UNIVERSITY OF OULU

FACULTY OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

Joel Rytkönen

CLUSTER ANALYSIS ON RADIO PRODUCT INTEGRATION TESTING FAULTS

Master's Thesis
Degree Programme in Computer Science and Engineering
September 2023

Rytkönen J. (2023) Cluster Analysis on Radio Product Integration Testing Faults. University of Oulu, Degree Programme in Computer Science and Engineering, 56 p.

ABSTRACT

Nowadays, when the different software systems keep getting larger and more complex, integration testing is necessary to make sure that the different components of the system work together correctly. With the large and complicated systems the analysis of the test faults can be difficult, as there are so many components that can cause the failure. Also with the increased usage of automated tests, the faults can often be caused by test environment or test automation issues.

Testing data and logs collected during the test executions are usually the main source of information that are used for test fault analysis. With the usage of text mining, natural language processing and machine learning methods, the fault analysis process is possible to be automated using the data and logs collected from the tests, as multiple studies have shown in the recent years.

In this thesis, an exploratory data study is done on data collected from radio product integration tests done at Nokia. Cluster analysis is used to find the different fault types that can be found from each of the collected file types. Different feature extraction methods are used and evaluated in terms of how well they separate the data for fault analysis.

The study done on this thesis paves the way for automated fault analysis in the future. The introduced methods can be applied for classifying the faults and the results and findings can be used to determine what are the next steps that can be taken to enable future implementations for automated fault analysis applications.

Keywords: machine learning, text mining, natural language processing, log analysis

Rytkönen J. (2023) Radiotuotteiden integraatiotestauksen vikojen klusterianalyysi. Oulun yliopisto, Tietotekniikan tutkinto-ohjelma, 56 s.

TIIVISTELMÄ

Nykypäivänä, kun erilaiset ohjelmistojärjestelmät jatkavat kasvamista ja muuttuvat monimutkaisimmaksi, integraatiotestaus on välttämätöntä, jotta voidaan varmistua siitä, että järjestelmän eri komponentit toimivat yhdessä oikein. Suurien ja monimutkaisten järjestelmien testivikojen analysointi voi olla vaikeaa, koska järjestelmissä on mukana niin monta komponenttia, jotka voivat aiheuttaa testien epäonnistumisen. Testien automatisoinnin lisääntymisen myötä testit voivat usein epäonnistua myös johtuen testiympäristön tai testiautomaation ongelmista.

Testien aikana kerätty testidata ja testilogit ovat yleensä tärkein tiedonlähde testivikojen analyysissä. Hyödyntämällä tekstinlouhinnan, luonnollisen kielen käsittelyn sekä koneoppimisen menetelmiä, testivikojen analyysiprosessi on mahdollista automatisoida käyttämällä testien aikana kerättyä testidataa ja testilogeja, kuten monet tutkimukset ovat viime vuosina osoittaneet.

Tässä tutkielmassa tehdään eksploratiivinen tutkimus Nokian radiotuotteiden integraatiotesteistä kerätyllä datalla. Erilaiset vikatyypit, jotka voidaan löytää kustakin kerätystä tiedostotyypistä, löydetään käyttämällä klusterianalyysiä. Ominaisuusvektorien laskentaan käytetään eri menetelmiä ja näiden menetelmien kykyä erotella dataa vika-analyysin näkökulmasta arvioidaan.

Tutkielmassa tehty tutkimus avaa tietä vika-analyysien automatisoinnille tulevaisuudessa. Esitettyjä menetelmiä voidaan käyttää vikojen luokittelussa ja tuloksien perusteella voidaan määritellä, mitkä ovat seuraavia askelia, jotta vika-analyysiprosessia voidaan automatisoida tulevaisuudessa.

Avainsanat: koneoppiminen, tekstinlouhinta, luonnollisen kielen käsittely, logianalyysi

TABLE OF CONTENTS

FOREWORD

This thesis work is done at Nokia Solutions and Networks Oy. I would like to thank Nokia and my supervisor Tuomo Pylkäs for providing me the possibility of doing my thesis work at Nokia and in Tuomo's team. Special thanks go to my technical supervisor Ahmed Ibrahim and my thesis supervisor Mourad Oussalah, who both helped me during the thesis work process.

Oulu, September 5th, 2023

Joel Rytkönen

LIST OF ABBREVIATIONS AND SYMBOLS

AI	artificial intelligence
BTS	base transceiver station
CAM	cause analysis model
EKNN	exclusive k nearest neighbors
HW	hardware
IDF	inverse document frequency
KNN	k nearest neighbors
ML	machine learning
NLP	natural language processing
NLTK	natural language toolkit
PCA	principal component analysis
RFSW	radio frequency software
SLCT	simple logfile clustering tool
SVD	singular value decomposition
SW	software
TF	term frequency
TF-IDF	term frequency-inverse document frequency
UE	user equipment
VSM	vector space model

1. INTRODUCTION

One of the most important steps in software (SW) quality assurance is testing. Nowadays, when systems keep growing larger and larger in the number of different SW and hardware (HW) components or modules, integration testing is necessary to ensure that the interfaces between each module works correctly and also that the full system consisting of these smaller modules is working as it should. This is different from unit testing, as unit testing is performed at statement level on one module, while integration testing focuses on the module interfaces and interactions [1].

When the systems keep getting more complex, the analysis of the test faults, when a failure happens in the integration tests, is getting harder. There are multiple different module interfaces tested at the same time, so it can be hard to find which of the modules is causing the test to fail. It does not make it any easier that often the SW and HW components that make up the system are developed by different teams or even different companies, so the analysis of the test faults is often difficult and requires experience.

In the recent years, multiple studies have shown that the fault analysis process can be automatized using the data and logs collected from the tests by applying different text mining and natural language processing (NLP) methods with machine learning (ML) models. Automating the fault analysis process can save a lot of time and help finding the reasons for faults even if the person doing the analysis does not have lots of experience.

At Nokia, SW integration testing is done for radio product related SW. The SW being tested is complex and consists of multiple different components developed by different teams. In the integration tests, the SW is tested by installing it in a base transceiver station (BTS) and/or a radio, so also complex HW is included in the tests. The complexity of the testing setup does not end at the tested product, as the test environment consists of many different parts, such as user equipment (UE), that is usually a mobile phone, radio rotator, cabling, attenuators and control PCs. Most of the integration tests are automated, so that also brings more complexity to the tests with different reservation systems, connection requirements and configurations. All in all, the test setup is complex with lots of different components that could cause a test to fail, which can make it difficult and time consuming for the test engineers to find the reason for test failures.

Some of the testing related data and logs are being collected automatically as a part of the test automation process, but there are also some data that the test engineers are collecting manually for the fault analysis. The automatically collected data can be used to try and automate the fault analysis process, but as some of the relevant data is not yet collected automatically, the automated fault analysis might not be complete in terms of finding and characterizing all the possible faults. Even though the data has been collected automatically for some time now and there is a good amount of historical data available, the fault analysis results for the historical data have not been documented in a way that they could be connected to the data, so this will make it harder to automate the fault analysis process.

In this thesis, an exploratory study is performed on the automatically collected data from the radio product related SW integration tests. The study focuses on the faults found from the collected data. As the work is done on unlabeled data, exploratory study is made using cluster analysis to find and group the different faults. The results from this study can be used to help in automating the fault analysis process in the future. Research questions for the study are:

1. Q_1: What kind of faults can be identified from the data collected from radio product related SW integration tests, and what kind of faults cannot be found? What data is required to be collected to fill the gaps, if there are any?

2. Q_2: Which feature extraction method can separate the data collected from radio product related SW integration tests best in terms of fault types from the following used feature extraction methods: TF-IDF, Word2Vec weighted by TF-IDF, and line-IDF?

3. Q_3: How can the fault analysis process be automatized in the future and what are the next steps to get there?

The structure of the rest of paper is defined in the following way. Chapter 2 introduces the basic concept of ML and the different categories of how ML models can be trained, giving a more detailed introduction of clustering. Chapter 3 introduces the basic concepts of text mining and NLP and dives into text clustering and classification in more detail. Chapter 4 introduces how log file analysis can be automated using ML, text mining, and NLP methods based on the methods found from literature and related research work. In Chapter 5, the general pipeline for data preprocessing and clustering done in this thesis work is presented with explanations of the used methods. Chapter 6 describes the data that is used in this work and the preprocessing steps that are used for each of the used file types. Chapter 7 describes how the clustering and the cluster analysis is done. In Chapter 8, the cluster analysis results and the findings from these results are collected. In Chapter 9, the findings from the results are discussed, the research questions are answered, and future improvement and implementation ideas are presented. Finally, the work done in this thesis work is summarized in Chapter 10.

2. MACHINE LEARNING

Machine learning (ML) studies the algorithms and models that try to mimic human-like experience based learning on machines [2]. It is one of the fields of artificial intelligence (AI). The ML models use collected data to "learn", i.e., the models are trained with data, similarly as humans use their experiences and observations to learn. The main objective of ML is to build algorithms or models that can find unidentified relationships between variables from a dataset and generate meaningful outputs based on what is found or learnt from the data [2]. Because the collected data is the source of intelligence for ML models, it is clear that the amount and quality of data is essential when training the models. ML approaches can be split into four different categories based on how the model is trained: (1) supervised learning, (2) unsupervised learning, (3) semi-supervised learning, and (4) reinforcement learning.

2.1. Supervised Learning

In supervised learning a typical goal is to create a ML model that can predict an output given a set of inputs. The word supervised comes from the notion that a "supervisor" has to instruct the model by labeling the training data [3], i.e., the training dataset must include inputs and the correct outputs. The outputs that the model is trying to predict can be either categorical (class labels) or continuous values. As an example of an categorical output, the model could be trying to predict if an image has a cat or dog in it, and as an example of a continuous output, the model could be trying to predict a stock value. In the case of categorical outputs, the prediction task is called classification, and in the case of continuous outputs the prediction task is called regression [4]. The major benefit of supervised learning is that the labeled training data gives definite criteria for optimizing the models [3], but often the process of collecting labeled data is laborious and requires manual work to label the data.

2.2. Unsupervised Learning

In unsupervised learning, ML models are trained without knowing the correct outputs, i.e., the dataset is not labeled. Typical goals for unsupervised learning methods are describing, representing, and finding meaningful characteristics of the input data. Two common examples of unsupervised learning methods are clustering and dimensionality reduction [5]. Because there are no correct outputs available in the training data, it is harder to evaluate the models, as the results cannot be compared with correct results. Unlabeled data, however, is easy to collect and can often be collected automatically without human intervention.

2.2.1. Clustering

Clustering methods try to group data samples into groups, so that the samples are similar in each group and dissimilar compared to the samples of other groups, based on some distance measure [6]. In clustering context, these groups are usually called clusters. With clustering and grouping, the difficult part is to find the correct features that should be used to determine the similarity between objects. For example, grouping a group of different animals could be done based on the color, size, number of legs or sound of the animal. Depending on which feature or features are used, the grouping results can be very different. Consequently, it can be hard to determine what are good clusters when there are no ground truth labels for the clustered data, i.e., when the correct clusters to which the objects should be assigned into are not known. Continuing on the animal grouping example, someone could group the animals based on the size and someone else could group them based on the number of legs. The resulting groups would most likely be very different but still on both cases the groups would be well defined based on the selected feature. There is no definite criteria of what is the best clustering of a specific group or dataset, or what a cluster is, as the definition of clustering or grouping can be different depending on the context [7]. The definition of good clustering depends on the goals of the cluster analysis, and a good domain knowledge is often required to evaluate the goodness of a clustering.

Because the definition of good clustering is highly dependent on the goals and context, numerous different clustering methods have been developed over the years. Each of the methods do clustering differently based on a different idea of how a cluster is defined. Clustering methods are often split into two main categories: partitional clustering and hierarchical clustering [6, 8].

Partitional clustering methods assign the data points into clusters by minimizing a selected metric, such as Euclidean distance [8]. The algorithms iterate over the data points and assign the points to clusters that minimize the metric, i.e., the points are assigned to the cluster that they are closest to, until convergence. Partitional clustering methods usually require that the number of clusters is defined beforehand. Density-based clustering methods are also included in partitional methods. Density-based methods create clusters based on the density of the data points. High density regions are considered as clusters and low density regions as noise or outliers [6]. Some examples of partitional clustering methods are k-means [9], fuzzy c-means [10], k-medoids algorithms, such as PAM [11], and DBSCAN [12].

Hierarchical clustering techniques form the clusters iteratively by either dividing the data into smaller clusters or merging smaller clusters into larger ones. Dividing bigger clusters into smaller ones is known as top-down approach or divisive hierarchical clustering and merging smaller clusters into larger ones is known as bottom-up approach or agglomerative hierarchical clustering [13]. Divisive hierarchical clustering techniques start with the whole dataset as one cluster and start dividing it into smaller ones until each data point is divided into separate clusters or a specific condition for termination is reached, whereas agglomerative hierarchical clustering techniques start with each data point as their own cluster and start merging them until all data is merged into one cluster or a specific condition for termination is reached [8]. Hierarchical clustering can generally be visualized with a dendrogram as shown in Figure 1. The sample labels are on the x-axis and the distance or dissimilarity between the clusters is on the y-axis. Some examples of hierarchical clustering algorithms are BIRCH [14], CURE [15], ROCK [16] and Chameleon [17].

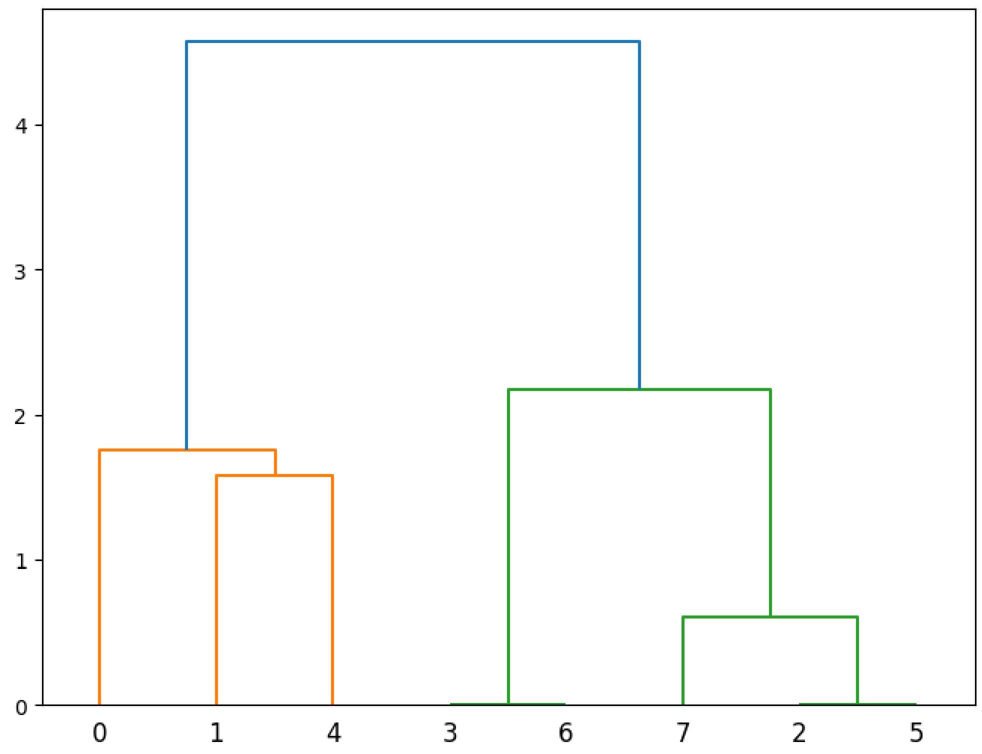


Figure 1. An example of a dendrogram.

2.3. Semi-Supervised Learning

Semi-supervised learning methods try to combine the advantages of supervised and unsupervised learning methods by using partially labeled data that contains large amount of unlabeled data with some labeled data. This way the laborious data labeling process is minimized while being able to define some criteria for model optimization [18]. With semi-supervised learning, most of the methods from both supervised and unsupervised learning, such as classification and clustering, are possible. The main challenge with semi-supervised learning is that the labeled dataset needs to be a good representation of the whole data, including the unlabeled and unseen data. Sometimes the labeled dataset has to be very big to have a good representation of the whole data, for example if there are lots of different classes to classify. In these cases it might just be better to use supervised learning, as the labeled dataset is already large enough to train an accurate classifier.

2.4. Reinforcement Learning

In reinforcement learning, the models are set free in a suitable environment, and the model performs different actions and collects feedback from the environment. This feedback is then used to train the model, so that it knows which actions it should do in each situation. Reinforcement learning approaches do not require labeled data for training the models but the models need feedback from their environment and the model must have clear definition of what is the optimal feedback. Reinforcement learning is used in AI planning, robot controlling and game playing, for example [19].

3. TEXT MINING AND NATURAL LANGUAGE PROCESSING

Text mining and NLP are fields of study that study different ways of processing textual data to transform it in such a form that a machine can understand and make use of it and to find patterns and characteristics from text. As the amount of unstructured data, for example in the world wide web, is huge and only keeps on growing, it is important to be able to process it and extract information from it.

3.1. Text Mining

Text mining has similarities with data mining, as it tries find and interesting information and useful knowledge from data. Data mining is usually used on structured data but text mining can also be used on semi-structured (e.g., e-mails, XML and HTML files) and unstructured files (e.g., PDF files) [20], i.e., text mining is used to extract information from natural language. Some examples of text mining applications are text clustering and classification, event, entity and relation extraction, and anomaly or trend detection [21, 22].

NLP methods, which will be discussed more in detail in the next section, can be used in text mining tasks, so the two are not mutually exclusive. Zhang et al. [23] use part-of-speech tagging in their construction site accident analysis using text mining. Abdullah et al. [24] use NLP and text mining for analyzing and forecasting investment decisions using data from stock market and related textual web sources.

3.1.1. Text Clustering and Classification

Text clustering and classification are techniques to organize text documents into collections that contain similar text documents in each collection [25]. Text clustering is generally an unsupervised learning technique, i.e., it is done to text data where the groups or collections are not known beforehand. Text clustering tries to separate the text documents based on some similarity measure in a way that similar documents are separated in their own groups separate from the dissimilar ones. Text classification is generally a supervised technique, i.e., the groups to which the text documents are being organized to must be known beforehand, and the algorithms must be trained with text documents that have already been assigned to these groups. Text classification tries to assign new text documents into the predefined groups based on similarity measures between the document and the groups, so that the document is classified in the group that it is most similar to. Because the similarity measures play a key part in text clustering and classification and it directly impacts the performance of the methods, it is essential to choose good similarity metric [25, 26]. Some common similarity measures are Euclidean distance, Cosine similarity, Pearson correlation coefficient and Jaccard coefficient [27].

Before the text documents can be compared together in terms of similarity, they must be represented in a way that makes the similarity comparison possible. The similarity metrics usually require that the text documents are first transformed into numerical representations, meaning that one needs to convert the text documents into multidimensional vectors. To get meaningful vector representations, one often needs to apply some preprocessing to the texts before doing the representations to normalize the texts. Common preprocessing steps are tokenization, filtering, pruning and stemming or lemmatization [28].

Tokenization is the task of dividing text into small pieces, tokens, which usually are either words or n-grams of words, i.e., pairs, triplets, quadruplets, etc. of words. Word tokenization is the simplest and it uses whitespaces to split each word into one token [28].

In the filtering step, stopwords, punctuation marks and special characters are removed from the text. Stopwords are words that do not help in comprehending the documents semantically. Stopwords vary between each language but often they are articles, pronouns, conjunctions or adverbs. The advantage of removing stopwords is that it reduces the number of words or tokens that must be considered when creating the document representation [28]. Depending on the use case and domain, the filtering can be extended to other characters or words as well, such as domain specific stopwords or digits for example.

Pruning removes the words that appear either very frequently or very infrequently in the corpus. Very frequently occurring words can be seen as corpus specific stopwords which do not help in the clustering or classification process since they appear in most of the texts. Very infrequently occurring words, on the other hand, can be seen as too rare words, which only appear in very few texts, and thus do not add any value to the clustering or classification process [28]. Pruning, much like filtering, can reduce the number of words or tokens needed for text representation, but the difficult part is finding the right thresholds to prune only the words that are too frequent or infrequent to add any value to the clustering or classification process.

Stemming and lemmatization steps try to normalize the words in the texts, so that different forms of the same word can be recognized as the same word. Stemming does this by removing suffixes and prefixes of words, so only the roots of the words are used in the text representations. Porter stemming algorithm [29] is one of the most commonly used stemming methods [28]. As an example of stemming, the words ’try’, ’trying’ and ’tried’ would all be stemmed to ’tri’. Lemmatization tries to find the dictionary form of each word. Lemmatization requires finding the part-of-speech tags for each word before it can determine the dictionary form of the words. The process of finding the part-of-speech tag and dictionary form of each word generally require an external dictionary [28]. As an example of lemmatization, the words ’try’, ’trying’ and ’tried’ would all be lemmatized to ’try’.

Representing the text documents is a crucial part of being able to compare the documents in terms of similarity, as the similarity is calculated for these representations. The three main methods used for representing text documents are vector space model (VSM), probabilistic topic model and statistical language model [25]. VSM represents each document with an n-dimensional vector, where n is the number of terms or features in the vocabulary. The representation values can be calculated for each term, with Boolean value, term frequency (TF) and term frequency-inverse document frequency (TF-IDF), for example [25]. Most commonly used method is the TF-IDF, which considers the frequency of a term inside one document and also the frequency of a term across all the documents, so it tries to find the terms that are specific for each document. Probabilistic topic model represents the documents as a collection of topics, where each topic is a probability distribution of words. Two commonly used probabilistic topic modelling methods are Latent Dirichlet Allocation (LDA) and Probabilistic Latent Semantic Indexing [25]. Statistical language model represents text as a probability distribution calculated with word sequences [25]. Most commonly used language modeling approach, used in speech recognition and information retrieval, for example, is to represent text with a product of n-gram probabilities, where the probability of the consequent word is calculated using the n previous words. In other words, the word probabilities are formed based on the frequencies of certain words appearing together in n-grams in the training data [30].

Alsmadi and Alhami [31] present their findings using different approaches to cluster email contents. They use TF with WordNet lexical database to group words into synonyms with different text preprocessing techniques to represent the email contents, and they have developed a distance based similarity measure where the maximum distance between two emails is normalized to one and the distance between identical emails is zero. Classification algorithms are used to evaluate the performance of the clustering and the results show that the they are able to get good classification results even with this unsupervised clustering based approach, but it is also noted that the performance of the classifier could be improved with a supervised approach.

Jalal and Ali [32] use TF-IDF with cosine similarity to classify research papers into five different categories based on the scope of the paper. They use the title, abstract and keywords of each paper for the classifying process. The results show that their approach can do really accurate classification into the five specified topics.

3.2. Natural Language Processing

NLP refers to a set of tools and techniques that attempt to extract a deeper representation of textual data, such as who is doing what, when, where, why, and to whom. It focuses more on linguistic and syntactic structures and components, such as part-of-speech tags, semantic role labelling, sentiment, text categorization, among others [21]. Therefore, NLP stands for techniques and algorithms that are used for automatically analyzing and representing human languages. This also includes, information retrieval, question-answering, automatic language translation, and text generations and dialogues. NLP related analysis can be split into three main categories: syntax analysis, semantic analysis, and discourse analysis [33]. Syntax analysis tries to either determine structure (s) from text or regularize the syntax structure (s) with semantic analysis. The goal of such an operation is to determine the meaning of the sentence. Discourse analysis tries to determine the meaning of text consisting of multiple sentences [33]. A common NLP based method used in feature extraction in different text mining tasks is Word embeddings. The latter are numerical vectors that are calculated for each word and they try to encode the meaning of the word in a numerical vector. Word embeddings can be calculated using models like Word2Vec [34, 35] or fastText [36, 37, 38].

4. AUTOMATED FAULT ANALYSIS USING LOG FILES

Logs are semi-structured [39] or unstructured [40] files that are produced by SW during run time which record information about the SW execution [39]. System logs are often the only way for developers and operators to make sure that the system is running as it should [40]. As the modern systems keep getting larger and more complex, the amount of generated logs is increasing as well and the manual analysis of these logs gets harder and more laborious. With increased research on data mining, automated log analysis techniques have also started to emerge [40]. In addition to playing a key role in ensuring system dependability, logs are also essential in system failure detection and analysis, when something goes wrong in the system. Two common log analysis applications related to system failures are anomaly detection [41, 42, 43] and fault or root cause analysis (RCA) [44, 45, 46].

Fault analysis and RCA are closely tied together and they are the process of finding out why a failure happens in a system. In some cases, RCA can be seen as a process that takes the analysis a bit deeper in trying to find the root cause why a fault happens, whereas fault analysis process might just try to identify different fault types. They are different from anomaly detection, as anomaly detection only tries to find failures or anomalies from logs. A common output of anomaly detection process is defining if a system execution is normal or not, but in fault analysis or RCA the execution is usually known to be anomalous. Depending on the system and the logs, the output of fault analysis or RCA process could be the system component, software function or line of code that caused an error in the execution, for example.

Basic workflow for automated log file analysis usually consists of log parsing, feature extraction and log mining [39, 40], as visualized in Figure 2. Online deployment can be considered as an optional fourth step in the workflow if the models trained in the log mining step should be deployed online [40]. Log partition might also be needed after the log parsing step if the log files contain logs from multiple different systems, and the logs originated from separate systems should be partitioned into separate logs [40].

The first phase of the automated log file analysis workflow, log parsing, has two common goals it tries to achieve: parsing raw logs into structured format [39, 40] and extracting relevant information from the logs, such as errors or failures [44]. The aim of parsing raw logs into structured format is to parse the key components, such as timestamp, IP address or severity level (e.g., "ERROR") from the logs. The log messages can then be abstracted by identifying between the constant and variable part of log message and using the constant parts to create event templates [39]. The process of parsing a raw log into structured format and abstracting the log message is illustrated with an arbitrary example in Figure 3. In this example, the raw log line is parsed to get the timestamp, severity, logging component and log message in a structured format. The log message is also abstracted by masking the variable parts: IP address and timestamp, and creating an log event template from the constant part 'Connection established' and the masked variable parts. Extracting relevant information, such as error messages, can reduce the amount of the data needed in the following steps and help focus only on the interesting parts of the logs. However, the relevant information is always dependent on the use case and the extraction must to be carefully considered, as it could also lead to loss of important information from the logs.

Similarly to other text mining tasks, during feature extraction step, interesting characteristics of the log files are extracted and transformed into numerical features. Depending on the feature extraction methods and the data that is used, some text preprocessing methods can be considered before feature extraction to enhance the feature representation. The ML algorithms that can be used in the log mining step require structured, usually numeric input, which is why the logs must be transformed into features before they can be used as input to the log mining step. Graphical features, such as graph models, can also be used but numerical features are more common in log analysis [39]. Examples of numerical feature extraction methods are log event based methods, such as log event sequence and log event count vector [39], and NLP based methods such as Word2Vec weighted by TF-IDF [41].

Moving to the log mining step, the first task is selecting appropriate ML model or models, depending on the use case, and training the selected model or models with the extracted features [39]. If there are multiple models that fit the use case, multiple of them can be trained and compared to find the best one. The main limiting factor when finding suitable ML models is the data that is being used, and more importantly if the data is labeled or not, i.e., if one should use supervised, unsupervised or semi-supervised models. The output of the log mining step can be a model that can detect anomalies by finding uncharacteristic patterns from logs or a model that can detect the root cause of a failure in a system by finding which system component seems to have caused the failure in the logs, for example.

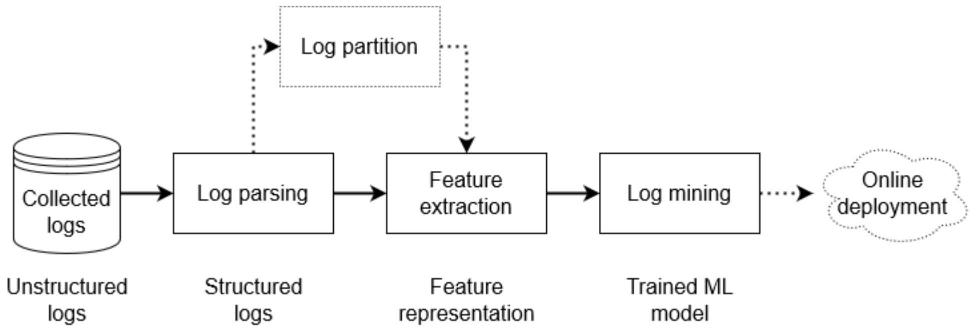
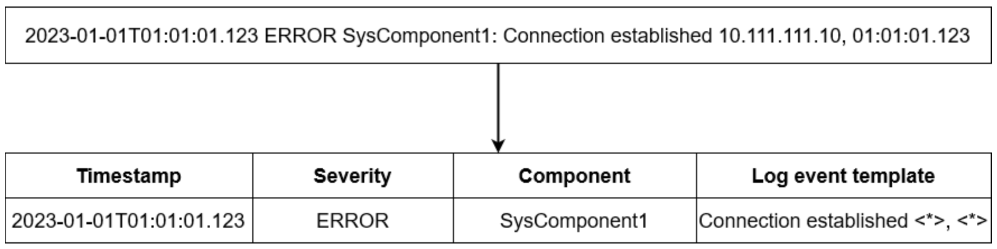


Figure 2. Basic workflow for automated log file analysis.



```
2023-01-01T01:01:01.123 ERROR SysComponent1: Connection established 10.111.111.10, 01:01:01.123
```

Timestamp	Severity	Component	Log event template
2023-01-01T01:01:01.123	ERROR	SysComponent1	Connection established <*>, <*>

Figure 3. Parsing a raw log message into structured format.

4.1. Related Work

There are lots of research done on automated log file based fault analysis but most of it is done on labeled data with the goal of classifying the faults [47, 48, 44]. Unsupervised approaches with unlabeled data are often used in log line abstraction and log line clustering [49, 50, 51]. Log line abstraction is a common log parsing method also used in automated fault analysis. Log line clustering can be used to create log file profiles and finding patterns from log files [49].

4.1.1. Fault Classifying

Jiang et al. [47] introduce an automated cause analysis approach for test alarms collected from system and integration testing named Cause Analysis Model (CAM). CAM analyzes the test logs that are collected during testing. CAM uses TF-IDF with cosine similarity to measure the similarity between new test logs and labeled historical test logs. Before calculating the similarities, only relevant logs are chosen from the historical log dataset to reduce the number of logs for which the similarity calculations must be calculated. If the highest similarity between a new test log and a historical test log is higher than a cause-specific threshold, the cause for the new log is predicted to be the same as the most similar log's cause. If the highest similarity is lower than a cause-specific threshold, K Nearest Neighbors (KNN) classification is used by summing the similarity values of the top-k most similar test logs for each cause, and predicting the cause to be the one with the highest summed similarity in the top-k most similar test logs. The cause-specific thresholds are calculated separately for each cause using the historical logs. The method of calculating the cause-specific thresholds is explained in detail in the paper. CAM is tested on two datasets with seven and six different alarm causes and is able to achieve classification accuracy of 58.3% and 65.8% on the datasets.

Amar and Rigby [48] present improvement approaches to CAM. The main improvement idea over CAM is to predict the log lines that are likely to show the cause of the failure. The aim is to flag as few log lines as possible while identifying the largest number of faults. They use log abstraction to normalize the logs into log events and find which of the log events have not been seen in the last passed log. Only those log events that are not seen in the last passed log are used for the feature extraction and similarity measures. Among the fault specific lines, the top N lines, i.e., the features with highest values, are flagged for investigation. The classification is improved by using Exclusive K Nearest Neighbors (EKNN) classification. Using EKNN classification, if any of the logs among the k-neighbours, i.e., top-k similar logs, has been labeled as a product fault, the new log will be classified as a product fault. This is done to deal with the skewed dataset, where product faults are few and far between, and to make it harder for the model to miss any product faults. The best approach, LogFaultFlagger, uses the prior knowledge of how many times a log line has been seen in past faults weighted by log line level inverse document frequency (IDF) for log vectorization. This approach is able to find 89% of all faults in the test dataset while flagging only 0.4% of the log lines.

Another study done on continuous integration testing data is presented by Sköld [44]. He is using NLP techniques to do automated root cause analysis on test logs. The logs are parsed by creating event templates where variables, such as IP-addresses, URLs and file paths are masked, and by searching only for the higher-order log events, such as errors, with keyword search. For each of the extracted higher-order log line, five previous log lines are also extracted to include context. These log groups, containing six log lines each, are then vectorized using fastText word embeddings weighted by TF-IDF. The classification is done for 16 different classes using few different models and the best results are achieved with XGBoost with an F1-score of 0.932.

4.1.2. Log Line Abstraction and Log Line Clustering

Simple Logfile Clustering Tool (SLCT) is presented by Vaarandi in his paper [49]. SLCT uses a three step clustering algorithm making two passes over the data. The first step of the clustering algorithm is identifying the frequent words from the dataset, i.e., the words that have more occurrences than a user-specified threshold. During the second step the cluster candidates are built into a candidate table with another pass over the dataset, using the frequent words information collected in the first step. The final step goes through the candidate table and marks the candidates that have a support value, i.e., density in terms of frequent words, equal or greater than a threshold as clusters. The algorithm reports the found clusters with line patterns without reporting each individual line belonging to the clusters.

Nagappan and Mladen modify the SLCT algorithm to make it more viable for log abstraction [50]. Their approach also passes through the data twice. During the first pass, the algorithm builds a frequency table that shows how many times each word occurs in each possible position in the log lines, so that the rows of the table correspond to words and columns correspond to positions inside a log line. In the second pass, every log line is inspected again. For each log line, the frequency threshold for a word to be considered constant or variable is retrieved using the frequency table. The clusters are looked for within a log line without a need for user-specified threshold, which enables the approach to find and abstract all event types in a log file, as long as the events occur at least two times in the log.

He et al. introduce Drain, an online log parsing approach with the goal of creating an efficient and accurate log abstraction method [51]. Online log parsing means that it can process log messages in a streaming way without needing all the logs beforehand. Drain uses a fixed depth parsing tree with carefully designed parsing rules as the tree nodes to make the parsing process efficient. Parsing tree is used to abstract log lines into log event templates and also clusters the log lines into clusters based on the event templates. The accuracy and efficiency of Drain is compared against four existing log parsing methods: LKE [52], IPLoM [53], SHISO [54] and Spell [55], using five different datasets. On four of these datasets, Drain achieved the highest accuracy and on the fifth one it had the second highest accuracy, and it was the fastest algorithm on all five datasets.

5. METHODOLOGY

As the study done on this thesis work is done on unlabeled data, unsupervised learning methods can be used to find answers to the research questions. Clustering is a suitable method for finding the different fault types in the radio product integration testing data and answering the first research question, as it will help group the similar faults into clusters, which can then be manually analyzed to find the fault types that form the clusters. Clustering can also be used to evaluate which of the used feature extraction methods can separate the data best in terms of fault types and answer the second research question by calculating cluster quality metrics, such as silhouette score, for the resulting clusters. Dimensionality reduction methods, such as principal component analysis (PCA), can be used to reduce the dimensionality of the used features to two, so the features and clustering results can be visualized in a two dimensional figure to visually analyze how well each feature extraction method separates the data into clusters to help answering the second research question. Based on the answers to the first two research questions, the third research question can be answered by finding out if the collection of more data is needed and if the used data processing and feature extraction methods seem to be able to separate the data in terms of fault types or if other methods should be experimented with to be able to do automated fault analysis in the future. Other steps that should be taken to enable automated log fault analysis applications in the future are most likely also found during the process of analyzing, processing, and clustering the data and these can be discussed as well to further cover the third research question.

To be able to do clustering on the data collected from the radio product integration tests, the data must be parsed and preprocessed, and feature extraction must be done on the preprocessed data. After these steps the actual clustering can be done and the results can be analyzed to find out what kind of faults can be found from each file type, can the different feature extraction methods separate the data in terms of fault types, and compare how well the different feature extraction methods seem to separate the data in terms of fault types.

The general pipeline for parsing, preprocessing, feature extraction and clustering used in this study is illustrated in Figure 4. First, the data is parsed to extract the relevant information for fault analysis and to transform the unstructured logs to structured format. This is done using keyword searches and regular expressions. After parsing the data, the log messages are abstracted using Drain algorithm [51] to normalize the log messages by finding the constant parts and masking the variable parts. Two different text preprocessing methods are used to normalize the data. The first one being a basic text preprocessing pipeline that is commonly used in general text mining tasks, and the second one being a reduced text preprocessing pipeline that could help preserving some of the characteristics of lower level log messages. Feature extraction is done on the parsed and preprocessed data using TF-IDF, Word2Vec weighted by TF-IDF and line-IDF. The extracted features are clustered using K-means clustering. Euclidean distance is used for K-means. The clustering results are analyzed using manual analysis, silhouette scores and PCA to find answers to the research questions. Implementation of the methodology is done using Python and Jupyter notebooks.

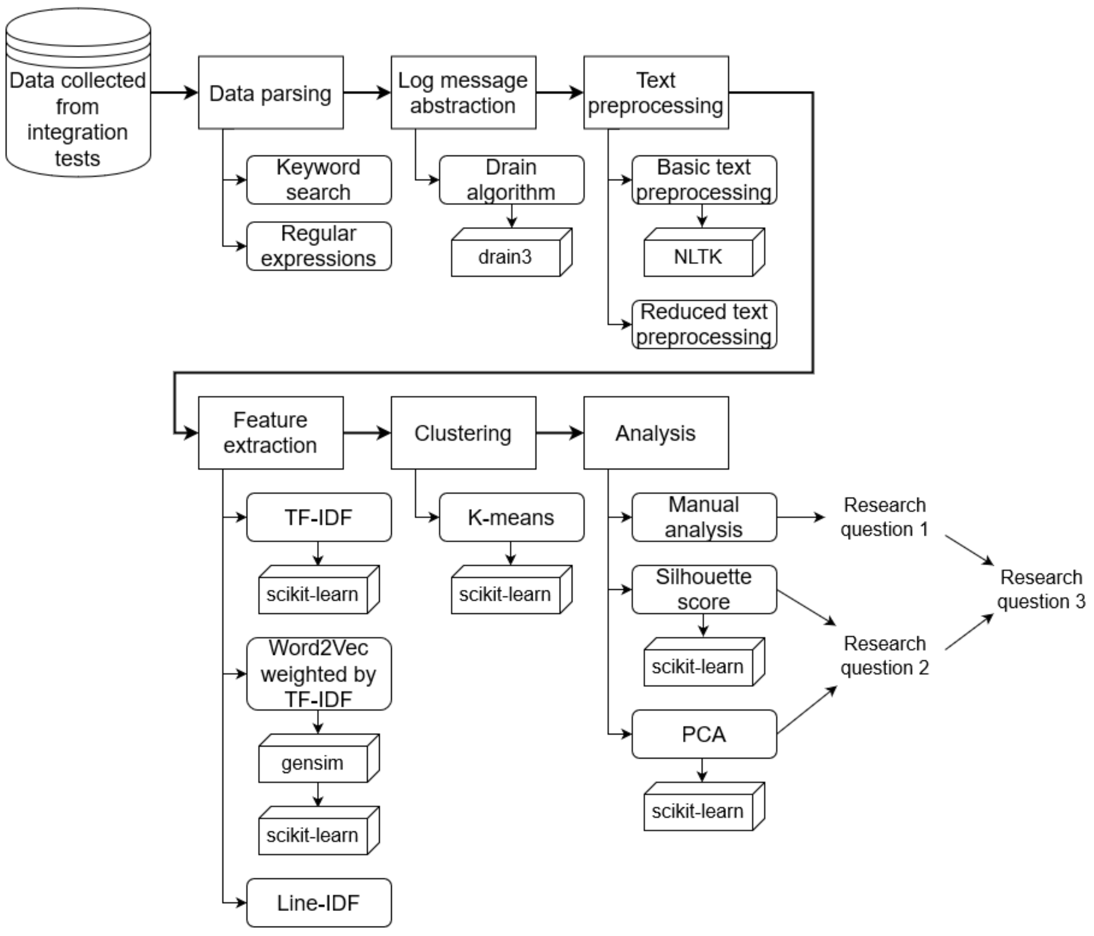


Figure 4. The general pipeline for data preprocessing and clustering used in this study.

5.1. Data Parsing

Keyword search is used to find the interesting rows from the log lines based on the logging severities. In the fault analysis use case, the interesting lines are the higher severity log lines: errors and criticals. Keyword search will extract all the log lines that contain the specified keyword, such as 'ERROR'.

Regular expressions are used to transform the unstructured logs into structured format by finding the different parts of the log messages, such as timestamps and logging entities. Regular expressions can be used to find and extract specific sets of characters from text, and with knowledge of the structure of the logs, they can be used to find and extract the different parts of the log lines.

5.2. Log Message Abstraction

Drain3 [56] Python implementation of the Drain algorithm [51] is used for log message abstraction. It uses a fixed-depth parsing tree to abstract log lines into log event templates. Before a new log message is input in the parsing tree, it is preprocessed with user-defined regular expressions. With these regular expressions, the user can mask some domain specific variables, such as IP addresses or block IDs.

The first step of traversing the parsing tree is to search a node that corresponds to the number of tokens in the preprocessed log message, based on the assumption that similar log messages will probably have same number of tokens. The next steps of traversing the parsing tree is to look at the $depth$ - 2 tokens at the beginning of the log message. $Depth$ is a parameter that is used to determine how deep the parsing tree will be for each log message. There are two other steps of traversing the parsing tree in addition to this one, which is why only the $depth$ - 2 tokens from the beginning are considered in this step. Considering the tokens in the beginning is originating from the presumption that tokens at the start of a log message are more probable to be constants. In this step only tokens that do not contain digits are considered to avoid parameter tokens. After this step Drain will have reached a leaf node.

The final step of traversing the parsing tree happens in the leaf node by choosing the most similar log group from all of the log groups inside the leaf node. Each of the log group inside a leaf node must fulfill the rules in the nodes that were traversed before the leaf node. The log groups contain the log event and a list of log IDs. The log event is a template containing the constant parts and masked parameters of a log message and presents the log messages in the group. The list of log IDs contains all the IDs of log messages that are clustered into this group. Similarity is measured as a token level sequence similarity between the new log message and the log events of all of the log groups, defined as

$$simSeq(seq_1, seq_2) = \frac{\sum_{i=1}^{n} equ(seq_1(i), seq_2(i))}{n}, \tag{1}$$

where seq_1 and seq_2 are the new log message and a log event of one of the log groups, $seq(i)$ is the i-th token of the log message or log event and n is the number of tokens in the log message and log event. The equality function equ is defined as

$$equ(t_1, t_2) = \begin{cases} 1, & \text{if } t_1 = t_2 \\ 0, & \text{otherwise} \end{cases} \tag{2}$$

where t_1 and t_2 are tokens that are compared. After the most similar log event is found, the similarity value $simSeq$ is compared against a similarity threshold parameter. If the similarity value is equal or greater than the threshold value, Drain will consider the log group as a match. Otherwise Drain will consider that there are no suitable groups for the log message.

Finally, the parsing tree will be updated depenging on if a matching log group was found for the log message or not. If Drain found a match for the log message, the ID of the log message will be added to the log group's list of IDs and the event template will be updated if needed. If the tokens in the same position are the same for the new log message and the template, the token will not change in the template, but if they are different, the token in that position will be replaced with a wild card mask (*) in the event template. If Drain did not find a match for the log message, it will create a new log group for that log message and update the parsing tree to include the new log group it just created based on the new log event.

5.3. Text Preprocessing

5.3.1. Basic Text Preprocessing

The basic text preprocessing pipeline steps are defined as:

1. stop word removal,
2. special character removal,
3. digit removal,
4. punctuation mark removal, except for underscores (_),
5. converting to lower case,
6. stemming the tokens using Porter stemmer, and
7. pruning infrequent tokens.

Underscores are not removed as they are very common in entity names, so not removing them will help preserving the entity names. An example of a special character is the newline character (\n). Natural Language Toolkit (NLTK) [57] Python library implementation of Porter stemming algorithm is used for stemming.

5.3.2. Reduced Text Preprocessing

The second text preprocessing approach uses reduced preprocessing with the steps being:

1. special character removal,
2. punctuation mark removal, except for underscores(_), forward slashes (/), colons (:), dashes (-) and dots (.)
3. converting to lower case, and
4. pruning infrequent tokens.

Stop word removal or stemming is not used. The reduced preprocessing is based on the idea that lower level logs, such as the BTS log collector logs, are not very similar to usual written natural language, and they can have slightly different characteristics, so using the same basic preprocessing pipeline that is commonly used for text mining might not be the best approach for lower level logs.

5.4. Feature Extraction Methods

5.4.1. TF-IDF

TF-IDF is a feature extraction method that measures how characteristic each word is to a document in a corpus. TF-IDF is a regularly used method for extracting text features in information retrieval and text mining tasks [32, 41, 44, 47, 58]. In this work, scikit-learn [59] Python library implementation of TF-IDF is used. TF-IDF is defined as a sum of TF and IDF:

$$TF\text{-}IDF = TF * IDF, \tag{3}$$

where TF is the count of how many times the word appears in the document. IDF measures how common or rare a word is in the corpus and is defined as:

$$IDF = log\frac{N}{DF}, \tag{4}$$

where DF is the document frequency, i.e., the count of documents that contain the word and N is the total number of documents [60].

5.4.2. Word2Vec

Word2Vec is a model framework that is made to extract meaningful word embeddings for NLP related tasks. Word embeddings are numerical vector representations of words, so they can be used in feature extraction. The Word2Vec word embeddings have been used a lot in feature extraction in different text mining or NLP related tasks [41, 61, 62, 63, 64]. Word2Vec is based on the two papers by Mikolov et al. [34, 35]. Gensim [65] Python library implementation of Word2Vec is used in this work. Word2Vec trains a two-layer neural network, with one hidden layer and an output layer, using the words in a corpus. Two separate model architectures are available for Word2Vec: continuous bag-of-words (CBOW) model and continuous skip-gram model. The CBOW model predicts the current word using the context, i.e., a specific number of words before and after the current word, while the skip-gram model tries to predict the context using the current word. In this work, the CBOW model is used. The architecture of the model is illustrated in Figure 5. The model uses the current word's context as the input, so a window size must be specified to determine how many words will be used before and after the current word. The word embeddings are learned in the projection or hidden layer and that is also where they are extracted from after the model has been trained. These word embeddings are then used in feature extraction. The output of the model is the predicted word based on the context.

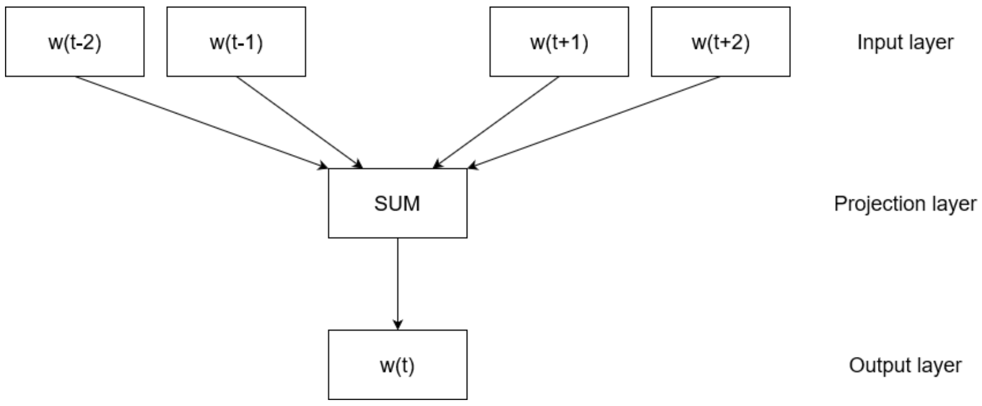


Figure 5. The CBOW model architecture.

5.4.3. Word2Vec Weighted by TF-IDF

Word2Vec model gives word embeddings for each word in a text. However for text classification or clustering tasks, the whole text needs a single feature representation, so the word embeddings must be somehow summed to get a feature representation for the whole text. TF-IDF weights are often used in this process by using the TF-IDF weights to weight the corresponding word embedding vectors and then summing the weighted word embedding vectors together [41, 61].

5.4.4. Line-IDF

Line-IDF is similar to IDF but it is calculated on the log line level instead of word level. Line-IDF is presented in the paper by Amar and Rigby [48], and they use it in two of their log line flagging approaches: LogFaultFlagger and Logliner. LogFaultFlagger is their best performing approach and it uses the prior knowledge of log lines' relation to product faults to weight the line-IDF values. Since the information about which log lines are often related to product faults is not available in the study presented in this paper, the raw line-IDF values are used. The same approach is used in the Logliner approach, which also performs well in the use case it is made for. The line-IDF score is calculated for each log line similarly to how IDF is calculated for words in Equation 4:

$$line\text{-}IDF = log\frac{N}{line\text{-}DF}, \tag{5}$$

where N is the total number of log files and $line\text{-}DF$ is the count of how many log files contain the log line.

5.5. Clustering

5.5.1. K-Means Clustering

K-means clustering algorithm is used for clustering. It is one the most established clustering algorithms [9]. K-means is a partitional clustering method and it is one of the simplest clustering algorithms [8]. Euclidean distance is commonly as the minimized distance metric with K-means [8]. K-means requires that the number of clusters k is defined before doing the clustering. The K-means implementation from scikit-learn [59] Python library is used in this work. The algorithm follows these four basic steps [8, 9]:

1. initialize k cluster centroids at random or using some statistics or prior knowledge of the used data, and for each data sample, calculate the distance to each cluster centroid,
2. assign each data sample to the cluster that has the nearest centroid,
3. recalculate the cluster centroids based on the samples assigned to each cluster:

$$c_i = \frac{1}{N_i} \sum_{x_j \in C_i} x_j, \tag{6}$$

where c_i is the new cluster centroid for cluster i, N_i is the number of samples in cluster i, C_i is the cluster i, i.e., the group of samples assigned to cluster i, and x_j is sample j in cluster i,

4. repeat steps 2-3 until there is no change in the centroids.

Some other termination rules can also be defined for step 4, such as maximum number of iterations.

5.5.2. Euclidean Distance

Euclidean distance is used as the minimized metric with K-means clustering in this work. Euclidean distance is the basic distance between two points and is calculated by taking the square root of the squared difference between the two points:

$$d_{Euc}(d_1, d_2) = [(d_1 - d_2)^2]^{\frac{1}{2}}, \tag{7}$$

where d_1 and d_2 are the points in n-dimensional space [66].

5.6. Metrics and Methods Used for Analysis

5.6.1. Manual Analysis

As a part of the exploratory analysis, the resulting clusters are manually analyzed to find the different fault categories that can be found from each file type. Different numbers of clusters are tried with K-means using TF-IDF feature extraction method with basic preprocessing pipeline and the resulting clusters are analyzed to find all the fault categories. TF-IDF with basic preprocessing method is used for manual analysis since these methods can and will be used for all file types and it is a simpler and less computationally costly approach than Word2Vec weighted by TF-IDF, which is also used for all file types, so experimenting with TF-IDF and finding the different fault types is easier. Line-IDF could also be used for log files but since it cannot be used for test case meta data files, TF-IDF is used for all file types.

5.6.2. Silhouette Score

Silhouette coefficient is a commonly used clustering evaluation method used with partitional clustering methods when the ground truth labels are not known or when the cluster quality is wanted to be described [27, 67, 68]. It uses the partitioning of a clustering algorithm to measure how well each individual cluster is defined based on the similarities between samples belonging to same clusters, and how well the clusters are separated from each other based on the dissimilarities between samples belonging to different clusters [69]. The silhoutte coefficient s can be calculated for a single sample i with equation:

$$s(i) = \frac{b(i) - a(i)}{max(a(i), b(i))}, \tag{8}$$

where $a(i)$ is the mean dissimilarity between sample i and all the other samples assigned to the same cluster with i, and $b(i)$ is the mean dissimilarity between sample i and all the other samples assigned to the neighboring cluster. As the clustering in this work is done using Euclidean distance as the minimized metric, Euclidean distance is also the similarity metric used to calculate the silhouette coefficient. This means that the dissimilarities inside and between the clusters can be viewed as the distances inside and between the clusters.

The overall silhouette coefficient for a clustering partition can be calculated as the average of each sample's silhouette coefficient [69]. In this paper the mean silhouette coefficient value for a clustering partition is referred to as silhouette score. Silhouette score can have values from -1 to 1. Silhouette score values close to -1 mean that the samples have been clustered into wrong clusters, values close to 0 mean that the clusters are overlapping, and values close to 1 mean that the clusters are well defined. Scikit-learn library [59] implementation for computing the mean silhouette coefficient is used to calculate the silhouette scores for the different preprocessing and feature extraction methods in this study.

Silhouette score can also be used to optimize the number of clusters for partitional clustering methods, where the number of clusters must be defined before doing the clustering, such as K-means. The optimal number of clusters is found by varying the number of clusters k over a range of suitable values and finding the value of k that gives the highest silhouette score [70]. In this work, this method is used to find the optimal numbers of clusters, so the resulting clusters can be visualized using PCA.

5.6.3. Principal Component Analysis

PCA is a dimensionality reduction method that tries to compress the size of data by keeping only the important information. It can be used to reduce high dimensional data into lower dimensions, so that the data can be visualized and analyzed easier. The reduction in the size of data is achieved by calculating the principal components of the original data, so that the first principal component will explain the largest variance from the data and the second component is orthogonal to the first one [71]. Principal components are calculated using singular value decomposition (SVD) but the mathematical derivation of computing the SVD and the principal components are not included in this paper, so the reader can refer for example to the articles by Abdi and Williams [71] or Bro and Smilde [72] to find more detailed mathematical explanations of the method.

The basic idea of PCA is illustrated with two dimensional toy data in Figures 6 and 7. The first principal component finds the direction where the data has the largest variance and the second component is set to be orthogonal to the first one. The two principal components are visualized on top of the original data in Figure 6. In Figure

7, the original data is projected onto the first and second principal component. The same principles apply when visualizing higher dimensional data using PCA.

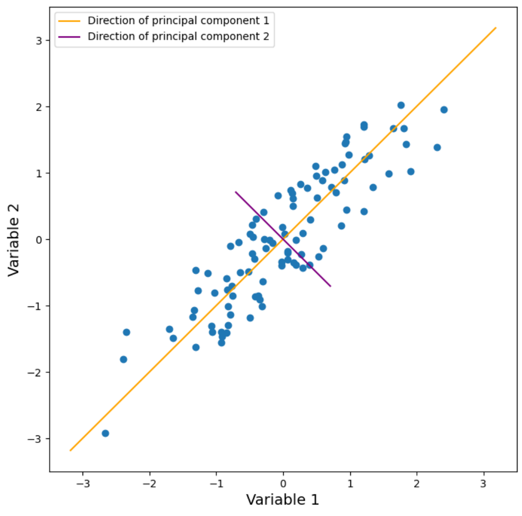


Figure 6. Principal components visualized on top of the original data.

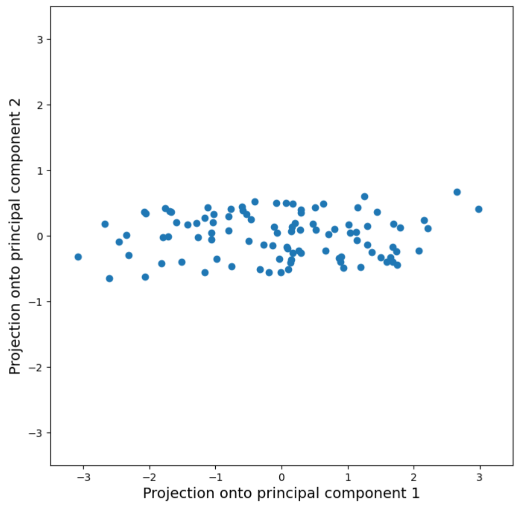


Figure 7. Data projected onto principal components.

6. DATA PREPROCESSING

The fault analysis study for radio product testing related data started by having some discussions with the test engineers to get more familiar with the testing environments, the data that is collected and used for fault analysis, and what are the possible causes for failures. Based on these discussions the scope for the data study was defined to include the automated integration tests for radio frequency software (RFSW), which are ran automatically multiple times a day. This was chosen, as there are lots of data available for these tests, and with the automated test executions, the possible causes for failures are more limited than in some other tests, as the test environments do not change much between the test runs. For the automated test executions, there are also automated test verdicts available, so for every test run, the result of the test is known, i.e., if the test failed or passed. This is important, so the study can actually focus on the fault analysis instead of anomaly detection. Most of the test runs do not have any labels for the fault types, though, so the fault analysis study is done mostly on unlabeled data.

6.1. Data Used for the Study

The data used for the study is unlabeled in terms of fault types. From the discussions with the test engineers, three relevant file types were identified that are collected automatically and can be used for the fault analysis study: test case meta data, test automation logs and BTS log collector logs.

Test case meta data is already collected and transformed to Azure Data Lake Storage as a Delta table, so the wanted data can be easily queried with SQL. Test automation logs and BTS log collector logs are still only stored in a S3 storage as parquet files, so these parquet files are downloaded manually using Python.

Some statistics and other information about the used file types are collected in Table 1 with the number of files collected, the distribution of number of files between failed and passed test cases, statistics about the size of the files and the different logging severities that can be found from the log files. Collection of test case meta data has been started the earliest, so there are more test case meta data files available than the other two file types. BTS log collector logs have been collected for the shortest time period, so there are less of those files available. The BTS log collector logs are the largest files containing over 40 000 rows on average.

Table 1. Statistics and information about the used file types.

File type	Total number of files	Number of passed cases	Number of failed cases	Average number of rows	Maximum number of rows	Minimum number of rows	Logging severities
Test case meta data	13676	6838	6838	1	1	1	NA
Test automation logs	10821	5558	5263	2834	91101	604	Info, warning, error, critical
BTS log collector logs	5558	2779	2779	41969	7916636	57	Debug, info, warning, error, very important

6.1.1. Test Case Meta Data

Test case meta data files collect general meta data about each test case. They are files with only one row with hundreds of columns containing the different meta data variables that are collected. For the study, there is a total of 13676 test case meta data files collected, 6838 of those being failed test cases and 6838 being passed test cases. There were even more files available for passed test cases but to keep the dataset balanced in terms of failed and passed test cases, the same number of files for passed test cases were collected as for failed test cases.

There are two interesting columns in the test case meta data files considering the fault analysis study: 'verification.detailed_reason' column and 'state.errorMessage' column. The 'verification.detailed_reason' column, which will be referred to as the verification detailed reason column in this text, contains the results of the automated verdicts. A test case can have multiple different verdict rules that need to be fulfilled for the test to be considered passed. An example of a verdict rule is that the throughput measurements, i.e., how much data is being sent from or received by the BTS, are above a certain threshold. The verification detailed reason column contains all the verdict rules that are set for the test case, the results for each verdict rule (failed or passed) and the reason why each rule is failed or passed. If the test execution is successful but the test still fails, the verdict rules will most likely give the reason why the test case failed. If the test execution fails before the test automation can make the verdicts, there are no verdict results available and some indication of test case execution failure is provided in this column.

The 'state.errorMessage' column, which will be referred to as the error message column in this text, collects the error messages happening on the main states or steps during the test execution. If some errors happen during these main steps that interrupt the test execution, the error messages will be collected in the error message column with the information of which of the execution steps failed, so this column usually gives a good idea of what went wrong on high level and what part of the test failed if the test execution fails.

The test case meta data files and especially the verification detailed reason and error message columns usually give a good explanation on high level why a test case has failed, and they are usually the first thing that the test engineers look at to get a general idea of what went wrong in a failed test case. Not all the error messages from the test execution are found from the error message column, though, and the actual cause for failure cannot always be derived from the high level error messages, so often other files need to be looked at to find the real root cause for failure.

6.1.2. Test Automation Logs

Test automation log files contain all the test execution steps that the test automation does during the test. There are all the calls to and responses from different services and APIs that are used during the test execution. The logging collected in the test automation logs is still happening on a high level but there are more logs collected compared to the error message column in test case meta data, as all of the test execution steps are logged here, not just the status of the main states. Test automation logs are usually the first log file the test engineers look at when a test fails, and often the cause for the failure can be derived from these logs, at least on high level, if the fault is caused by the test automation or the test environment. A total of 10821 test automation logs is collected for the study, from which 5263 are failed test cases and 5558 are passed cases. These are all the test automation logs that were available at the time of collection, and the ratio between failed and passed test cases is still balanced, so all of the available logs are used for the study.

There are two formats for the basic log lines in test automation logs, which are illustrated in Figure 8. The first format includes logging timestamp, timestamp from the logging entity, severity of the log, logging entity and log message. The second format is otherwise the same as the first one but it does not have the logging entity. The possible logging severities are INFO, WARNING, ERROR and CRITICAL. If a logging entity is included in the log, the timestamp from the logging entity, the severity, the logging entity and the log message are separated with pipe symbols (|). There are also some multi line logging statements with indentation and unstructured format, but after studying the files, most of the relevant information will be collected from these multi line statements in the well structured log lines.

<Timestamp> <Timestamp from entity> | <Severity> | <Entity> | <Log message>
<Timestamp> <Timestamp from entity> <Severity> <Log message>

Figure 8. Log line formats for test automation logs.

6.1.3. BTS Log Collector Logs

BTS log collector logs contain logs from the BTS and radio that are used in the test. The BTS log collector logs can give a deeper insight on what is happening in the BTS and radio and if a failure is caused by either of them. These logs can be important if an actual product fault happens in the test, as the they can show on the software component level, which component is causing the fault, and for which software team the fault report regarding the product fault needs to be directed to. 5558 BTS log collector log files are collected for the study with 2779 failed test cases and 2779 passed test cases. Again, more passed test cases were available than failed test cases, but to keep the dataset balanced, the same number of passed test cases were collected as failed test cases.

The format for the BTS log collector logs log lines is shown in Figure 9. The log lines contain the IP address from which the log is collected, a running hexadecimal number, BTS component 1, timestamp, BTS component 2, logging severity flag, the logging SW component and log message. The possible severity flags are DBG, INF, WRN, ERR and VIP, corresponding to debug, info, warning, error and very important severities. The BTS component 1 and BTS component 2 are some HW component identifiers on different levels. The test engineers are more interested in the SW component identifier, as this is usually needed for the fault report in case of a product fault.

```
<IP address> <Hex> <BTS Component1> <Timestamp> <BTS Component2> <Severity flag>/<SW component>  <Log message>
```

Figure 9. Log line format for BTS log collector logs.

6.2. Parsing the Data

Further processing and studying of the files require that the relevant information regarding the failure causes is parsed and extracted from the raw files. The log files require more parsing than the meta data files, as the columns used from the meta data are already parsed information.

6.2.1. Parsing Test Case Meta Data

The first step of extracting relevant information from the test case meta data files is extracting the verification detailed reason and error message columns from the files. The error message column only contains error messages happening in the main states of the test execution, so it does not need any more parsing, as all the error messages can be relevant in the fault analysis. The verification detailed reason column contains all verdict rules that the test case needs to pass, so there can be some rules that are passed and some rules that are failed. For fault analysis, the failed rules are the relevant ones, so only the failed rules will be extracted by using keyword search for further processing from the verification detailed reason column.

6.2.2. Parsing Test Automation Logs and BTS Log Collector Logs

Test automation logs and BTS log collector logs are actual log files, so there is more parsing to be done compared to the meta data files. First, the relevant log lines are extracted from the full logs, then the extracted log lines are parsed into structured format and finally the parsed log messages are abstracted into log event templates.

Test automation log files usually have thousands of log lines in them and BTS log collector logs can have hundreds of thousands or even millions of log lines, as shown in Table 1, so the relevant log lines in terms of fault analysis need to be extracted from the logs, as trying to analyze the faults using thousands or even millions of log lines would be very difficult. According to the test engineers, the reason for faults can usually be found from the error lines in the logs, so these are the lines that are extracted along with the critical log lines in the test automation logs. Warning lines could be extracted as well, but for the clustering they might only make the results worse by increasing the dimensionality and possibly adding noise if they are not relevant for the fault analysis. Keyword search is used to extract the log lines with wanted severities from the log files. The keywords used for the test automation logs are 'ERROR' and 'CRITICAL'. For the BTS log collector logs, 'ERR' keyword is used.

Parsing the extracted log lines into structured format is done by using regular expressions that fit the log line formats. The log lines are parsed into tables where each extracted part of the log line is in its own column, similarly to the illustration in

Figure 3, with the exception that the log messages are not abstracted yet into event templates. From test automation log lines, the timestamps, logging severities, logging entities and log messages are parsed. From BTS log collector logs, the IP addresses, timestamps, logging SW components, logging severities and log messages are parsed.

Abstraction of the log messages is done by using Drain3 open-source log template miner [56] based on the Drain algorithm [51]. Log event templates are parsed with Drain3 from the extracted and parsed log messages for both test automation log messages and BTS log collector log messages. The default parameters from the project repository configuration file example [56] are used for Drain3, with the exception that no extra delimiters are used.

The log event templates are then used to find the fail specific log events by considering only those log events that are seen in the failed cases and not in the passed cases, similarly to what is done by Amar and Rigby in their failure causing log line flagging implementations [48], except that all of the historical passed cases are used in this work instead of just the last passed case. The log events that are not occurring in the passed cases are more likely to be relevant to the actual cause of the failure. With this approach, the number of log lines considered per test case is much lower than if all of the extracted lines are considered, even if lower severity log lines, such as warnings, are extracted. This approach could make it feasible to include also lower severity log lines for clustering, and it is something that could be experimented with in the future, but for this work, only the error and critical severity log lines are considered, as these are be the most relevant log lines for fault analysis. Adding more log lines to the feature extraction step will also increase the risk that the important log lines or important information from specific log lines will be lost in the noise caused by non-relevant log lines.

6.3. Feature Extraction

After parsing the important information from the raw meta data and log files, the extracted and parsed data must be transformed into features, so they can be used for log mining. In this case, the text data is transformed into numerical feature vectors, so that each test case in each file type will have its own feature vector, which will then be used for clustering. Before extracting the features, some text preprocessing is done.

6.3.1. Features from Test Case Meta Data

Before preprocessing the text from test case meta data files, the parsed verification detailed reason column and error message column are concatenated as a one string for each file. This is done because after looking at the collected files, some legacy files had the error messages in the verification detailed reason column, so by concatenating these two columns the error messages will be compared against other error messages even if they occur in the other column.

Test case meta data is preprocessed with the basic text preprocessing pipeline. The threshold for considering a token as infrequent for pruning is set at five for test case meta, i.e., tokens that appear less than five times in all of the test case meta data files are pruned.

Two feature extraction methods are used for test case meta data: TF-IDF and Word2Vec weighted by TF-IDF. The features are extracted from the preprocessed string containing the verification detailed reason and error message.

6.3.2. Features from Test Automation Logs and BTS Log Collector Logs

Three different feature extraction methods with two different text preprocessing approaches are experimented on test automation logs and BTS log collector logs. The first approach is using the failure specific log event templates to form the feature vectors by calculating the line-IDF value for each failure specific log event template. The resulting feature vector's elements will correspond to the failure specific log event templates and the value of each element will be the line-IDF value of the corresponding event template. This approach does not require any text preprocessing, since only the occurrences of log event templates are calculated and the meaning or contents of the log event templates are not considered.

For the other two feature extraction methods two different text preprocessing approaches are used. The first one is the same basic preprocessing pipeline that is also used for test case meta data and the second one is the reduced text preprocessing pipeline. The thresholds for considering a token as infrequent for pruning are set at five for test automation logs and three for BTS log collector logs. A smaller threshold is used for BTS log collector logs, as there are fewer samples.

The two other feature extraction methods are TF-IDF and Word2Vec weighted by TF-IDF. The features are calculated for the parsed and preprocessed failure specific log messages, so only those log messages that correspond to one of the failure specific log templates are used for the feature representation.

For feature extraction and clustering, only those test cases that contain at least one failure specific log message are considered. If a log file does not contain any failure specific log lines it is considered to not be relevant in the fault analysis. This reduces the number of test automation logs used in the clustering step to 3311 failed test cases and the number of BTS log collector logs to 252 failed test cases. The reduction in BTS log collector logs is drastic but it makes sense, as product faults should be much more rare than other faults, and the faults found from BTS log collector logs are most likely related to product faults.

7. CLUSTERING

The faults from radio product integration tests can be separated on high level to three different categories: environment faults, automation faults and product faults. The environment faults are caused by the test environment. For example, there could be some disconnected cables or someone could open the test chamber door in the middle of a test execution and stop the test prematurely. In the case of environment errors, the test engineers must locate the cause for the error and fix it, if possible, so that the next tests will not fail because of the same problem in the test environment. Automation faults are failures happening in the test automation, such as errors in the test configuration. Test automation issues cannot always be fixed by the test engineers, so they might need to contact the team responsible for the test automation to fix these issues. Product faults are faults happening in the actual product that is being tested. In the RFSW integration tests, the product faults are usually caused by the RFSW because the hardware does not change between the test runs, but also HW failures are possible. In the case of product failures, and more specifically the RFSW failures, the test engineers must try and find the software component that is causing the failure, so they can direct the fault report about the product failure to the correct software team.

Clustering of the meta data and the log files is done using K-means clustering. Clustering is done separately for each file type to find what kind of clusters and what kind of causes for faults can be found from each file type. This will help determining which files are needed to find specific faults and if there are any differences in what kind of faults can be found from each file type. The found fault causes can also be compared with the list of possible fault causes defined with the test engineers during the discussions with them to see which of the possible faults cannot be found from the used files and what kind of data is still required to be collected to find the missing faults.

The first step of the cluster analysis is done in an exploratory manner. Different numbers of clusters are tried using TF-IDF with basic text preprocessing pipeline and the resulting clusters are manually analyzed to identify the different fault types from each file type. The fault types are also cross-checked between the file types, i.e., if the faults seem to be similar for files generated from the same test case and what kind of differences the error messages have between the files.

The second step of cluster analysis is comparing the used feature extraction methods: TF-IDF, Word2Vec weighted by TF-IDF and line-IDF with basic text preprocessing and reduced text preprocessing pipelines. The clustering done with the different preprocessing and feature extraction methods are compared using silhouette score. The clustering results got with the highest silhouette scores for each feature extraction method for each file type are also visualized using PCA, so the goodness of the clusters and the feature extraction method's ability to separate the data will also be visually evaluated.

8. RESULTS

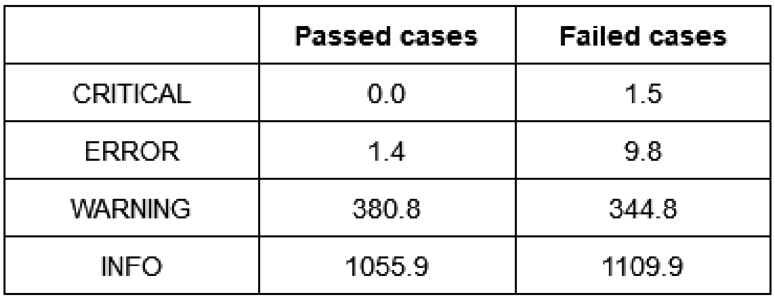
The cluster analysis for the radio product testing faults is done in two parts. The first part is the exploratory part where the different fault types are identified from each file type with cross-checking done between the file types. The full lists of different fault types are not listed in this paper, as they are confidential information. The second step evaluates how the basic text preprocessing and reduced text preprocessing pipelines with TF-IDF, Word2Vec weighted by TF-IDF, and line-IDF feature extraction methods can separate the data in terms of failure causes using silhouette score and PCA.

8.1. Exploratory Analysis

As a part of the exploratory analysis, the average number of occurrences of different logging severities in test automation and BTS log collector logs are also calculated to verify that extracting only the selected severities makes sense. The results for test automation logs are shown in Table 2 and for BTS log collector logs in Table 3. For both file types, the number of occurrences per file starts to get really high when going to the severities lower than the errors, and for BTS log collector logs, the number of errors is already high for failed cases. Including the warnings or the other lower severity logs would lead to a very big number of log lines that need to be considered for each test case, so it seems that including these lower severities might only make it harder to find the causes for faults, sticking to the assumption that the causes for faults are most likely found in the error lines. The VIP, i.e., very important, log lines in BTS log collector logs contain important information about SW startup, HW versions, etc., and there is also a big number of these log lines seen in both failed and passed cases. These log lines are considered more important than the info and warning log lines, but, as the large number of occurrences in passed cases suggests, usually these lines are still part of the normal functionality, so they are not as relevant to the fault analysis as the error lines are.

The full lists of clusters found in the exploratory analysis cannot be listed, as they are confidential information but the findings from the exploratory cluster analysis will be discussed in more detail in Chapter 9.

Table 2. Average number of occurrences of each logging severity in a test automation log.



	Passed cases	Failed cases
CRITICAL	0.0	1.5
ERROR	1.4	9.8
WARNING	380.8	344.8
INFO	1055.9	1109.9

Table 3. Average number of occurrences of each logging severity in a BTS log collector log.

	Passed cases	Failed cases
VIP	1353.0	1757.8
ERR	19.1	210.2
WRN	255.1	406.8
INF	21690.4	33640.6
DBG	7088.0	8186.9

8.2. Evaluating the Feature Extraction Methods

The used text preprocessing and feature extraction methods are evaluated using silhouette score and visualizations done with the help of PCA. To calculate the silhouette scores, the number of clusters k is varied from two to 50 to find which value of k gives the most well defined clusters based on the silhouette score. 50 is chosen as the upper limit based on the findings from the exploratory analysis, since the number of distinct clusters seems to be well under 50 for each file type, so using 50 clusters as the upper limit should be a safe choice to find all the distinct clusters based on the fault types. In the figures visualizing the resulting silhouette scores for each file type, the points where the maximum scores are achieved for each feature extraction method is marked with a black circle. Using the numbers of clusters that give the highest silhouette score for each feature extraction method, the resulting clustering will also be visualized using PCA for each file type to visually evaluate the results. In the resulting figures, different clusters are plotted in different colors and the cluster centers are marked with black crosses.

8.2.1. Test Case Meta Data

The silhouette scores got with the TF-IDF and Word2Vec weighted by TF-IDF on test case meta data are shown in Figure 10. Based on the silhouette scores Word2Vec weighted by TF-IDF seems to find better defined clusters. The clustering results are visualized using PCA in Figure 11. Visually analyzing the resulting clusters, neither of the methods seems to result in clearly better defined clusters, so based on the silhouette scores, Word2Vec weighted by TF-IDF seems to be able to separate the test case meta data better.

8.2.2. Test Automation Logs

The silhouette scores got with line-IDF, TF-IDF, and Word2Vec weighted by TF-IDF with basic and reduced text preprocessing pipelines on test automation logs are visualized in Figure 12. The basic preprocessing pipeline is referred to as the

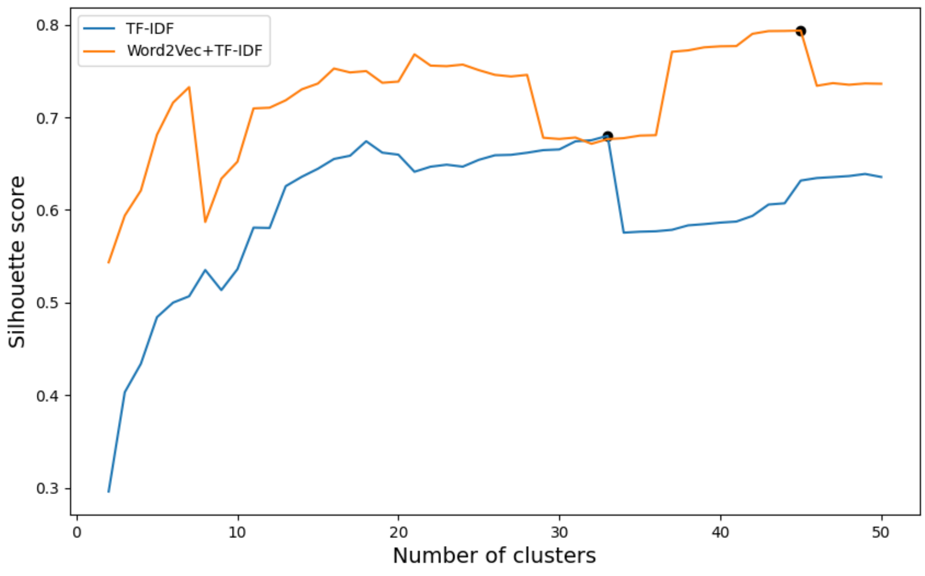
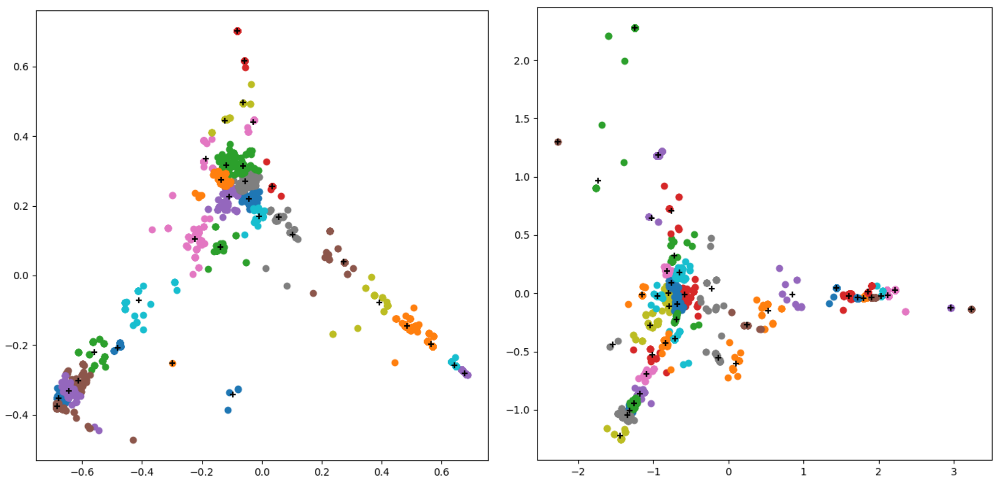


Figure 10. Silhouette scores got with the used feature extraction methods on test case meta data.

preprocessing method 1 in the legend and the reduced text preprocessing pipeline as the preprocessing method 2. With lower numbers of clusters, Word2Vec weighted by TF-IDF with both text preprocessing methods achieves better silhouette scores than the other methods but at higher numbers of clusters TF-IDF with basic text preprocessing pipeline and line-IDF achieve the highest scores. The resulting clusters are visualized using PCA in Figure 13. The scales of the axes on the Word2Vec weighted by TF-IDF visualizations are on a larger scale compared to the other visualizations, so this method is able to separate the data well also based on the visualizations. As Word2Vec weighted by TF-IDF with basic preprocessing pipeline is able to achieve the best silhouette scores at lower numbers of clusters and very comparable scores even at higher numbers of clusters compared to line-IDF and TF-IDF, overall, it seems to be able to separate test automation logs best.

8.2.3. BTS Log Collector Logs

The silhouette scores got with line-IDF, TF-IDF, and Word2Vec weighted by TF-IDF with basic and reduced text preprocessing pipelines on BTS log collector logs are visualized in Figure 14. Again, the basic preprocessing pipeline is referred to as the preprocessing method 1 in the legend and the reduced text preprocessing pipeline as the preprocessing method 2. Word2Vec weighted by TF-IDF achieves higher silhouette scores with both text preprocessing methods than the other feature extraction methods until high numbers of clusters. When the number of clusters is getting higher than 40, TF-IDF with basic text preprocessing pipeline starts to overtake with the highest silhouette scores. The resulting clusters are visualized using PCA in Figure 15. As



(a) TF-IDF, k=33 (b) Word2Vec weighted by TF-IDF, k=45

Figure 11. Visualizations of clustering done on test case meta data. Figure (a) shows the results got with TF-IDF and figure (b) shows the results got with Word2Vec weighted by TF-IDF.

the scales of axes on the Word2Vec weighted by TF-IDF visualizations are very large compared to the other visualizations, additional visualizations zoomed in on the clusters where most of the samples are assigned to are provided in Figure 16. This figure shows that there is separation between the samples even in this cluster, which is why even more than three clusters can be found with good silhouette scores with Word2Vec weighted by TF-IDF. Based on the silhouette scores and visualizations, Word2Vec weighted by TF-IDF seems to separate the BTS log collector logs best. There is no big difference between the preprocessing methods but overall the basic text preprocessing pipeline seems to be better also on BTS log collector logs with both TF-IDF and Word2Vec weighted by TF-IDF.

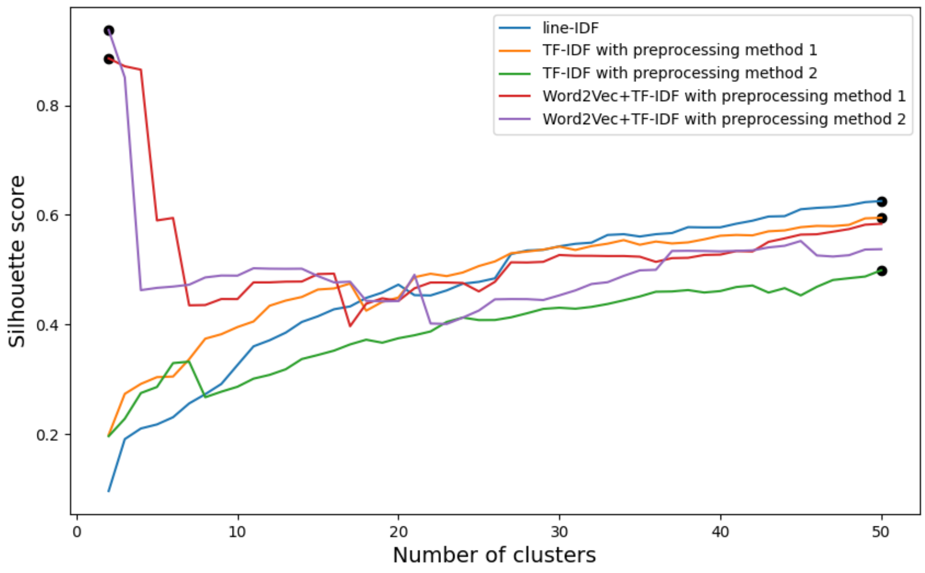


Figure 12. Silhouette scores got with the used feature extraction methods on test automation logs.

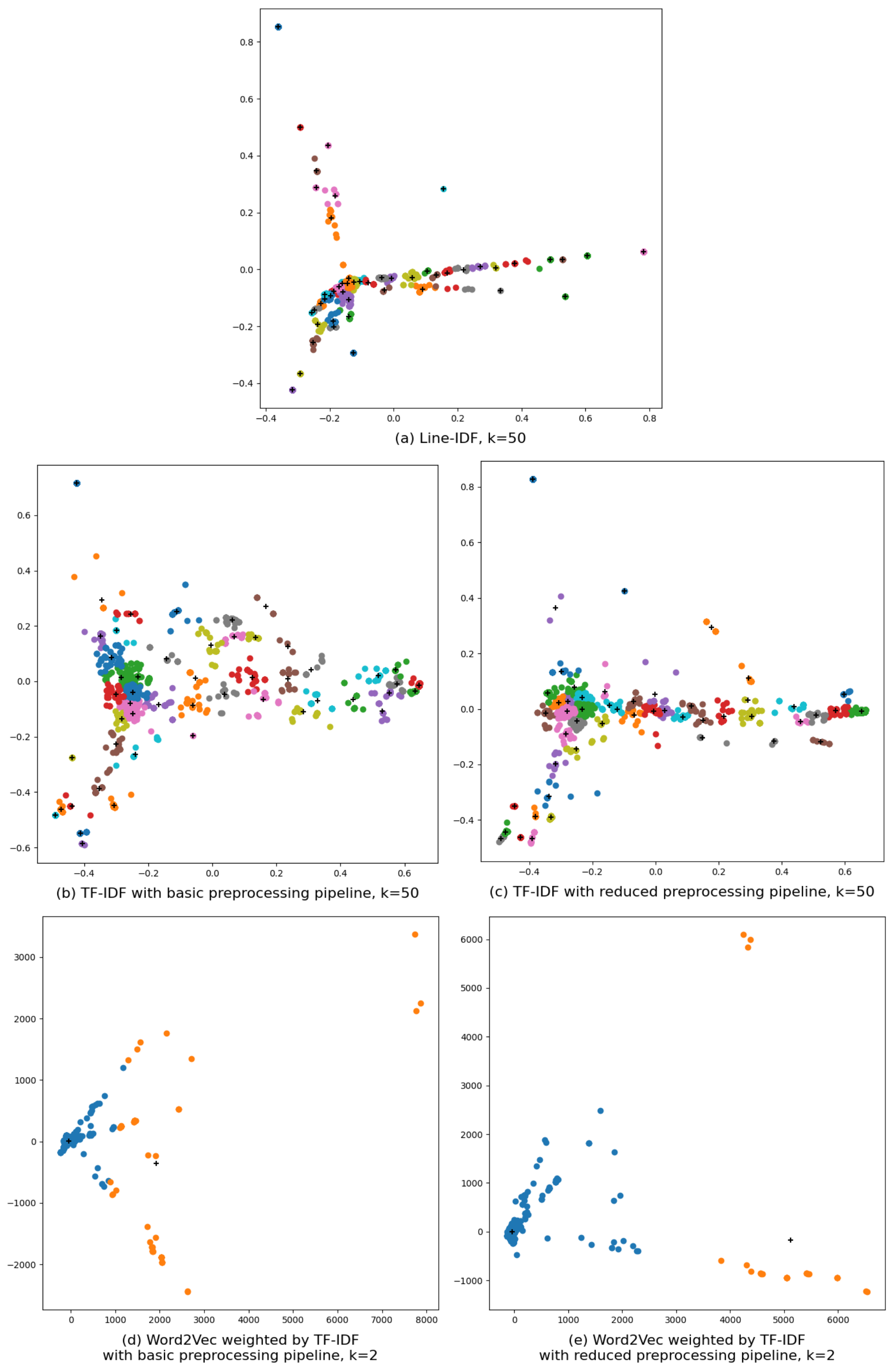


Figure 13. Visualizations of clustering done on test automation logs. Figure (a) shows the results got with line-IDF, figure (b) with TF-IDF and basic text preprocessing pipeline, figure (c) with TF-IDF and reduced text preprocessing pipeline, figure (d) with Word2Vec weighted by TF-IDF and basic text preprocessing pipeline, and figure (e) with Word2Vec weighted by TF-IDF and reduced text preprocessing pipeline.

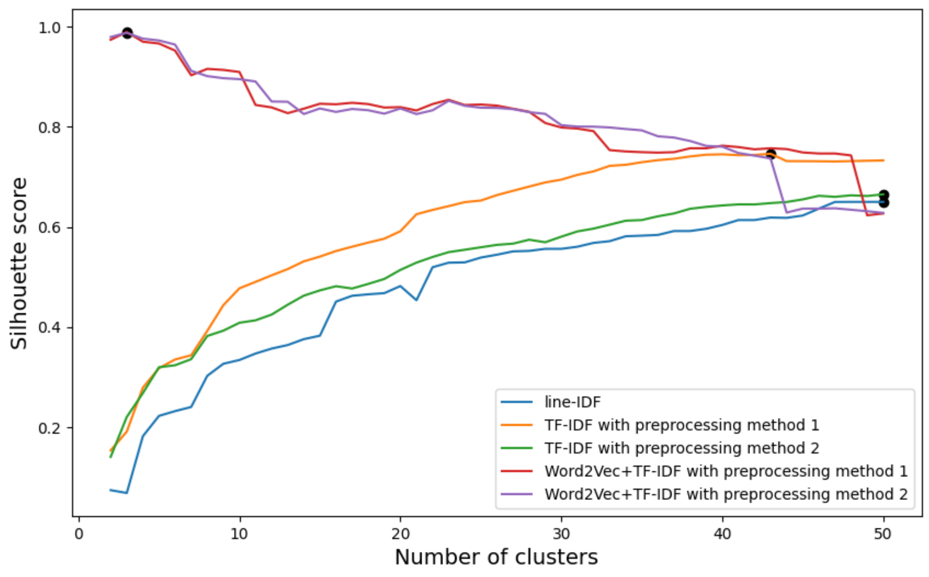


Figure 14. Silhouette scores got with the used feature extraction methods on BTS log collector logs.

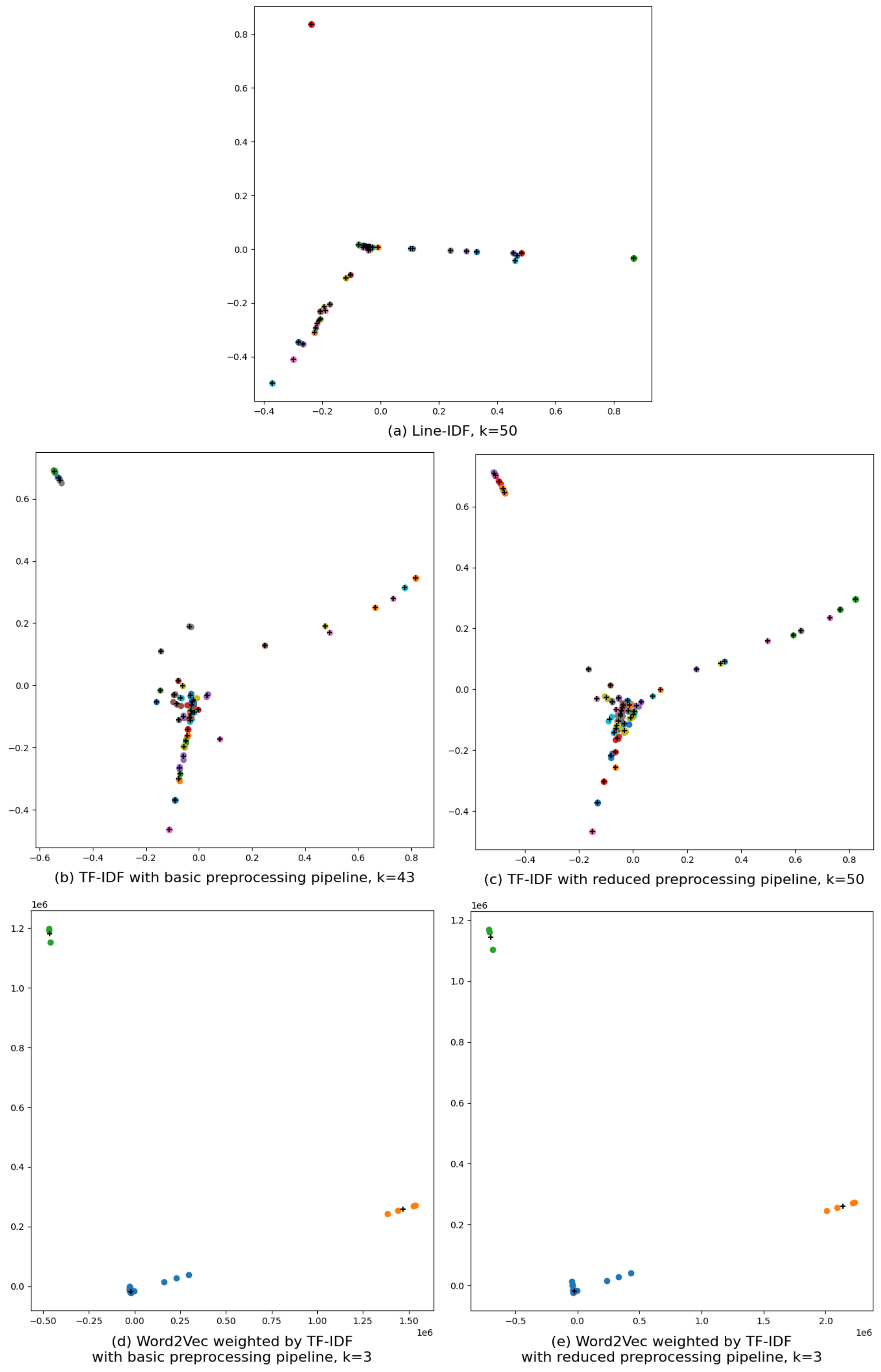
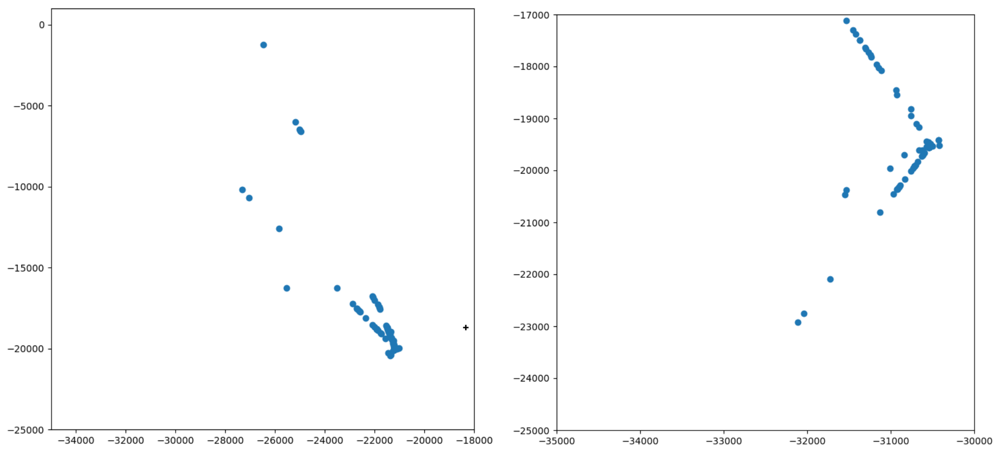


Figure 15. Visualizations of clustering done on BTS log collector logs. Figure (a) shows the results got with line-IDF, figure (b) with TF-IDF and basic text preprocessing pipeline, figure (c) with TF-IDF and reduced text preprocessing pipeline, figure (d) with Word2Vec weighted by TF-IDF and basic text preprocessing pipeline, and figure (e) with Word2Vec weighted by TF-IDF and reduced text preprocessing pipeline.



(a) Word2Vec weighted by TF-IDF
with basic preprocessing pipeline

(b) Word2Vec weighted by TF-IDF
with reduced preprocessing pipeline

Figure 16. Zoomed visualizations of clustering done on BTS log collector logs with Word2Vec weighted by TF-IDF. Figure (a) shows the results got with basic text preprocessing pipeline and figure (b) the results got with reduced text preprocessing pipeline.

9. DISCUSSION

9.1. Fault Types Found in the Used File Types

Four main groups of fault causes are found from test case meta data: environment faults, automation faults, product faults and verdict failures. The verdict failures can be viewed as test automation faults but they could be caused by environment or product faults. To highlight their appearance in test case meta data, they are listed as a separate fault group. Because the verification detailed reason column is included in the clustering, the different types of verdict failures, i.e., which automated verdict rule has failed, can be found from the clusters. This is specific for test case meta data and cannot be found from the other files as easily. Lots of different fault types can be found under each of the four main groups, and 26 different fault types are identified in total. The faults found from the test case meta data files are quite generic, so they can lead the testers in the right direction but most likely they cannot point to the actual root cause of the failure directly.

The failures found from the test automation logs are similar to those found from the test case meta but there are not as many different fault types. The first notable difference is that the different verdict failures cannot be found from the test automation logs clusters, so there are only failures related to environment faults, automation faults and product faults. A total of 15 different fault types are found from the test automation logs clusters. The main advantage of test automation logs compared to the test case meta data is found from cross-checking the logs. It is common that even if the test case meta data does not have any meaningful error message for a failed test case, the corresponding test automation logs usually contain some errors that can explain the fault that has happened. So even though the fault types found from test automation logs and test case meta data are very similar, the test automation logs seem to more reliable in collecting the relevant error messages.

The BTS log collector logs are not as easy to interpret as test case meta data or test automation logs, as the logging is done on a lower level. When analyzing the clusters, it is hard to know what are the root causes behind different sets of error messages without domain expertise, but the similarity of the error message sets can still be looked at. Cross-checking the error messages from BTS log collector logs to the errors seen in test case meta data and test automation logs in the respective test cases, there are some sets of error messages that always correspond to similar errors in test case meta and test automation logs. For example, some of the error message sets always correspond to RFSW update failures. There are definitely some trends that can be found when comparing the BTS log collector logs to the other files and these trends could help explaining the errors in more detail and possibly reveal the real root cause of some of the failures, if they are product related. However, drawing any strict conclusions from these comparisons, a domain expert would need to verify these results or there would need to be more data to back up the found trends. Currently there are only 252 test cases considered, so the found clusters, or the similar sets of error messages, have only less than 30 samples each, so it is possible that some of the trends are just coincidence.

In this study, the only available lower level logs were the BTS log collector logs. As the test environment is complex and contains multiple entities, finding the root cause for the faults will often need analysis of other lower level logs. There are no lower level logs collected related to the environment related faults currently, so this is something that could be implemented in the future. For example, UE related issues might need logs from the UE, to be able to find the actual root cause of the failure, as the higher level error messages can only point that there is something possibly wrong in the UE.

9.2. Separating the Data in Terms of Fault Types with Different Feature Extraction Methods

Evaluating the different feature extraction methods is done using silhouette score and visual analysis using PCA. Based on these evaluation methods, the basic text preprocessing pipeline seems to be more reliable than the reduced text preprocessing pipeline even with lower level logs, such as the BTS log collector logs, as better data separation and clustering is achieved across all the used file types using this preprocessing method. Word2Vec weighted by TF-IDF seems to be the best feature extraction method across all three file types to separate the data. Manually analyzing the resulting clusters got with Word2Vec weighted by TF-IDF, the clusters do actually include different fault types, so the method seems to be able to separate the data in terms of fault types well. For possible further implementations, such as automating the fault analysis process with classifiers, the reduced preprocessing pipeline and Word2Vec weighted by TF-IDF are the suggested preprocessing and feature extraction methods based on the results got from this cluster analysis.

Even though, the best silhouette scores are achieved with only two and three clusters on test automation and BTS log collector logs with Word2Vec weighted by TF-IDF, higher numbers of clusters will be needed to find all the different fault types. For example, in the exploratory analysis 15 different fault types were identified in the test automation logs, so two clusters are not enough to separate each of these fault types in a separate cluster. If the resulting clusters are wanted to be used in extended use cases, such as labeling the data in a semi-supervised approach, the used number of clusters must be defined based on the use case and how general or detailed the fault types should be inside each cluster.

9.3. Future Improvement Ideas and Steps to Automate the Fault Analysis Process in the Future

There are still more approaches that could be tried for the cluster analysis of the test case faults that could not be fit in the scope of this study, such as:

- using different clustering algorithms, e.g., hierarchical methods,
- doing the clustering by combining all the three file types together,
- trying different feature extraction methods, e.g., using the abstracted event templates instead of the log messages to create the feature vectors, and
- studying the effects of including lower severity logs, such as warnings, to the clustering results.

To improve the clustering and the log based fault analysis for test environment related faults, more data is needed to be collected from the test environment. More specifically, lower level logs from different entities from the test environment, e.g., UE, are needed to make more specific fault analysis.

An important finding during the exploratory analysis is that the parsed failure specific log lines can be parsed and provided to the testers using the same approach that is used in the clustering process. Flagging the failure specific log lines, similarly to what is done in the approaches by Amar and Rigby [48], is seen as a potential approach to ease the fault analysis, as the process is similar to how the testers are manually searching for the error messages in the files currently, but this way the error messages can be parsed automatically and the historical data can also be used to filter the errors that are more likely to be the cause for failure. This approach can be implemented as a part of the data collection and transformation pipelines to partly automate the fault analysis process.

Based on the current results and findings, the next steps that could be done to enable automating the fault analysis process are:

- adding the log parsing and log abstraction steps in data collection pipelines to provide the failure specific log messages to test engineers,
- analyzing the trends from BTS log collector logs in more detail when more data is available or with domain experts,
- using the resulting clusters to determine how the data should be labeled and start labeling the data, and
- predicting fault types for new failed test cases by using the resulting clusters with KNN classifier, for example.

If more labeled data is gathered in the future, the results from this study could be evaluated in a more robust manner. With labeled data, the goodness of the clustering and the different feature extraction methods could be evaluated by using some performance metrics, such as accuracy, and also the different parameters for different steps of the clustering process, e.g., Drain, TF-IDF, and Word2Vec parameters, could be optimized.

Labeled data would also enable some extended implementations, such as using the clustering results with the labeled data in semi-supervised approaches to create more robust classifiers to classify new failed test cases in terms of failure causes. The same log parsing and log abstraction methods introduced in the clustering work can also be used in creating classifiers using supervised ML methods but labeled data is needed. If some classifier models are created using the resulting clusters, reinforcement learning approaches can be used to enhance the model by allowing the test engineers to correct the model's predictions and using that input to tune the model.

10. SUMMARY

In this thesis work, an exploratory study is done on testing data collected from automated RFSW integration tests using clustering. Three different automatically collected file types are considered in the study: test case meta data, test automation logs and BTS log collector logs. Cluster analysis is used to find the different fault types and the benefits and deficiencies in terms of fault analysis for each file type. Using the cluster analysis results, the areas from which more data must to be collected for more precise fault analysis are identified.

For more precise log based fault analysis, more lower level logs are needed to be collected from the test environment. Currently there are no lower level logs collected from the test environment, so the ability to do log based fault analysis for environment faults is very limited.

The logs are parsed using common log parsing methods used in the literature to be able to use them in log mining use cases. The log parsing and log abstraction steps introduced in the clustering process can be used to ease and automate the fault analysis that the test engineers are doing for the failed test cases.

Three different feature extraction methods (TF-IDF, Word2Vec weighted by TF-IDF, and line-IDF) with two different text preprocessing methods (basic and reduced text preprocessing pipeline) are evaluated in terms of how well they can separate the data into different fault types. The basic text preprocessing pipeline and Word2Vec weighted by TF-IDF are performing the best based on the cluster analysis results.

The resulting clusters can be used in fault type prediction for new failed test cases in the future. Ideally, some labeled data would be used to validate the clusters and create classifiers in a semi-supervised manner for more robust and reliable classifying. To define the labels for the data, the resulting clusters can be analyzed with the test engineers to come up with meaningful cluster labels for the clusters, and those cluster labels can then also be used for labeling new test cases when manual fault analysis is done.

11. REFERENCES

[1] Leung H. & White L. (1990) A study of integration testing and software regression at the integration level. In: Proceedings. Conference on Software Maintenance 1990, pp. 290–301. URL: `https://doi.org/10.1109/ICSM.1990.131377`.

[2] Baloglu O., Latifi S.Q. & Nazha A. (2022) What is machine learning? Archives of Disease in Childhood - Education and Practice 107, pp. 386–388. URL: `https://ep.bmj.com/content/107/5/386`.

[3] Cunningham P., Cord M. & Delany S.J. (2008) Supervised learning. In: M. Cord & P. Cunningham (eds.) Machine Learning Techniques for Multimedia: Case Studies on Organization and Retrieval, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 21–49. URL: `https://doi.org/10.1007/978-3-540-75171-7_2`.

[4] Hastie T., Tibshirani R., Friedman J., Hastie T., Tibshirani R. & Friedman J. (2009) Overview of supervised learning. The elements of statistical learning: Data mining, inference, and prediction , pp. 9–41.

[5] Ghahramani Z. (2004) Unsupervised learning. In: O. Bousquet, U. von Luxburg & G. Rätsch (eds.) Advanced Lectures on Machine Learning: ML Summer Schools 2003, Canberra, Australia, February 2 - 14, 2003, Tübingen, Germany, August 4 - 16, 2003, Revised Lectures, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 72–112. URL: `https://doi.org/10.1007/978-3-540-28650-9_5`.

[6] Madhulatha T.S. (2012) An overview on clustering methods. IOSR Journal of Engineering 2, pp. 719–725. URL: `https://doi.org/10.48550/arXiv.1205.1117`.

[7] Hennig C., Meila M., Murtagh F. & Rocci R. (2015) Handbook of cluster analysis. CRC press.

[8] Saxena A., Prasad M., Gupta A., Bharill N., Patel O.P., Tiwari A., Er M.J., Ding W. & Lin C.T. (2017) A review of clustering techniques and developments. Neurocomputing 267, pp. 664–681. URL: `https://www.sciencedirect.com/science/article/pii/S0925231217311815`.

[9] Lam D. & Wunsch D.C. (2014) Chapter 20 - clustering. In: P.S. Diniz, J.A. Suykens, R. Chellappa & S. Theodoridis (eds.) Academic Press Library in Signal Processing: Volume 1, Academic Press Library in Signal Processing, vol. 1, Elsevier, pp. 1115–1149. URL: `https://www.sciencedirect.com/science/article/pii/B9780123965028000206`.

[10] Bezdek J.C. (2013) Pattern recognition with fuzzy objective function algorithms. Springer Science & Business Media.

[11] Kaufman L. & Rousseeuw P.J. (2009) Finding groups in data: an introduction to cluster analysis. John Wiley & Sons.

[12] Ester M., Kriegel H.P., Sander J., Xu X. et al. (1996) A density-based algorithm for discovering clusters in large spatial databases with noise. In: kdd, pp. 226–231. URL: `https://cdn.aaai.org/KDD/1996/KDD96-037.pdf`.

[13] Murtagh F. (1983) A Survey of Recent Advances in Hierarchical Clustering Algorithms. The Computer Journal 26, pp. 354–359. URL: `https://doi.org/10.1093/comjnl/26.4.354`.

[14] Zhang T., Ramakrishnan R. & Livny M. (1996) Birch: An efficient clustering method for very large data bases. In: ACM SIGMOD Workshop on Research Issues on Data Mining and Knowledge Discovery, pp. 103–114. URL: `https://doi.org/10.1145/235968.233324`.

[15] Guha S., Rastogi R. & Shim K. (1998) Cure: An efficient clustering algorithm for large databases. In: Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data, SIGMOD '98, Association for Computing Machinery, New York, NY, USA, p. 7384. URL: `https://doi.org/10.1145/276304.276312`.

[16] Guha S., Rastogi R. & Shim K. (2000) Rock: A robust clustering algorithm for categorical attributes. Information Systems 25, pp. 345–366. URL: `https://www.sciencedirect.com/science/article/pii/S0306437900000223`.

[17] Karypis G., Han E.H. & Kumar V. (1999) Chameleon: hierarchical clustering using dynamic modeling. Computer 32, pp. 68–75. URL: `https://doi.org/10.1109/2.781637`.

[18] Zhu X.J. (2005) Semi-supervised learning literature survey. Tech. rep., University of Wisconsin-Madison Department of Computer Sciences. URL: `https://minds.wisconsin.edu/handle/1793/60444`.

[19] van Otterlo M. & Wiering M. (2012) Reinforcement learning and markov decision processes. In: M. Wiering & M. van Otterlo (eds.) Reinforcement Learning: State-of-the-Art, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 3–42. URL: `https://doi.org/10.1007/978-3-642-27645-3_1`.

[20] Gharehchopogh F.S. & Khalifelu Z.A. (2011) Analysis and evaluation of unstructured data: text mining versus natural language processing. In: 2011 5th International Conference on Application of Information and Communication Technologies (AICT), pp. 1–4. URL: `https://doi.org/10.1109/ICAICT.2011.6111017`.

[21] Kao A. & Poteet S.R. (2007) Natural language processing and text mining. Springer Science & Business Media.

[22] Berry M.W. & Kogan J. (2010) Text mining: applications and theory. John Wiley & Sons.

[23] Zhang F., Fleyeh H., Wang X. & Lu M. (2019) Construction site accident analysis using text mining and natural language processing techniques. Automation in Construction 99, pp. 238–248. URL: `https://www.sciencedirect.com/science/article/pii/S0926580518306137`.

[24] Abdullah S.S., Rahaman M.S. & Rahman M.S. (2013) Analysis of stock market using text mining and natural language processing. In: 2013 International Conference on Informatics, Electronics and Vision (ICIEV), pp. 1–6. URL: `https://doi.org/10.1109/ICIEV.2013.6572673`.

[25] Singh K., Devi H.M., Mahanta A.K. et al. (2017) Document representation techniques and their effect on the document clustering and classification: A review. International Journal of Advanced Research in Computer Science 8.

[26] Amer A.A. & Abdalla H.I. (2020) A set theory based similarity measure for text clustering and classification. Journal of Big Data 7, pp. 1–43. URL: `https://doi.org/10.1186/s40537-020-00344-3`.

[27] Bafna P., Pramod D. & Vaidya A. (2016) Document clustering: Tf-idf approach. In: 2016 International Conference on Electrical, Electronics, and Optimization Techniques (ICEEOT), pp. 61–66. URL: `https://doi.org/10.1109/ICEEOT.2016.7754750`.

[28] Cozzolino I. & Ferraro M.B. (2022) Document clustering. WIREs Computational Statistics 14, p. e1588. URL: `https://wires.onlinelibrary.wiley.com/doi/abs/10.1002/wics.1588`.

[29] Porter M. (1980) An algorithm for suffix stripping program: electronic library and information systems. Program: electronic library and information systems 14, pp. 130–137. URL: `https://doi.org/10.1108/eb046814`.

[30] Liu X. & Croft W.B. (2004) Statistical language modeling. Annual Review of Information Science and Technology 39, pp. 3–32.

[31] Alsmadi I. & Alhami I. (2015) Clustering and classification of email contents. Journal of King Saud University - Computer and Information Sciences 27, pp. 46–57. URL: `https://www.sciencedirect.com/science/article/pii/S1319157814000573`.

[32] Jalal A.A. & Ali B.H. (2021) Text documents clustering using data mining techniques. International Journal of Electrical & Computer Engineering (2088-8708) 11.

[33] Chowdhary K.R. (2020) Natural language processing. In: Fundamentals of Artificial Intelligence, Springer India, New Delhi, pp. 603–649. URL: `https://doi.org/10.1007/978-81-322-3972-7_19`.

[34] Mikolov T., Chen K., Corrado G. & Dean J. (2013), Efficient estimation of word representations in vector space. URL: `https://arxiv.org/pdf/1310.4546.pdf`.

[35] Mikolov T., Sutskever I., Chen K., Corrado G.S. & Dean J. (2013) Distributed representations of words and phrases and their compositionality. Advances in Neural Information Processing Systems 26. URL: `https://proceedings.neurips.cc/paper_files/paper/2013/file/9aa42b31882ec039965f3c4923ce901b-Paper.pdf`.

[36] Bojanowski P., Grave E., Joulin A. & Mikolov T. (2017), Enriching word vectors with subword information. URL: `https://doi.org/10.48550/arXiv.1607.04606`.

[37] Joulin A., Grave E., Bojanowski P. & Mikolov T. (2016), Bag of tricks for efficient text classification. URL: `https://doi.org/10.48550/arXiv.1607.01759`.

[38] Joulin A., Grave E., Bojanowski P., Douze M., Jégou H. & Mikolov T. (2016), Fasttext.zip: Compressing text classification models. URL: `https://doi.org/10.48550/arXiv.1612.03651`.

[39] He S., He P., Chen Z., Yang T., Su Y. & Lyu M.R. (2021) A survey on automated log analysis for reliability engineering. ACM Comput. Surv. 54. URL: `https://doi.org/10.1145/3460345`.

[40] He P., Zhu J., He S., Li J. & Lyu M.R. (2018) Towards automated log parsing for large-scale log data analysis. IEEE Transactions on Dependable and Secure Computing 15, pp. 931–944. URL: `https://doi.org/10.1109/TDSC.2017.2762673`.

[41] Bertero C., Roy M., Sauvanaud C. & Tredan G. (2017) Experience report: Log mining using natural language processing and application to anomaly detection. In: 2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE), pp. 351–360. URL: `https://doi.org/10.1109/ISSRE.2017.43`.

[42] Anttila T. (2020), Developing a log file analysis tool: A machine learning approach for anomaly detection. URL: `http://jultika.oulu.fi/files/nbnfioulu-202006132368.pdf`.

[43] He S., Zhu J., He P. & Lyu M.R. (2016) Experience report: System log analysis for anomaly detection. In: 2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE), pp. 207–218. URL: `https://doi.org/10.1109/ISSRE.2016.21`.

[44] Sköld M. (2020), Using nlp techniques for log analysis to recommend activities for troubleshooting processes. URL: `https://www.diva-portal.org/smash/get/diva2:1523606/FULLTEXT01.pdf`.

[45] Suriadi S., Ouyang C., van der Aalst W.M.P. & ter Hofstede A.H.M. (2013) Root cause analysis with enriched process logs. In: M. La Rosa & P. Soffer (eds.) Business Process Management Workshops, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 174–186. URL: `https://doi.org/10.1007/978-3-642-36285-9_18`.

[46] Zou D.Q., Qin H. & Jin H. (2016) Uilog: Improving log-based fault diagnosis by log analysis. Journal of computer science and technology 31, pp. 1038–1052. URL: https://doi.org/10.1007/s11390-016-1678-7.

[47] Jiang H., Li X., Yang Z. & Xuan J. (2017) What causes my test alarm? automatic cause analysis for test alarms in system and integration testing. In: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), pp. 712–723. URL: https://doi.org/10.1109/ICSE.2017.71.

[48] Amar A. & Rigby P.C. (2019) Mining historical test logs to predict bugs and localize faults in the test logs. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), pp. 140–151. URL: https://doi.org/10.1109/ICSE.2019.00031.

[49] Vaarandi R. (2003) A data clustering algorithm for mining patterns from event logs. In: Proceedings of the 3rd IEEE Workshop on IP Operations & Management (IPOM 2003) (IEEE Cat. No.03EX764), pp. 119–126. URL: https://doi.org/10.1109/IPOM.2003.1251233.

[50] Nagappan M. & Vouk M.A. (2010) Abstracting log lines to log event types for mining software system logs. In: 2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010), pp. 114–117. URL: https://doi.org/10.1109/MSR.2010.5463281.

[51] He P., Zhu J., Zheng Z. & Lyu M.R. (2017) Drain: An online log parsing approach with fixed depth tree. In: 2017 IEEE International Conference on Web Services (ICWS), pp. 33–40. URL: https://doi.org/10.1109/ICWS.2017.13.

[52] Fu Q., Lou J.G., Wang Y. & Li J. (2009) Execution anomaly detection in distributed systems through unstructured log analysis. In: 2009 Ninth IEEE International Conference on Data Mining, pp. 149–158. URL: https://doi.org/10.1109/ICDM.2009.60.

[53] Makanju A., Zincir-Heywood A.N. & Milios E.E. (2012) A lightweight algorithm for message type extraction in system application logs. IEEE Transactions on Knowledge and Data Engineering 24, pp. 1921–1936. URL: https://doi.org/10.1109/TKDE.2011.138.

[54] Mizutani M. (2013) Incremental mining of system log format. In: 2013 IEEE International Conference on Services Computing, pp. 595–602. URL: https://doi.org/10.1109/SCC.2013.73.

[55] Du M. & Li F. (2019) Spell: Online streaming parsing of large unstructured system logs. IEEE Transactions on Knowledge and Data Engineering 31, pp. 2213–2227. URL: https://doi.org/10.1109/TKDE.2018.2875442.

[56] Drain3 log-template miner. URL: https://github.com/logpai/Drain3.

[57] Natural language toolkit python library. URL: `https://www.nltk.org/`.

[58] Ramos J. et al. (2003) Using tf-idf to determine word relevance in document queries. In: Proceedings of the first instructional conference on machine learning, vol. 242, Citeseer, vol. 242, pp. 29–48. URL: `https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=b3bf6373ff41a115197cb5b30e57830c16130c2c`.

[59] Pedregosa F., Varoquaux G., Gramfort A., Michel V., Thirion B., Grisel O., Blondel M., Prettenhofer P., Weiss R., Dubourg V., Vanderplas J., Passos A., Cournapeau D., Brucher M., Perrot M. & Duchesnay E. (2011) Scikit-learn: Machine learning in Python. Journal of Machine Learning Research 12, pp. 2825–2830.

[60] Kuang Q. & Xu X. (2010) Improvement and application of tfidf method based on text classification. In: 2010 International Conference on Internet Technology and Applications, pp. 1–4. URL: `https://doi.org/10.1109/ITAPP.2010.5566113`.

[61] Lilleberg J., Zhu Y. & Zhang Y. (2015) Support vector machines and word2vec for text classification with semantic features. In: 2015 IEEE 14th International Conference on Cognitive Informatics & Cognitive Computing (ICCI*CC), pp. 136–140. URL: `https://doi.org/10.1109/ICCI-CC.2015.7259377`.

[62] Xiao L., Wang G. & Zuo Y. (2018) Research on patent text classification based on word2vec and lstm. In: 2018 11th International Symposium on Computational Intelligence and Design (ISCID), vol. 01, vol. 01, pp. 71–74. URL: `https://doi.org/10.1109/ISCID.2018.00023`.

[63] Gao M., Li T. & Huang P. (2019) Text classification research based on improved word2vec and cnn. In: X. Liu, M. Mrissa, L. Zhang, D. Benslimane, A. Ghose, Z. Wang, A. Bucchiarone, W. Zhang, Y. Zou & Q. Yu (eds.) Service-Oriented Computing – ICSOC 2018 Workshops, Springer International Publishing, Cham, pp. 126–135. URL: `https://doi.org/10.1007/978-3-030-17642-6_11`.

[64] Kosar A., De Pauw G. & Daelemans W. (2022) Unsupervised text classification with neural word embeddings. Computational Linguistics in the Netherlands Journal 12, p. 165181. URL: `https://www.clinjournal.org/clinj/article/view/153`.

[65] Řehůřek R. & Sojka P. (2010) Software Framework for Topic Modelling with Large Corpora. In: Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks, ELRA, Valletta, Malta, pp. 45–50. `http://is.muni.cz/publication/884893/en`.

[66] Vijaymeena M. & Kavitha K. (2016) A survey on similarity measures in text mining. Machine Learning and Applications: An International Journal 3, pp. 19–28. URL: `https://doi.org/10.5121/mlaij.2016.3103`.

[67] Shahapure K.R. & Nicholas C. (2020) Cluster quality analysis using silhouette score. In: 2020 IEEE 7th International Conference on Data Science and Advanced Analytics (DSAA), pp. 747–748. URL: `https://doi.org/10.1109/DSAA49011.2020.00096`.

[68] Ogbuabor G. & Ugwoke F. (2018) Clustering algorithm for a healthcare dataset using silhouette score value. Int. J. Comput. Sci. Inf. Technol 10, pp. 27–37.

[69] Rousseeuw P.J. (1987) Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. Journal of Computational and Applied Mathematics 20, pp. 53–65. URL: `https://www.sciencedirect.com/science/article/pii/0377042787901257`.

[70] Kodinariya T.M., Makwana P.R. et al. (2013) Review on determining number of cluster in k-means clustering. International Journal 1, pp. 90–95.

[71] Abdi H. & Williams L.J. (2010) Principal component analysis.(2010). Computational Statistics, John Wiley and Sons , pp. 433–459.

[72] Bro R. & Smilde A.K. (2014) Principal component analysis. Analytical methods 6, pp. 2812–2831. URL: `https://doi.org/10.1039/C3AY41907J`.