failures analysis · MarketSquare/robotframework-browser@60890f1. https://github.com/MarketSquare/robotframework-browser/runs/6932823910?check_suite_focus=true, Accessed on 17th June 2022.

[13] Features GitHub Actions. https://github.com/features/actions/, Accessed on 5th September 2022.

[14] Home - Ivves. https://ivves.eu/, Accessed on 6th June 2022.

[15] Home | WithSecureâ¢. https://www.withsecure.com/en/home, Accessed on 9th May 2022.

[16] itertools Functions creating iterators for efficient looping â Python 3.10.3 documentation. https://docs.python.org/3/library/itertools.html, Accessed on 24th March 2022.

[17] Jenkins. https://www.jenkins.io/, Accessed on 5th September 2022.

[18] Logparserâs Documentation logparser 0.1 documentation. https://logparser.readthedocs.io/en/latest/README.html, Accessed on 13th March 2022.

[19] Marketsquare/robotframeworkbrowser: Robot framework browser library powered by playwright. https://github.com/MarketSquare/robotframework-browser, Accessed on 6th June 2020.

[20] pytest: helps you write better programs pytest documentation. https://docs.pytest.org/en/7.1.x/, Accessed on 12th May 2022.

[21] Python Package Health Analysis. https://snyk.io/advisor/python/pytest#popularity, Accessed on 28th March 2022.

[22] Robot Framework. https://robotframework.org/, Accessed on 17th June 2022.

[23] SLCT logparser 0.1 documentation. https://logparser.readthedocs.io/en/latest/tools/SLCT.html, Accessed on 14th June 2022.

[24] Test Automation Frameworks | SmartBear. https://smartbear.com/learn/automated-testing/test-automation-frameworks/, Accessed on 14th September 2022.

[25] Test log overview - IBM Documentation. https://www.ibm.com/docs/en/rstfsq/10.0.0.0?topic=SSNKWF_10.0.0/com.ibm.rational.test.lt.common.doc/topics/ttestlogoverview.html/, Accessed on 14th September 2022.

[26] Tools for Continuous Integration at Google Scale - YouTube. https://www.youtube.com/watch?v=KH2_sB1A6lA, Accessed on 12th March 2022.

[27] Understanding Log Analytics at Scale. https://learning.oreilly.com/library/view/understanding-log-analytics/9781492076254/, Accessed on 19th June 2022.

[28] What is Continuous Integration? Amazon Web Services. https://aws.amazon.com/devops/continuous-integration/, Accessed on 14th May 2022.

[29] What is Log Analytics | Amazon Web Services. https://aws.amazon.com/log-analytics/#:~:text=Log%20analytics%20involves%20searching%2C%20analyzing,of%20rapidly%20proliferating%20machine%20data, Accessed on 19th June 2022.

[30] F-Secure Q4 Financial Report 2020. 2020. https://www.f-secure.com/content/dam/f-secure/en/investors/materials/interim-reports/2020/f-secure-interim-report-q4-2020.pdf, Accessed on 25th September 2022.

[31] A. Amar and P. C. Rigby. Mining Historical Test Logs to Predict Bugs and Localize Faults in the Test Logs. In *Proceedings - International Conference on Software Engineering*, volume 2019-May, pages 140–151. IEEE Computer Society, 5 2019.

[32] W. Aoudi, M. Iturbe, and M. Almgren. Truth will out: Departure-based process-level detection of stealthy attacks on control systems. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 817–831. Association for Computing Machinery, 10 2018.

[33] M. Bilenko and R. J. Mooney. Adaptive duplicate detection using learnable string similarity measures. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '03*, page 39, New York, New York, USA, 2003. ACM Press.

[34] B. Chen, J. Song, P. Xu, X. Hu, and Z. M. Jiang. An automated approach to estimating code coverage measures via execution logs. In *ASE 2018 - Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 305–316. Association for Computing Machinery, Inc, 9 2018.

[35] W. W. Cohen, P. Ravikumar, and S. Fienberg. A Comparison of String Metrics for Matching Names and Records. 2003.

[36] M. Du and F. Li. Spell: Online Streaming Parsing of Large Unstructured System Logs. *IEEE Transactions on Knowledge and Data Engineering*, 31(11):2213–2227, 11 2019.

[37] M. Du, F. Li, G. Zheng, and V. Srikumar. DeepLog: Anomaly detection and diagnosis from system logs through deep learning. *Proceedings of the ACM Conference on Computer and Communications Security*, pages 1285–1298, 2017.

[38] S. Elbaum, G. Rothermel, and J. Penix. Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 16-21-November-2014, pages 235–245. Association for Computing Machinery, 11 2014.

[39] C. Feng, V. R. Palleti, A. Mathur, and D. Chana. A Systematic Framework to Generate Invariants for Anomaly Detection in Industrial Control Systems. Internet Society, 3 2019.

[40] Q. Fu, J.-G. Lou, Y. Wang, and J. Li. Execution Anomaly Detection in Distributed Systems through Unstructured Log Analysis. In *2009 Ninth IEEE International Conference on Data Mining*, pages 149–158. IEEE, 12 2009.

[41] Q. Fu, J. Zhu, W. Hu, J. G. Lou, R. Ding, Q. Lin, D. Zhang, and T. Xie. Where do developers log? An empirical study on logging practices in industry. In *36th International Conference on Software Engineering, ICSE Companion 2014 - Proceedings*, pages 24–33. Association for Computing Machinery, 2014.

[42] H. Hamooni, B. Debnath, J. Xu, H. Zhang, G. Jiang, and A. Mueen. LogMine: Fast pattern recognition for log analytics. *International Conference on Information and Knowledge Management, Proceedings*, 24-28-Octo:1573–1582, 2016.

[43] J. Han, M. Kamber, and J. Pei. Getting to Know Your Data. In *Data Mining*, pages 39–82. Elsevier, 2012.

[44] S. Han, Q. Wu, H. Zhang, B. Qin, J. Hu, X. Shi, L. Liu, and X. Yin. Log-Based Anomaly Detection With Robust Feature Extraction and Online Learning. *IEEE Transactions on Information Forensics and Security*, 16:2300–2311, 2021.

[45] P. He, J. Zhu, Z. Zheng, and M. R. Lyu. Drain: An Online Log Parsing Approach with Fixed Depth Tree. In *2017 IEEE International Conference on Web Services (ICWS)*, pages 33–40. IEEE, 6 2017.

[46] A. Hevner and S. Chatterjee. Design Research in Information Systems. 22, 2010.

[47] A. R. Hevner, S. T. March, J. Park, and S. Ram. DESIGN SCIENCE IN INFORMATION SYSTEMS RESEARCH 1. Technical Report 1, 2004.

[48] IEEE Communications Society and Institute of Electrical and Electronics Engineers. *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications.*

[49] IEEE Computer Society. and IEEE Computer Society. Technical Council on Software Engineering. *Log-based testing.* IEEE, 2012.

[50] A. Islam and D. Inkpen. Semantic Text Similarity Using Corpus-Based Word Similarity and String Similarity. *ACM Transactions on Knowledge Discovery from Data*, 2(2):1–25, 7 2008.

[51] P. Jaccard. THE DISTRIBUTION OF THE FLORA IN THE ALPINE ZONE.1. *New Phytologist*, 11(2):37–50, 2 1912.

[52] A. Juvonen, T. Sipola, and T. Hämäläinen. Online anomaly detection using dimensionality reduction techniques for HTTP log analysis. *Computer Networks*, 91:46–56, 11 2015.

[53] H. Kahlouche, C. Viho, and M. Zendri. An industrial experiment in automatic generation of executable test suites for a cache coherency protocol. In *Testing of Communicating Systems*, pages 211–226. Springer US, Boston, MA, 1998.

[54] S. Kobayashi, K. Otomo, K. Fukuda, and H. Esaki. Mining Causality of Network Events in Log Data. *IEEE Transactions on Network and Service Management*, 15(1):53–67, 2018.

[55] E. Kowalczyk, K. Nair, Z. Gao, L. Silberstein, T. Long, and A. Memon. Modeling and ranking flaky tests at apple. *Proceedings - International Conference on Software Engineering*, pages 110–119, 2020.

[56] V. T. Kramar, J. K. Nurminen, and T. Aalto. Grouping Pytest Logs with the Same Root Cause Using String Similarity Algorithms for Easier Debugging. Technical report, 2022.

[57] A. R. Lahitani, A. E. Permanasari, and N. A. Setiawan. Cosine similarity to determine similarity measure: Study case in online essay assessment. In *Proceedings of 2016 4th International Conference on Cyber and IT Service Management, CITSM 2016*. Institute of Electrical and Electronics Engineers Inc., 9 2016.

[58] V. I. Levenshtein and others. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710, 1966.

[59] T. Li, Y. Jiang, C. Zeng, B. Xia, Z. Liu, W. Zhou, X. Zhu, W. Wang, L. Zhang, J. Wu, L. Xue, and D. Bao. FLAP: An end-to-end event log analysis platform for system management. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, volume Part F129685, pages 1547–1556. Association for Computing Machinery, 8 2017.

[60] M. H. Lim, J. G. Lou, H. Zhang, Q. Fu, A. B. J. Teoh, Q. Lin, R. Ding, and D. Zhang. Identifying Recurrent and Unknown Performance Issues. *Proceedings - IEEE International Conference on Data Mining, ICDM*, 2015-Janua(January):320–329, 2014.

[61] Q. Lin, H. Zhang, J. G. Lou, Y. Zhang, and X. Chen. Log clustering based problem identification for online service systems. *Proceedings - International Conference on Software Engineering*, pages 102–111, 2016.

[62] A. Makanju, A. Nur Zincir-Heywood, and E. E. Milios. Clustering Event Logs Using Iterative Partitioning. *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '09*, 2009.

[63] A. Memon, Zebao Gao, Bao Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco. Taming Google-scale continuous testing. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 233–242. IEEE, 5 2017.

[64] W. Meng, Y. Liu, Y. Zhu, S. Zhang, D. Pei, Y. Liu, Y. Chen, R. Zhang, S. Tao, P. Sun, and R. Zhou. LogAnomaly: Unsupervised Detection of Sequential and Quantitative Anomalies in Unstructured Logs. Technical report, 2019.

[65] S. Messaoudi, A. Panichella, D. Bianculli, L. Briand, and R. Sasnauskas. A search-based approach for accurate identification of log message formats. In *Proceedings - International Conference on Software Engineering*, pages 167–177. IEEE Computer Society, 5 2018.

[66] S. Messaoudi, D. Shin, A. Panichella, D. Bianculli, and L. C. Briand. Log-based slicing for system-level test cases. In *ISSTA 2021 - Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 517–528. Association for Computing Machinery, Inc, 7 2021.

[67] H. Mi, H. Wang, Y. Zhou, M. R. T. Lyu, and H. Cai. Toward fine-grained, unsupervised, scalable performance diagnosis for production cloud computing systems. *IEEE Transactions on Parallel and Distributed Systems*, 24(6):1245–1255, 2013.

[68] J. Micco. The State of Continuous Integration Testing @Google. Technical report.

[69] M. Mizutani. Incremental Mining of System Log Format. In *2013 IEEE International Conference on Services Computing*, pages 595–602. IEEE, 6 2013.

[70] M. Nagappan and M. A. Vouk. Abstracting log lines to log event types for mining software system logs. In *Proceedings - International Conference on Software Engineering*, pages 114–117, 2010.

[71] K. Nagaraj, C. Killian, and J. Neville. Structured Comparative Analysis of Systems Logs to Diagnose Performance Problems. Technical report.

[72] C. S. C. S. Peirce, N. Houser, C. J. W. Kloesel, and Peirce Edition Project. The essential Peirce : selected philosophical writings. 1992.

[73] T. Reidemeister, M. Jiang, and P. A. Ward. Mining unstructured log files for recurrent fault diagnosis. In *Proceedings of the 12th IFIP/IEEE International Symposium on Integrated Network Management, IM 2011*, pages 377–384, 2011.

[74] Y. Ren, Z. Gu, Z. Wang, Z. Tian, C. Liu, H. Lu, X. Du, and M. Guizani. System Log Detection Model Based on Conformal Prediction. *Electronics*, 9(2):232, 1 2020.

[75] R. E. Rice and C. L. Borgman. The Use of Computer-Monitored Data in information Science and Communication Research. Technical report, 1983.

[76] M. Shahin, M. Ali Babar, and L. Zhu. Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices, 2017.

[77] K. Shima. Length Matters: Clustering System Log Messages using Length of Words. 11 2016.

[78] M. Soleimani, F. Campean, and D. Neagu. Integration of Hidden Markov Modelling and Bayesian Network for fault detection and prediction of complex engineered systems. *Reliability Engineering and System Safety*, 215, 11 2021.

[79] L. Tang, T. Li, and C. S. Perng. LogSig: Generating system events from raw textual logs. *International Conference on Information and Knowledge Management, Proceedings*, (October 2011):785–794, 2011.

[80] A. Tosun, O. Turkgulu, D. Razon, H. Y. Aydemir, and A. Gureller. Predicting defects using test execution logs in an industrial setting. In *Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering Companion, ICSE-C 2017*, pages 294–296. Institute of Electrical and Electronics Engineers Inc., 6 2017.

[81] M. B. Uspenskij. Log mining and knowledgebased models in data storage systems diagnostics. *E3S Web of Conferences*, 140:03006, 12 2019.

[82] R. Vaarandi. A data clustering algorithm for mining patterns from event logs. In *Proceedings of the 3rd IEEE Workshop on IP Operations and Management, IPOM 2003*, pages 119–126. Institute of Electrical and Electronics Engineers Inc., 2003.

[83] R. Vaarandi and M. Pihelgas. Using security logs for collecting and reporting technical security metrics. In *Proceedings - IEEE Military Communications Conference MILCOM*, pages 294–299. Institute of Electrical and Electronics Engineers Inc., 11 2014.

[84] R. Vaarandi and M. Pihelgas. LogCluster - A data clustering and pattern mining algorithm for event logs. In *Proceedings of the 11th International Conference on Network and Service Management, CNSM 2015*, pages 1–7. Institute of Electrical and Electronics Engineers Inc., 12 2015.

[85] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan. Online system problem detection by mining patterns of console logs. In *Proceedings - IEEE International Conference on Data Mining, ICDM*, pages 588–597. IEEE, 12 2009.

[86] B. Zhang, H. Zhang, P. Moscato, and A. Zhang. Anomaly Detection via Mining Numerical Workflow Relations from Logs. In *Proceedings of the IEEE Symposium on Reliable Distributed Systems*, volume 2020-September, pages 195–204. IEEE Computer Society, 9 2020.

[87] M. J. Zhen, A. E. Hassan, P. Flora, and G. Hamann. Abstracting execution logs to execution events for enterprise applications. In *Proceedings - International Conference on Quality Software*, pages 181–186, 2008.

[88] J. Zhu, S. He, J. Liu, P. He, Q. Xie, Z. Zheng, and M. R. Lyu. Tools and Benchmarks for Automated Log Parsing. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 121–130. IEEE, 5 2019.