FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

Master Thesis

Log-Based Unsupervised Anomaly Detection in Mission Critical System

Analysis of Logs Produced by Push-to-Talk Radios and Infrastructure in Motorola Solution's SmartConnect

Gabriela Dvořáková <kzh186@alumni.ku.dk>
Denis Drobný <htp440@alumni.ku.dk>

Supervisors
Tijs Slaats <slaats@di.ku.dk>
Lucian Tirca <lucian.tirca@motorolasolutions.com>

Title: Log-Based Unsupervised Anomaly Detection in Mission Critical System

Authors: Gabriela Dvořáková, Denis Drobný

Supervisors: Tijs Slaats, Lucian Tirca

Abstract: In software systems, logging is traditionally introduced to record information about the execution of a program. Typically, logs are analyzed by humans after an obvious error occurs or is reported by an end user. However, as software grows, it is no longer possible to monitor application behaviour and troubleshoot errors with the naked eye. Machine learning based anomaly detection can overcome these problems and eventually provide a tool to detect bugs at an early stage while they are still relatively harmless. In our thesis, we analyze time series of log data produced by Motorola Solution's SmartConnect. We aim to prove that it is possible to detect anomalies in a log dataset produced by a real-world system consisting of two main actors - the infrastructure and the push-to-talk radios connected to SmartConnect. We propose an anomaly detection architecture that consists of collecting data from the system, applying log parsing with Drain3, extracting event count and TF-IDF features, and feeding the extracted fixed-size time window form into four anomaly detection machine learning models: Isolation Forest, PCA, Invariants Mining and Log Clustering. Since this is a highly confidential domain, we had to find an efficient way to train machine learning models based solely on datasets generated in a test phase, which may be different from the datasets generated in the production environment. Two of the four algorithms, PCA and Log Clustering, achieved perfect accuracy on the test dataset in distinguishing between normal and anomalous behaviour. To evaluate the models on the unknown dataset, we asked the experts from the SmartConnect team to evaluate the predictions. They confirmed that the PCA model was able to detect another anomaly that was not known before the experiment. However, due to the complexity and the large amount of logs they were given to study, they could not tell whether the models had correctly classified non-anomalous samples. Finally, we found that our models could also be used to gain insight into the code coverage of the system tests.

Acknowledgements

First and foremost, we would like to thank Motorola Solutions Denmark for giving us the wonderful opportunity to intern there and give us a glimpse into the development of such a complex software product. We are very grateful for their efforts in making us feel comfortable in their work environment throughout the thesis writing process. It was a great experience.

We would like to thank our university supervisor Tijs Slaats for his continuous guidance, insightful comments and time spent during our meetings and reading our thesis.

Also, our big thanks go to Lucian Tirca, our company supervisor, who threw around great ideas and always helped us with our struggles. His endless patience and positivity made the whole research a pleasure.

We would also like to thank the members of the SmartConnect team for every bit of advice. We also appreciated the fruitful discussions with Maarten Faddegon and Jakub Slupecki at Motorola and their reviews throughout the process.

Not to forget, we send hugs to our very informal supervisors Miguel Angel Tovar(ish) and Laura Victoria Florez, who helped us complete the mission.

Contents

List of Abbreviations

AD **A**nomaly **D**etection

IDF **I**nverse **D**ocument **F**requency

IF **I**solation **F**orest

ML **M**achine **L**earning

MSA **M**icro**s**ervices **A**rchitecture

MSI **M**otorola **S**olut**i**ons

OCC **O**ne-**C**lass **C**lassification/Learning

PCA **P**rincipal **C**omponent **A**nalysis

PTT **P**ush **T**o **T**alk radio

SC **S**mart**C**onnect

t-SNE **t-**Distributed **S**tochastic **N**eighbor **E**mbedding

TF-IDF **T**erm **F**requency–**I**nverse **D**ocument **F**requency

1. Introduction

Nowadays, the Internet and software systems are being used even in an old-fashioned area like communication via push-to-talk devices. Even though it was not so long ago that this was only done over analog networks. Today, the traditional use case can be complemented by software that detects poor analogue network coverage and eventually switches to a digital connection if necessary.

The Motorola SmartConnect [1] system was developed for this purpose. Since this software system serves people in critical professional positions such as police officers or firefighters, it is crucial that the SmartConnect infrastructure is as robust as possible.

In order to improve the robustness of the system, we are trying to propose a solution that is able to speed up and automate the process of handling a fault.

When something goes wrong, the first step is to identify that a problem has occurred. Then it is desired to be able to answer when and where the problem occurred so that one can respond to it.

In order to monitor the state of a software system and troubleshoot issues, it is a good practice for developers to implement logging. Motorola SmartConnect is a complex system that consists of many different services. It produces dozens of gigabytes of log data per day in logs that are increasingly difficult to evaluate manually [2].

Not only is it far too much information in terms of size, the logs from different parts of the software can be located at different places and it is unrealistic to expect one person to uncover the internal relationships between different services that lead to errors.

It is also not tangible to use human workers to observe the live logs of the system. Furthermore, it would be desirable to anticipate potential faults and raise alarms while they have not yet done any damage. But in reality, we usually don't know about a problem until something crashes, despite red flags might have been raised earlier, when the system started behaving in a non-standard way. In that gap, between the moment when the software appears to be in a state that is not stable and leads to an error and the actual error, can be a space for intervention that can get the program back on the right track.

The goal of our thesis is to address all of these problems. In our research, we will investigate whether anomaly detection techniques can be used to develop a reliable solution that makes the error recovery process less dependent on human intervention [3] [4] [5] [6]. The solution should help the developers of Motorola SmartConnect, the customer support team and finally the end users who would have a better tool for their critical communication.

Looking at the logs generated by the telecommunication system as time series data should allow us to experiment with machine learning techniques [7]. These techniques should allow us to explore scenarios that lead to faulty runs of the system.

After proving that it is indeed possible to observe anomalies in such data, we want to propose an automated tool capable of performing this cumbersome work autonomously, monitoring the Motorola SmartConnect software system and raising an alarm when a critical finding is present.

The output of our tool would initially serve as a fairly specific piece of information that a responsible person can use and proceed with solving the reported problem with significantly narrowed scope of the issue and further investigate for the alleged problem.

In this thesis, we attempt to define the steps necessary to achieve such a solution. We also show which possible strategies and tools can be used in each of the steps, reason and justify which possibilities are the optimal ones for our use case.

1.1 Research Questions

In this work, we will investigate anomalies in Motorola SmartConnect logs and apply machine learning based anomaly detection techniques to the dataset.

First, we conducted a systematic literature review on research work that has been done so far in the field of anomaly detection on log data. There has been a good amount of work done before, so we can evaluate which methods have been used on problems similar to ours and what their advantages and disadvantages are. Based on this, we select some approaches that seem to be well suited, and we argue why it is useful to apply them to our data.

Since the main goal of our thesis is to apply anomaly detection methods to a real-world dataset, it may be noisy, which is often the case for large system logs that were not intended as input to machine learning-based anomaly detection in the first place. We also need to find out if the logs are generated in a way that is suitable and sufficient for anomaly detection. Thus, our first research question is:

RQ1: *Can anomaly detection be applied on the log data that are at the moment produced by Motorola Solutions SmartConnect system?*

The first research question lies at the heart of our research. To find an answer, we will develop and follow a methodology to study log-based anomaly detection in SmartConnect data. We consider the first question answered if we can find a means to group logs based on their underlying events. Once identified, the anomalous or non-anomalous class of these groups or clusters should be distinguishable. This could be done both by exploring the dataset and by conducting experiments.

Since the challenging part of the experiments is to design an appropriate embedding of logs after extracting log templates, our second research question is:

RQ2: *Does a weighted event vector representation of raw log messages logged by Motorola Solution's SmartConnect production system provide sufficient information for spotting anomalous outliers in the data?*

If this is the case and we are able to detect the outliers in plots based on our embedding, it should be possible to find an algorithm that can do this task automatically. Since we have large datasets available, we plan to use machine learning algorithms, which in theory should be more scalable and flexible for this kind of problems in large software systems. Therefore, we are looking for

an answer to the next research question:

RQ3: *Which anomaly detection techniques and approaches are applicable on time series data produced by Motorola SmartConnect system?*

Assuming we find some algorithms that meet our expectations and are well suited for this use case, we will proceed with the selected ones and perform a series of experiments. We will evaluate them and justify the results. In the case of poor results, we will elaborate on what could be improved in the machine learning approaches or what would be a better logging strategy that would give us better results. So the final question to be addressed in the thesis is:

RQ4 *How are the techniques performing on the given log dataset and why they do or do not yield satisfying results? If not, what could be improved about the log that would increase the performance of anomaly detection?*

1.2 Related Work

Log-based anomaly detection has been widely studied. Anomaly detection research usually follows similar steps. First, a log parser is used to extract log templates from unstructured log data. Then, logs are transformed into a numerical feature vectors using the log templates. Last, anomaly detection techniques are applied. Different approaches to each of these steps have been presented in previous work on this topic.

Xu et al. [8] was one of the first to use Principal Component Analysis (PCA) to detect anomalies in log data. They generated event templates based on the source code and used the event count matrix as input to PCA. To form an input, Xu uses the concept of windowing by session, where all logs with the same session ID are grouped together. The event count matrix is constructed such that each vector in a row represents events in a session with the same ID and each column vector is an event type. The dataset used for their experiments was collected from Hadoop Distributed File System (HDFS). A similar approach was used by He [9], which also used an event count matrix. To extract features, they group log data into three groups: fixed windows, sliding windows, and session windows.

Invariant Mining (IM) method was developed by Lou et al. [5], also using the log event count matrix as input. IM mines the invariants between log events from the event count vectors. Event count vectors that do not satisfy the determined invariants are considered as anomalies.

Since Deep Learning performs better than traditional machine learning as the amount of data increases [10], neural networks have recently been applied to the problem of anomaly detection in system logs.

Du et al. [11] used a Deep Neural Network (DNN) model consisting of Long Short-Term Memory (LSTM) units to model logs as a natural language sequence. The model is trained with non-anomalous log data. In this way, the LSTM can automatically learn the normal behaviour, forecast the next event type, and identify anomalies when it differs from the actual event type. Similarly, Zhang et al. [12] and DeepLog [11] also use an LSTM deep learning

approach. The former borrows an idea from Term Frequency-Inverse Document Frequency (TF-IDF) by considering all logs in each time window as one document and using TF-IDF weight as a feature representation. The latter work differs from other DNN approaches in that it preserves the timestamp when encoding the log message and performs anomaly detection for each log entry rather than for each session.

1.3 Outline

In Chapter 2, we introduce the domain of our problem by describing the SmartConnect product from Motorola Solution, whose log data is examined in the thesis. We also introduce the reader to the software architecture of SmartConnect.

In the following chapter, Chapter 3, methods relevant to our work and used in our approach are analyzed in detail.

Methodology (Chapter 4), explains exactly how we proceeded with our solution. It starts with a simple example that provides the motivation for an anomaly detection tool in SmartConnect. We then outline the exact steps we took and describe them comprehensively. More specifically, it consists of data collection, log parsing, feature engineering and anomaly detection. Finally, it also describes how are predictions mapped back to log entries.

Later in Chapter 5 we describe the data we work with in our research. From the test dataset to the data used for validation. Some concrete examples of anomalies that occur in the system and can be recorded by logs are presented.

Chapter 6 is devoted to experiments. It discusses in detail what we can learn from datasets by examining the data using data exploration techniques. It provides a strong intuition for what we expect to learn when we apply machine learning before we actually do it. The metrics we use for evaluation are explained along with working out the setup for the experiments.

Chapter 7 presents the results we obtained by applying four different machine learning techniques to our anomaly detection problem. We explain why the different machine learning methods give slightly more or less accurate results on our datasets.

Finally, in Chapter 8, we reflect on our research and suggest new questions that arise from our work that we think may be worth investigating further in the future.

2. Motorola SmartConnect

In this chapter, we introduce the reader to the domain of the analyzed product. We aim to detect anomalies in logs produced by software system called SmartConnect by company Motorola Solutions.

We introduce the business perspective of the product which should help the reader gain insight into why the mission critical system would benefit from an anomaly detection tool. Also, we provide a high level introduction into the software architecture of the solution that is crucial to understanding what anomalies can happen, where logs are created and how to collect them.

2.1 Domain Description

Two-way radio (also a push-to-talk, informally a walkie-talkie [13]) is an electronic device that enables a group of people to communicate.

A two-way radio works by converting audio signal to radio waves that are transmitted through the air to receivers. On the receiving end, the waves are converted back to audio signal which allows the recipients to hear the original message. There are two alternatives of what kind of signal is being transmitted through the air - it can be either analogue or digital. The advantage of radios that support digital signal is that they can transfer various types of data over the channel, not only the audio. On the other hand, analogue transmission has been the standard for audio data transmission for many years.

Two-way radios are using frequencies between 30MHz and 1000MHz [13]. The interval between 30MHz and 300MHz is referred to as Very High Frequency and the remaining upper part is called Ultra High Frequency [13].

2.1.1 Motorola SmartConnect

SmartConnect is a software product by company Motorola Solutions. The state of the art Motorola two-way radios [14] support automatic switching in favor of the source strongest signal.

The classical way of interconnecting a set of walkie-talkies is through narrowband land mobile radio (LMR) sites. However, due to specifics of the network, there is many use-cases where a push-to-talk device is out of range of LMR [15].

There are many different environments where customers of Motorola Solution's operate and it must be made sure that they get the best connectivity possible. However, not everywhere LMR coverage is sufficient. Indoor areas such as hospitals, offices or schools are usually equipped with high quality WiFi connection which can be exploited instead. Also, if mobile broadband, such as LTE [16], is more accessible than LMR, cellular data can be also taken advantage of. And for the least accessible areas, radio may connect to a satellite modem to ensure the customer doesn't lose contact with colleagues.

Motorola products are compliant with Project 25 (P25 for short) which is an LMR standard that is well suited for fast, secure and interoperable connection [17].

A radio with SmartConnect support is able to automatically switch to LTE, WiFi, satellite broadband or even wired internet connection ensuring continuity of push-to-talk voice communications in case strength of the LMR signal drops below specified RSSI (received signal strength indication) threshold. The radio is able to switch back to LMR when the signal strengthens with no user intervention required [1] [18]. The structure of a Motorola PTT radio network is schematically explained in Figure 2.1.

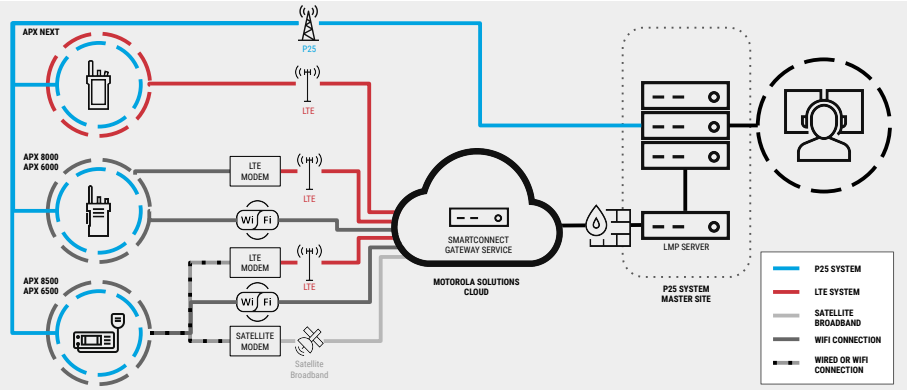


Figure 2.1: Overview of Motorola two-way radio network showing the various technologies that can be exploited to interconnect push-to-talk radios [15].

When a radio utilizes the SmartConnect technology, voice packets are being transferred through the broadband bearer to a cloud-based gateway. This gateway in the cloud connects to the LMR master site. This allows the radios communicating over SmartConnect call back in the LMR system [15].

Therefore, Motorola Solution's SmartConnect helps teams stay connected no matter where they are located by leveraging various means of voice packet transmission.

2.2 Software Architecture of SmartConnect

Now let's have a look at the core concepts that form the whole cloud based gateway functionality from the software architecture perspective.

Having familiarity with the architecture of SmartConnect is important for being able to tackle the anomaly detection problem in an informed way. It gives us a better understanding of what are the weak links that are prone to error about the software. In this thesis we want to target anomalies based on logs. Knowing the architecture also helps with identifying what parts of the system produce logs that can be collected and extracted information from.

The software follows the *Microservices Architecture* (MSA). Nadareishvili et al. [19] define a *microservice* as an independently deployable component of bounded scope that supports interoperability through message-based communication.

Fowler [20] understands the microservice architectural style as a way of building a single application by connecting a set of small services. Each of the services runs in its own process and communicates with lightweight mechanisms.

Another attribute of such an architecture is that management of the service is decentralized, therefore it allows for the individual services to be written in various programming languages and use different technologies for storing their data.

A collection of services that communicate together form a *system* [21].

2.2.1 Deployment

In the analyzed system, the individual services are deployed as Docker[1] containers in Kubernetes Pods[2] which are self-standing applications inside of Kubernetes[3] clusters.

This approach allows the services to be decoupled as each one resides in its own isolated virtual container.

As described in more detail in Chapter 4.2 on Data Collection, the services log each at a different location by default. In order to analyze the behaviour of the system holistically, an extra layer for processing logs has to be introduced. SmartConnect, like many others, for this use-case utilizes the Elasticsearch engine that also runs as one of the services in the cluster.

High level view of a Motorola SmartConnect Kubernetes cluster is presented later when discussing the way we deployed our tool for downloading logs from the Elasticsearch service in Figure 4.4.

2.2.2 Elasticsearch

Elasticsearch[4] is an open-source, distributed search engine. Gormley and Tong [22] argue that it can be used for exploring data at unprecedented speed. It is mostly recognized for its exceptional performance on text based data. Operations such as full-text search, structured search and analysis are supported efficiently. Elasticsearch can also be defined as a NoSQL database since it gathers JSON based `documents` of a specific `type` together and multiple `types` are organized into `indices`[5].

2.2.3 Messaging

As mentioned earlier, services in an MSA application may be designed in a different way and two services are free to use two dissimilar sets of technologies. That makes the communication within a system more complicated than calling a function - what one would do in a traditional, monolithic architecture.

Two main strategies of passing messages between system's services are *synchronous* and *asynchronous* communication. One of the most common types of synchronous communication is Representational state transfer (REST) [21].

On the other hand, asynchronous communication promotes autonomy between services as the communicating client does not need to wait for the response. For implementation of asynchronous protocols, the concept of a *broker* is introduced. A broker is a centralized entity with high-availability [21].

[1] `https://www.docker.com/`
[2] `https://kubernetes.io/docs/concepts/workloads/pods/`
[3] `https://kubernetes.io/`
[4] `https://www.elastic.co/elasticsearch/`
[5] `https://www.elastic.co/blog/what-is-an-elasticsearch-index`

In SmartConnect, services are passing values asynchronously. In order to achieve some high-level goal, services form a chain of 1 or more microservices. Each link of the chain receives an input, processes it and if needed, forwards its output through a broker to another microservice(s) that take(s) this data as input.

For dealing with asynchronous messages, `RabbitMQ`[6] brokers are deployed. `RabbitMQ` is an open-source publisher/subscriber message broker.

2.2.4 Storing Data

In microservices architecture systems, microservices that are immutable and stateless are favoured [21]. Therefore, data that the internal state of a microservice comprises of needs to be persisted in storage that is external to memory of a microservice application.

Traditionally, a database is utilized for persisting data externally to the application. However, it is well known that there are major set back when it comes to performance of read/write operations that can be improved by deploying supporting caches [23].

Performance of the system is fairly crucial as, among others, UDP (user datagram protocol) audio packets of push-to-talk radio calls are being transferred and processed in the system. In order to satisfy strict requirements on jitter in the audio signal, a NoSQL data storage Redis[7] service is running inside of the Kubernetes cluster to serve requests for the stored data rapidly.

[6] https://rabbitmq.com/documentation.html
[7] https://redis.io/

3. Literature Review

Before we begin the actual development of our anomaly detection tool, we first needed to establish some core concepts and review the good work that has already been done in this area.

Therefore, in the following sections, we will introduce the notion of anomaly detection. We will present ideas that the reader should be familiar with in order to fully understand anomaly detection techniques and, by extension, our work.

We then explain the three main categories of machine learning - supervised, semi-supervised and unsupervised. We discuss what their distinctive features are when applied in the anomaly detection domain. We then go into detail about four different machine learning algorithms and argue why they are the most suitable candidates for our task.

We also explain what we understand by a log, log template, and discuss the most common techniques for log parsing and log-template mining.

We conclude the chapter with an analysis of various log parsing tools. The overview allows us to argue which log parser would be best suited for our application.

3.1 Anomaly Detection

Anomaly detection is a problem of finding patterns in data that do not follow the expected normal behaviour represented by the majority of data points. It follows that defining the normal behaviour is one of the crucial challenges of anomaly detection. The unusual patterns are also called *outliers* or *anomalies* and anomaly detection is often called *outlier detection*. Outlier was defined by Ord [24] as an outlying observation that appears to be significantly different from other members of the sample in which it occurs. In other words, the statistical properties of the anomalous data points are not in alignment with the rest of the data. Outliers vary by domain and can occur for a variety of reasons, such as fraudulent behaviour in credit card fraud, computer network intrusions, system failures, mechanical failures in industrial applications, deviations due to natural behaviour, or human error.

Anomaly detection is also related to the term *noise*. Noise is defined as an undesirable phenomenon in data that is not of interest to the analyst but hinders data analysis [25]. Noise is caused by an external factor unrelated to the distribution that generates the data [26] and leads to excessively complex models with degraded performance [27].

In our thesis, we focus on anomaly detection in *time series* data. Time series outlier analysis examines anomalies in the behaviour of data over time [28]. An outlier in time series data is a data point that does not follow a common behaviour, either a general long-term trend or a tendency of the data to increase or decrease in seasonal patterns.

Originally, outliers were detected manually by the hand of a domain expert. Nowadays, the focus of anomaly detection is on automatic detection of anomalous behaviour. It has been proven on many occasions that the problem of automatic anomaly detection can be successfully tackled by learning from data [9] [7].

We can divide anomaly detection methods into three broad categories: *supervised*, *semi-supervised* and *unsupervised* anomaly detection. Supervised learning algorithms are trained by examples and require a dataset that is already labeled. Unsupervised methods, on the other hand, do not require any labeled data at all. Semi-supervised methods combine both approaches; in addition, some semi-supervised approaches assume that normal class instances are labeled.

3.1.1 Supervised Anomaly Detection

Supervised methods require previously labeled data, where the labels describe whether the behaviour in each data instance is normal or abnormal (anomalous). The main focus of supervised learning methods is then to derive a model from the labeled data that maximizes the discrimination between the classes (normal and anomalous). A drawback of the supervised method is that it can only be trained on the types of anomalies introduced in the training examples. It cannot handle previously unseen anomalies.

3.1.2 Semi-supervised Anomaly Detection

Semi-supervised learning is defined as a machine learning paradigm that studies how computers, or even natural systems like humans, learn in the presence of both labeled and unlabeled data [29].

3.1.3 Unsupervised Anomaly Detection

Unsupervised learning, unlike supervised learning, does not require any prior knowledge about the data. One of the main difficulties faced by anomaly detection is unlabeled data. In most cases, logs with labeled anomalies are not available. The process of log annotation would have to be done manually by the experts - developers who are familiar with the system. Along with the growth of data volume, labeling becoms very time consuming. For this reason, in practice, anomaly detection mostly relies on unsupervised learning to deal with unlabeled data. Unsupervised anomaly detection algorithms learn what normal behaviour looks like. Based on these observations, systems based on unsupervised methods detect anomalies as outliers that are significantly different from other examples. Moreover, any type of anomaly can be detected by these systems. Thus, the main difference between supervised and unsupervised learning techniques is that the former focuses on discriminating concept classes, while the latter rather focuses on data characterization [30].

One-Class Classification

When it comes to anomaly detection, it is common to use a special method called One-Class Classification (or One-Class Learning) formulated by Tax [31].

In OCC, labels are only provided for instances of the normal class, while labels are not required for classes describing anomalies. In fact, it is discouraged that they occur in the training dataset. There should be so few instances of outliers that they do not form a statistically representative sample of the positive concept, where positive refers to anomalous class [32].

Thus, in a semi-supervised mode, the model that captures normal behaviour is created and used to identify anomalous behaviour. A class describing normal data is used primarily because it is more readily available, while a labeled dataset that would cover all anomalies is difficult to obtain [25].

Due to the nature of our data, we used unsupervised anomaly detection methods in a one-class manner for this study.

3.1.3.1 Isolation Forest

Isolation Forest or iForest [6] is an ensemble approach for anomaly detection. Majority of existing model-based methods attempt to model normal behaviour, and then a deviation from the normal region that does not fit the model well is considered an anomaly [33]. Isolation Forest approach, on the other hand, isolates anomalies directly without using the distance from the previously defined normal region. It exploits two properties that can be observed in anomalies. Anomalous instances are the minority of data points and their attribute-values are very different from normal instances. Due to these properties, anomalies are susceptible to a concept called *isolation*, which is the main idea behind Isolation Forests. Liu et al. [6] defined isolation in their research paper as "separation of an instance from the rest of the instances".

Isolation Forest starts by selecting a random attribute and then creates a random partition between the maximum and minimum values of that attribute. This process is applied recursively until all samples are isolated, which can be represented by a binary tree structure (Isolation Tree of iTree). Then, the number of partitions performed to isolate a point is equal to the length of the traversal path from the root node to a terminating node. The iForest is built by adding a given number of iTrees obtained by randomly generated partitions, and their averaged traversal path lengths are then used as a measure of anomaly score. It is easy to see that the isolation of anomaly instances happens closer to the root of the tree, hence the path lengths are also shorter.

Since the anomaly score is the average path length of iTree, which corresponds to an unsuccessful search in the binary search tree, the equation for the anomaly score is derived from the analysis of the binary search tree. Let n be the number of instances in a dataset X. The anomaly score s of an instance $x \in X$ is defined as:

$$s(x, n) = 2^{-\frac{E(h(x))}{c(n)}}, \tag{3.1}$$

where $h(x)$ a length of a traversed path from the root node to a terminating node x, $E(h(x))$ is the average of $h(x)$ in all isolation trees and $c(n)$ is a normalization factor defined as:

$$c(n) = \begin{cases} c(n) = 2H(n-1) - (2(n-1)/n), & \text{if } n > 2 \\ 1, & \text{if } n = 2 \\ 0, & \text{otherwise} \end{cases}$$

where $H(i)$ is the harmonic number estimated by $ln(i) + 0.5772156649$.

The Isolation Forest algorithm operates in linear time complexity and is particularly suitable for large and high dimensional datasets due to its low memory requirements. It is proven to be an accurate and effective anomaly detector.

Anomaly Detection Forest

Recently, a variant of the Isolation Forest algorithm has been studied by Sternby et al. [34] for the specific use case of Anomaly Detection - Anomaly Detection Forest. It is also commonly referred to as One-Class Isolation Forest. It falls under the category of One-Class Classification algorithms.

In their work they claim that the algorithm outperforms the state-of-the-art algorithms Isolation Forest and One-Class Random Forest for the task of anomaly detection in the one-class learning setting. They argue that IF has an intrinsic bias towards labeling similar unusual normal instances as anomalies.

They try to improve that by introducing two new concepts:

- *Anomaly leaves* with the aim of catching anomalies with feature values that are not contained in the range of the subsample of tree training observations.
- *Isolation level*, which is a lower size bound on the training samples in a node during training, and determines when anomaly leaves should be created rather than further subdividing the training samples.

Although the results seem promising, we decided not to proceed with this algorithm because the reference implementation is proprietary. Our goal is to optimize the original IF algorithm so that it performs reasonably well when learning from normal data only.

3.1.3.2 PCA

One problem with high dimensional data is that it is often noisy and even redundant. In such a case, dimensionality reduction is a useful tool to address this problem. Principal Component Analysis (PCA) is one of the most commonly used dimension reduction methods. The reduction is done by projecting the data onto a lower dimensional subspace that captures the "essence" of the data [35]. The similarities and differences in the data become visible in the new coordinate system.

PCA uses projection methods on a given set of input data points with many input features. Once a PCA model is obtained, a distance from test log entries (either positive or negative) to normal space can be calculated to detect anomalies. Correlations in the original feature space are searched for and the combination of variables that maximizes variance is found to preserve all information with minimal redundancy. A new, more representative and compact feature space is generated. This feature space with lower dimensions is called the *principal components*. The features in the principal components are uncorrelated and ordered by the proportion of the data variance that each feature captures.

To put it formally, let's say we have an m-dimensional vector $\mathbf{Y} \in \mathbb{R}^n$ with n standardized input features in the original data. The PCA model decomposes $\mathbf{Y}$ into two parts [4]:

$$\mathbf{Y} = \hat{\mathbf{Y}} + \widetilde{\mathbf{Y}} \tag{3.2}$$

Where $\hat{\mathbf{Y}}$ represents the modeled part of the projection - the selected principal components. $\widetilde{\mathbf{Y}}$ represents the residual part of the projection. The modeled part is obtained by projecting the original data $\mathbf{Y}$. The projection is shown in the equation 3.3, where $\mathbf{P} \in \mathbb{R}^{n \times k}$ is the PCA loading matrix, k is the number of principal components, and the matrix $\mathbf{C} = \mathbf{P}\mathbf{P}^{\mathbf{T}}$ is a projection matrix. Thus, $\mathbf{C}$ is a matrix for a projection of the original data onto a model space, so called normal space S_d with n dimensions.

$$\hat{\mathbf{Y}} = \mathbf{P}\mathbf{P}^T\mathbf{Y} = \mathbf{C}\mathbf{Y} \tag{3.3}$$

The remaining $(n - k)$ dimensions form the residual subspace, also called the anomaly space S_a. $\widetilde{\mathbf{Y}}$ corresponds to the projection of $\mathbf{Y}$ onto the residual subspace S_a. The equation 3.4 shows how the $\widetilde{\mathbf{Y}}$ can be mathematically computed.

$$\widetilde{\mathbf{Y}} = (\mathbf{I} - \mathbf{C})\mathbf{Y} = \widetilde{\mathbf{C}}\mathbf{Y} \tag{3.4}$$

When PCA is applied to an anomaly detection task, for each new input its projection is computed. An anomaly can be detected with a large change in variable correlation if normal behaviour is not preserved. If this is the case, the projection on the residual subspace S_a is increased. As a side effect, unusual values of the magnitude of $\widetilde{\mathbf{Y}}$ can be observed. In other words, an anomalous vector is farther away from the normal space S_d.

The squared prediction error (SPE) in the equation 3.5 is a useful evaluation metric to measure the distance from S_d.

$$SPE = \| \widetilde{\mathbf{Y}} \|^2 = \| \widetilde{\mathbf{C}}\mathbf{Y} \|^2 \tag{3.5}$$

SPE is used as an anomaly score: The higher the prediction error, the more anomalous the data instance. A process block is considered an anomaly if:

$$SPE \leq \delta^2 \tag{3.6}$$

where δ^2 is a threshold for the error. A standard implementation of the threshold δ^2 in anomaly detection is based on the *Q-statistics* proposed by Jackson [36]. According to this, an anomaly is detected if the following holds:

$$SPE = \| \widetilde{\mathbf{Y}} \|^2 > Q_\alpha \tag{3.7}$$

where Q_α denotes a Q-value statistics providing a $(1 - \alpha)$ confidence level.

3.1.3.3 Invariants Mining

A linear program invariant is a predicate that always holds the same value under different normal executions - different workloads or inputs. Lou et al. were the first to propose automatic anomaly detection by mining linear invariants from logs [5]. It is based on the assumption that log sequences provide enough information about the execution paths of the system. Linear invariants are extracted from execution path properties by analyzing log sequences that follow workflow logic. The ideal candidate for the dataset for mining invariants are positive log samples. We know that some examples of linear relationship between logs can also be found in the Motorola SmartConnect system dataset. There are steps in the call logic, such as subscribing to a call group, calling, and unsubscribing that are recorded and could potentially be exploited for anomaly detection. The occurrence of the above messages in pairs is an expected normal behaviour. Therefore, the number of these event types should be the same in one execution of the program.

We can formally define a linear invariant as a linear equation:

$$a_0 + a_1 x_1 + ... + a_m x_m = 0, \tag{3.8}$$

where x_i is the event count of the event type with identifier i and $\theta = [a_0, a_1, ...a_m]^T$ is the vector representing the coefficients. It holds that:

$$\mathbf{X}\theta = \begin{bmatrix} 1 & x_{11} & x_{12} & \dots & x_{1m} \\ 1 & x_{21} & x_{22} & \ddots & x_{2m} \\ 1 & \vdots & \vdots & \vdots & \vdots \\ 1 & x_{n1} & x_{n2} & \dots & x_{nm} \end{bmatrix} \theta = 0 \tag{3.9}$$

Log datasets that contain information about the specific execution flow, such as session ID or job ID, are best suited for invariant mining because they reflect the execution workflow of that session or job. An execution workflow consists of smaller elementary units called elementary workflow structures, such as *sequential*, *branched*, *joint*, or *looping* structures. For example, in the execution flow shown in Figure 3.1, we can see a sequential workflow structure from A to B, a branched workflow structure from B to C or D, and a shared workflow structure from C to E or from D to E. Multiple execution instances of the program may run concurrently, they may execute different branches of the flow, and the logs produced may intersperse. In any case, the following should hold:

$$c(A) = c(B) = c(E) \tag{3.10}$$

$$c(B) = c(C) + c(D), \tag{3.11}$$

where $c(x)$ denotes the number of log messages of event type x.

Intuitively, invariant mining reveals the linear relationships between the logs and their respective events in the program workflows, e.g., the invariant $c(B) = c(C)+c(D)$ tells us that there could be a branched or shared elementary structure in the workflow.

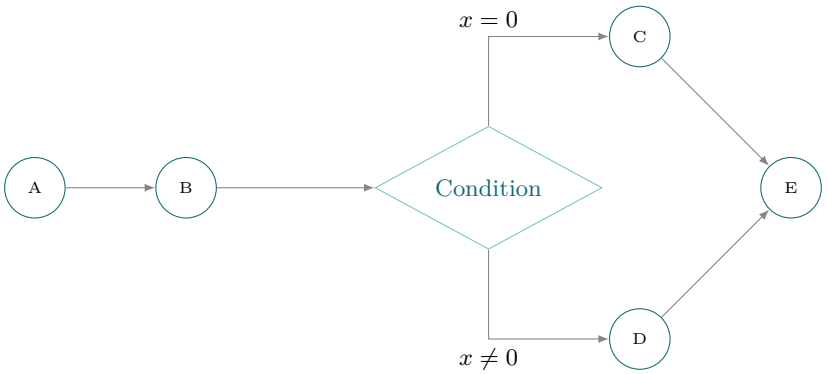


Figure 3.1: An example of an execution flow of a program that sequentially executes A and B, then based on a given condition branches and proceeds with either execution of C or D, and finally runs E.

In invariant mining, anomalies are detected by checking the invariant rules, which are broken by some log events. This not only detects an anomaly, but we can also see the logical reasoning behind an anomaly by looking at the invariants.

The workflow of the invariant mining anomaly detection approach is partially identical to other methods described in our thesis. Unstructured free-form log messages representing normal behaviour are parsed into a structured form. The second step requires grouping messages. Grouping is performed for log messages that contain cogenetic parameters (e.g., session ID). An event count vector is generated for each group. Next, sparse integer valued invariants are discovered on the event count vectors using a greedy invariant mining algorithm. Finally, the learned invariant rules are used to detect anomalies.

3.1.3.4 Log Clustering

Clustering-based approaches to anomaly detection rely on an intuitive principle. Normal data samples generate clusters, and data that do not conform to these clusters (they are far from them) are considered as anomalies. LogCluster is a tool developed by Lin et al. [3] that enables unsupervised anomaly detection by clustering log sequences that are similar.

LogCluster consists of two phases: *construction* phase and *production* phase.

Construction Phase

The construction phase can again be divided into four steps: Log vectorization, log clustering, extraction of representative log sequences, and recurrence checking.

1. **Log vectorization**:In the log vectorization step, unstructured log messages are parsed into numeric vectors by abstracting log events. Then, log sequences are obtained by grouping logs with the same task ID and vectorization by counting the frequency of the event ID in the log sequence. Event ID frequency is further weighted using the Inverse Doc-

ument Frequency based weighting. The Inverse Document Frequency (IDF) weighting technique is also used in our research and we describe it in more detail in Section 4.4.2.

While in our research we combine IDF weight with event type frequency per log sequence, in LogCluster they use *contrast-based* weighting. The intuition behind the contrast-based weighting is the observation that an event that occurs in both the test and production environments is less discriminative for anomaly detection than an event that appears only in the production environment. Events that occur only in the production environment are assigned a higher weight because they are assumed to better reflect failures.

2. **Log clustering**: The vector representation of log sequences allows the computation of distances between two vectors A and B of dimension n. The similarity measure used in LogCluster is the cosine similarity:

$$\begin{aligned} similarity(A, B) = cos(\theta) &= \frac{A \cdot B}{\|A\|\|B\|} \\ &= \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}} \end{aligned}$$

Normal and anomalous log sequences are partitioned into clusters using the agglomerative hierarchical clustering technique [37]. Initially, each log sequence belongs to its own cluster. The closest pair of clusters is then merged. The metric for measuring the distance between two clusters is defined as the maximum distance of all pairs of elements between two clusters. The stopping criterion for agglomerative hierarchical clustering is given by a distance threshold θ, which is empirically determined but initialized to 0.5. An example in Figure 3.2 shows the process of producing clusters with agglomerative hierarchical clustering.

3. **Representative log sequence extraction**: In this step, one representative log sequence called *centroid* must be selected for each of the clusters obtained. A centroid is a log sequence with a minimum score computed for each log sequence in the cluster as the average distance from the rest of the log sequences in the same cluster. Formally, we compute the score for a log sequence L_i in a cluster as follows:

$$score(i) = \frac{1}{m-1} \sum_{j=1}^{m} (1 - similarity(L_i, L_j))$$

where m is the number of log sequences in the cluster.

4. **Recurrence checking**: In the next and final step of the construction phase, each of the clusters is checked to see if it is a recurrent cluster by querying a knowledge base. The knowledge base stores the centroids of the log clusters from the past executions. If a cluster is not considered a recurrent failure, it is returned to the engineers to be manually examined.

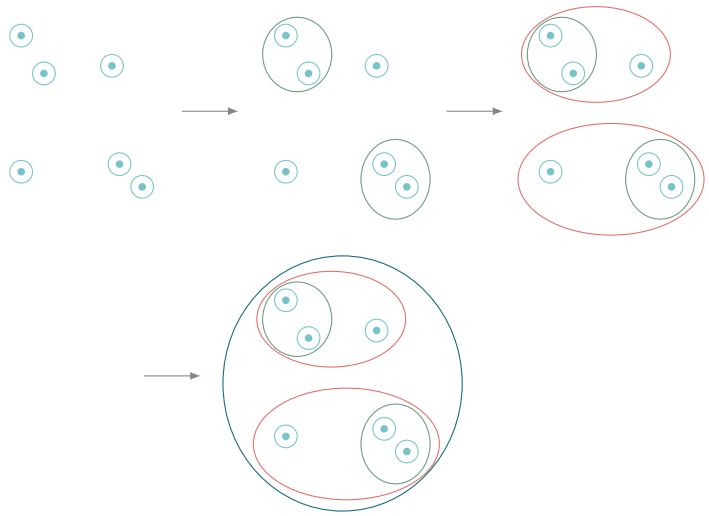


Figure 3.2: A visualization of agglomerative hierarchical clustering. At the beginning, each object is a cluster. In each step, the two closest clusters are merged. The process is finished when the maximum distance between the clusters reaches the threshold θ. This figure illustrates the case without a stopping criterion, where the process terminates when all objects are in one cluster.

Production Phase

In the production phase, logs are collected from the actual production environment (or more generally, the monitored environment). The process is then the same as in the construction phase: the logs are vectorized into log sequences and the log sequences are clustered. For each cluster, a representative is extracted. Then, the algorithm checks if all the representatives fall under one of the clusters seen earlier in the knowledge base. After that, engineers are required to manually examine the log sequences that have not been seen before for anomalies. After resolution, the knowledge base is updated with new clusters.

3.2 Log Parsing

The time series data we are working with is a sequence of *logs* produced by a development or production system. *Log* is a textual trace of variable length runtime information designed to be easily readable for humans.

Let's consider a set of log entries produced by a particular system. Each log entry is recorded by the service that performs the action described in the log. In general, a log typically contains a *timestamp* (which records *when* a particular event occurred) and a *message* (free-form text describing the event). A log may also contain additional metadata, such as *log level* (log severity describing how important a log message is, e.g., INFO, DEBUG, ERROR, WARN) or *program name* (name of the node that caused the event). An example of properties in a log entry generated by Motorola SmartConnect is described in Appendix A.1.

In our thesis, we consider only the *message* and *timestamp* parts of the log. Since log messages are usually written into the source code by different developers, we defensively assume that logs contain only these two properties. *Timestamp* is added by default by the majority of the most commonly used logging frameworks [38] [39] [40], so we will assume that *timestamp* is present along with the necessary *message*.

According to Legeza et al. [41], structured logging is an approach where each log entry is represented in a format that can be easily processed by a computer, such as XML, JSON, or other well-defined structures. When this is not the case, we call logs that do not follow a strict format *unstructured*.

Most commonly, [38] [39] [40], *timestamp* comes as the first field of the log line, so can be easily parsed even if the log does not meet the definition of *structured* log.

However, even with *structured* logs, there is a part we call *message* that contains the contents of the log and answers questions like "what", "why", and "what's going to happen next" [41].

It can be seen that each log message consists of a *constant* part and a *variable* part. Both can possibly be empty and the *variable* part may consist of zero or more *variables*. As the names imply, the constant part remains the same throughout the execution of the code, while the variable parts may change from one run to another.
The *constant* part can be understood as the part that a developer explicitly writes in plain text and the variables are generated during the execution of a system based on the state of the placeholder variables in the code.

Let's define an *event type* as a class of logs, where each of the logs belonging to that class has a *message* that follows the same *template*. This means that all of the logs in the same class have identical constant part and their variable part consists of an identical number of variables of the same data types in the same positions.

Two different messages may contain different string values when they track the same event under different circumstances. Circumstances are described by values that form the *variable* part of a log message. Variables are inserted dynamically into the plain message text, e.g. by string interpolation[1].

To have a tool to generalize a log message template in technical terms, we refer to a log template as a regular expression. The expression is encoded as a string with the constant part and regular expressions describing the variables and possibly their data types.

The purpose of log parsing is to obtain a structured input for further log analysis by extracting these event types from the natural language messages into categories. In other words, the goal is to associate an event type with each log that the system produces.

Now let's look at an example of what the log entry we just defined looks like. When a programmer writes code, he calls a logging framework at places

[1] `https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/tokens/interpolated`

where something noteworthy happens. The lines that call such a framework are shown in Listing 3.1.

```
1 Log.info("Cached default value,tenant:
2             #{inspect(tenant_id)}, param key: #{key}")
```

Listing 3.1: Example of how logging is done in source code

This line of code may, as we pointed out earlier, result in distinct log messages, as shown in Listing 3.2.

```
1 Cached default value, tenant: 12556, param key: 12
2 Cached default value, tenant: 45, param key: 4
3 Cached default value, tenant: 789, param key: 54
4 Cached default value, tenant: 12556, param key: 78
```

Listing 3.2: Possible outputs of the code in Listing 3.1

Importantly, all log messages in Listing 3.2 describe the same event type, where a default value was written to the cache and the variable part describes details of the changed context, which was different. Therefore, they are of the same event type and follow the same template.

Now it is easy to derive a log template. For example, a template for the messages we saw in Listing 3.2 can be described by the template shown in Listing 3.3

```
1 Cached default value, tenant: [0-9]+, param key: [0-9]+
```

Listing 3.3: Template for log messages in Listing 3.2, regular expressions are shown in blue.

Therefore, we say that two logs have the same event type if their messages match the same log message template. Now that we have an idea of what the goal of log parsing is, let's take a closer look at the process and technique of *log parsing*. We think of log parsing as a superset of the *log-template mining* process.

3.2.1 Log-Template Mining

Log-template mining is a process of extracting an information about the event from unstructured log messages by separating its constant and its variable parts. High accuracy of log parsing is of great importance for effective log mining [42]. For this reason, we will give a brief overview and evaluation of different log parsing approaches that have been proposed and used in the past.

Originally, log parsing was based on regular expressions written manually by developers themselves. As we can assume, manual parsing is not an ideal solution for several reasons. Indeed, designing regular expressions by hand is

a very error-prone and time-consuming process, as the size of the code may be too large for a single developer. Moreover, the ever-increasing complexity of modern software systems makes it recognizably more difficult to update and maintain these regular expressions. According to Xu et al. [8], the number of new logs introduced into Google's newly developed system amounts to about a hundred per month.

As an alternative to writing regular expressions, several automated methods have been proposed that typically operate in one of two modes. Either in *online* ([43]) or *offline* ([44], [45], [46]) mode. Most of the current log parsers work offline, which means that they need to have a full collection of logs available before the actual log processing takes place. This implies that the entire log dataset must be loaded into memory. One such example is a study by Xu et al. [47], in which they use the source code and a search function in a text editor to search for each occurrence of the logging function. From each log in the file, they extract a regular expression and compare them to find the best match. The disadvantage of this approach is that it is not always possible to have access to the source code, as is the case when using third-party libraries, etc. On the other hand, online log parsing methods process logs immediately when they arrive in a stream.
According to the log parsing technique, parsers can be classified into four main groups, including *Clustering*, *Frequent Pattern Mining*, *Heuristic*, and *Fixed Depth Tree* based methods. In the following, we will present more details and an example of a representative of each of these methods, with the overall overview available in Table 3.1.

3.2.2 Log Parsing Techniques

Clustering

A popular solution to this problem is to use data clustering algorithms. Data clustering algorithms are used in data mining or machine learning, where the set of objects is divided into groups called clusters. Objects in each cluster are similar to each other and as dissimilar as possible to objects from other clusters [44]. An example of this approach is the LogSig [45] algorithm based on message signatures. It consists of three steps:

1. *Term pair generation.* Each log message is converted into a set of pairs of terms. In other words, a pair is formed from every two words in a log message, while maintaining the order of the words. An example of such term pair generation shown in LogSig [45] is a log message:

   ```
   2010-05-02 11:34:06 Command: mkdir ".indexes"
   ```

   Then its corresponding generated set of term pairs is:

   ```
   {2010-05-02, 11:34:06}
   ```

```
{2010-05-02, Command:}
{2010-05-02, mkdir}
{2010-05-02, ".indexes"}
{11:34:06, Command:}
{11:34:06, mkdir}
{11:34:06, ".indexes"}
{Command:, mkdir}
{Command:, ".indexes"}
{mkdir, ".indexes"}
```

2. *Log messages partition.* The goal of the second step of LogSig is to use the word pairs to find a partitioning of the log messages into clusters, maximizing the objective function. The algorithm iterates over the messages, moving messages between clusters to increase the potential value of the objective function. The number of clusters is predefined.

3. *Message Signature Construction.* The final step of the algorithm constructs the message signature based on identified common pairs for each cluster.

Frequent Pattern Mining

Vaarandi [44] presented a clustering algorithm for finding frequent patterns in logs with his tool SLCT (Simple Log File Clustering Tool). A *frequent pattern* is defined as a set of elements that occur frequently in a data set [48]. As we described earlier, we can also think of event templates as a set of constants that occur frequently in log messages. SLCT reformulates the problem of finding event templates as a problem of mining frequent itemsets. The algorithm consists of three steps, with two passes over the data. The algorithm requires a threshold of support, which is set by the user beforehand.

1. During the first step (data summarization), the algorithm first performs a pass to build a summary vector from words and their frequencies. After the summary vector is created, a vocabulary is constructed. To optimize the size of the vocabulary, only frequent words (dense 1-regions) are stored. A word is considered frequent if it occurs at least N times in the dataset, where N is the user-specified threshold for support. This step is a modification of the Apriori algorithm for finding frequent itemsets [49].

2. In the second step, another pass is made over the data to assemble all cluster candidates into a candidate table. The entire dataset is processed to form a cluster candidate from the line if a combination of the dense 1-regions occurs on the line. If the cluster candidate exists, it is inserted into the table with incremented support value. Otherwise, it is inserted into the table with support value 1.

3. In a final step, the cluster candidates are inspected, and all candidates whose support values are equal to or greater than the support threshold are reported as clusters.

Heuristics

IPLoM, proposed by Makanju et al. [46], is an example of a heuristics-based log parsing method. IPLoM stands for *Iterative Partitioning Log Mining*, where their algorithm partitions the log messages into their respective clusters through a 3-step hierarchical partitioning process. In the final 4th stage, IPLoM generates discovered clusters and a line format describing each cluster. We will describe each step in more detail.

1. *Partition by event size.* The first step of IPLoM applies an event size heuristics based on the assumption that log messages of the same event type have the same event size (number of tokens). The result is a partitioning of log messages into groups of equal length, e.g. *"Connection from 255.255.255.255"* and *"Connection from 0.0.0.0"* both contain 3 tokens and would end up in the same cluster. The cases where events with different event sizes belong to the same cluster are handled in post-processing.

2. *Partition by token position.* The next step assumes that the tokens at the same position in the log message that have the least number of unique words are likely to contain constants. Then, the heuristics works to find the token position with the lowest number of unique values and further splits each cluster based on the unique values at that position.

3. *Partition by searching for bijection.* The final partitioning step is based on searching for bijective (one-to-one) relationships between the set of unique tokens in two token positions. If a bijective relationship is found between two elements in the sets of tokens, log messages will be separated into new partitions at the corresponding token positions. The heuristics works as follows: The number of unique tokens at each token position and the most frequent number of unique tokens greater than 1 are determined. The two token positions with the highest number of tokens are chosen. The idea is that the most frequent token count can indicate the number of event types. For example, in log messages:
   `Command has completed successfully`
   `Command has been aborted`
   `Command failed on starting`,
   first position of the token has one unique token {Command}, the second position has two unique tokens {has, failed}, the third position has three unique tokens {completed, been, on} and the final fourth position also has three unique tokens {successfully, aborted, starting}. Therefore, the tokens at positions three and four, e.g. "completed" and "successfully", are in a $1-1$ bijective relationship with "been" and "aborted", since all lines containing "completed" in position three contain "successfully" in position four and vice versa. There is also a $1-M$ relationship with the tokens "Command", "completed", "been", and "on" because all lines containing "Command" also contain "completed", or "been", or "on" at position three. Another type of relationship is an $M-1$, which works in reverse to the previous scenario. The last type of relationship is $M-M$. This occurs in our example when positions two and three are chosen by the heuristics.

Therefore, another heuristics is implemented to handle the case of $1-M$ and $M-1$ relationships.

4. *Discover message type descriptions (line formats) from each partition.* In this step, the algorithm is terminated and descriptions of clusters or line formats should be discovered.

Fixed-depth tree

Drain [43] uses a parse tree with a fixed-depth structure to represent log messages and then partition them into groups that describe event types efficiently. Drain parses raw log messages individually from streams and does not require an offline preprocessing step, so it works in an online manner.
The structure of a parse tree is shown in 3.3. The node in the top layer is the *root node* of the parse tree and the nodes in the bottom layer are called *leaf node*. Nodes between the first and last layer are called *internal nodes*. Each path in the parse tree ends with a leaf node that stores a list of log groups. The depth of the tree is fixed by a predefined parameter *depth*.

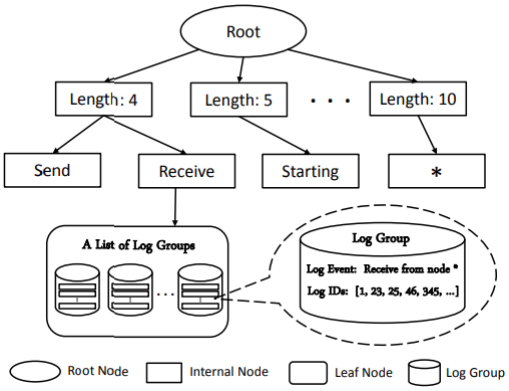


Figure 3.3: Structure of a simple parsing tree of depth 3 in Drain [43].

The algorithm consists of five steps:

1. *Preprocess by Domain Knowledge.* According to the empirical study on log parsing methods by He et al. [42], preprocessing improves parsing accuracy. Thus, when a new raw log message arrives, Drain will preprocess it with regular expressions provided by the user based on domain knowledge before constructing the actual parse tree. This may include frequently used variables, such as IP address, and any matched tokens will be removed.

2. *Search by Log Message Length.* The next step is to build the parsing tree. The process is based on the assumption that log messages with the same log event can have the same log message length. Starting from the root node, each node in the first layer represents log groups whose log messages have different lengths. The message length is specified by the number of tokens. Cases where log messages with the same log event

have different lengths can be handled by post-processing. For example, the path to the preprocessed log message `Packet has been sent` to a first layer node would be *Length: 4*.

3. *Search by Preceding Tokens.* In the third step, the parsing tree is traversed to a leaf node in the second layer. In this step, it is assumed that the first tokens in a log message are likely to be constants. For example, in the log message `Packet has been sent`, the tree is traversed from the first layer node *Length: 4* to the leaf node containing the internal node *Packet*, since "Packet" is considered a constant. Therefore, the second layer contains nodes with unique first words. Depending on the *depth* of the tree, there are more internal nodes which search by the tokens at the first, second, third... etc. position. However, sometimes a variable word may appear in the initial positions, such as `120 bytes received`. This is solved by considering only tokens that do not contain digits. If a token contains a digit, it matches a special internal node *"*"*.

4. *Search by Token Similarity.* After step 3, lists of log groups are contained in leaf nodes. Each log group has *log event* and *log IDs*. Log event is the template that describes the log messages of the group, and log IDs are the log IDs of the log messages in that group. Each log group contains log messages of the same length that start with the same word. In this step, Drain calculates the similarity between the log message and the log event of each log group. The similarity is calculated over each token position. The highest similarity score is compared to a predefined similarity threshold, which indicates whether the given log group is suitable.

5. *Update the Parse Tree.* In the last step, the log ID of the current log message is added to the most suitable log group from step 4 and log event in the log group is updated (different tokens are replaced by wildcards *).

3.2.3 Summary

In this section, we presented an overview of automated log parsing techniques and their representative tools. Table 3.1 provides a summary of all log parsing tools we have reviewed in this paper. The tools are compared from different aspects that we considered important for our use case. **Mode** denotes the online or offline mode, while **Method** denotes the log parsing technique used by the tool. **Preprocessing** describes whether an additional manual preprocessing step is required. **Performance** categorizes tools into three levels based on their efficiency: high, medium, and low, as proposed in [48]. Finally, it is of great importance for practical use that log parsing tools are freely available. Therefore, the last column is devoted to the **Open Source** characteristics of the existing tools.

After understanding and comparing the properties of different log parsing tools described in this section, as well as evaluating these tools in detail on different systems in the paper by Zhu et al. [48], we decided to use Drain. The benchmarking results showed that Drain is superior in terms of accuracy, robustness and efficiency. Moreover, its online nature is of great practical importance for our research, considering that the Motorola SmartConnect system

Log Parsing Tool	Year	Mode	Method	Preprocessing	Performance	Open Source
SLCT	2003	Offline	Frequent Pattern Mining	✗	High	✓
LogSig	2011	Offline	Clustering	✗	Medium	✗
IPLoM	2012	Offline	Iterative Partitioning	✗	High	✗
Drain	2017	Online	Fixed-depth tree	✓	High	✓

Table 3.1: Summary of automated log parsing tools

is constantly subject to change. Also, we often work with smaller time intervals of data, so log lines generated in different time windows may result in a different event template set. Not having to provide the entire log dataset before parsing is an undeniable advantage.

4. Methodology

In order to fulfill the goal of this thesis, it is important to first investigate what steps are necessary to achieve the results. Once these steps are known, different methodological approaches need to be analyzed and compared.

In this chapter, we will first go through a simple illustrative example, with the aim of giving the reader a better overview of the domain and the idea behind getting log messages into more organized structures. This will be followed by a detailed description of the proposed approaches to develop a machine learning based anomaly detection tool.

4.1 Overview of the Proposed Approach

The goal of our approach is to fulfill the objective of the thesis and answer the research questions listed in Chapter 1. Trivially, the last step to be performed is the application of anomaly detection methods. Since our dataset is mostly unlabeled, we need to use unsupervised anomaly detection methods using unlabeled data. However, the main focus of the methodological approach of our research is on the preprocessing steps leading to the final application of machine learning. Figure 4.1 shows the pipeline of the main steps used in this study. Despite the simple structure of the workflow pipeline, in reality the process becomes much more complex and requires several back and forth iterations due to some inconsistencies in the datasets, new insights from observing the data, etc.

Because we are using a real-world dataset, we had to familiarize ourselves with the domain, select specific parts of the log data that are valid for our research, and then investigate which characteristic properties extracted from the log data reflect the underlying events. Since it is important that the data processing is done correctly, it is further divided into three phases.

In the first step, it is necessary to collect the log data from the monitored system. We explore several ways to collect data from the SmartConnect system and explain which ones we choose and why. We show how these logs are parsed and event types are extracted from the raw log messages. We continue with a discussion of feature engineering and how we apply windowing to the time series data. We then explain how we apply the selected machine learning algorithms to the preprocessed data.

Finally, the following four steps of our solution are described in more detail later in this chapter.

1. Data Collection
2. Log Parsing
3. Feature Engineering
4. Anomaly Detection

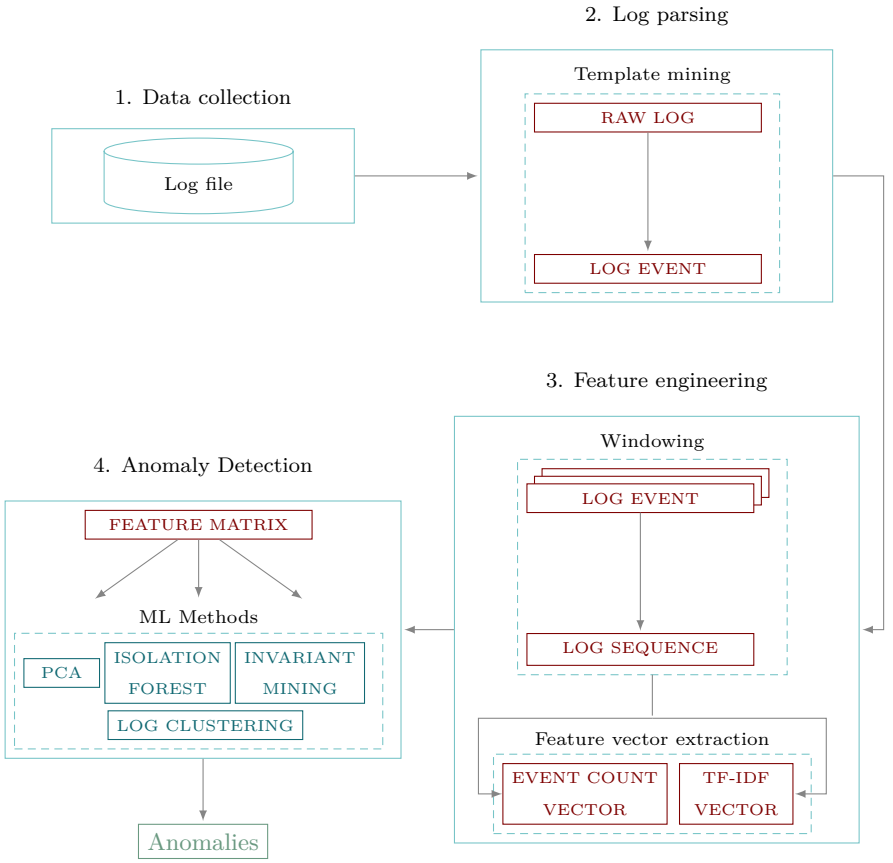


Figure 4.1: The workflow of our anomaly detection solution consisting of four phases - data collection, log parsing, feature engineering and anomaly detection.

4.1.1 Toy Example

To better understand the mathematical modeling and encoding that is required to apply machine learning algorithms to anomaly detection, and also to provide a proof of concept that an anomalous change in logs can indeed be detected, it is nice to have a simple toy scenario. We will simulate a chain of microservices sending traffic from one end to the other.

Let's set m as a number of services denoted s_i, which are connected in a sequential manner: $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow ... \rightarrow s_{m-1}$. Packets are sent from s_1 to s_m.

Over the course of n epochs, we send n packets labeled $p_1, p_2, p_3, ..., p_n$ to s_1.

At each epoch, a service s_i can perform one of the following simple events:

- $\mathbf{E_1}$: Successfully sent a packet (SEND)
- $\mathbf{E_2}$: No packet was sent (FAIL)

For each event, we assign a probability that the event will occur. For events, it holds that

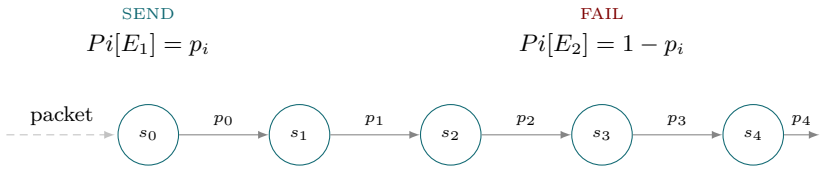


Figure 4.2: An example of a simulation configuration with 5 services, where each service has its own probability distribution of events Pr_i

.

$$Pr[E_1] + Pr[E_2] = 1$$

and each service may have a different event probability distribution.

At each step, each service s_i logs the choice of event for each packet present, denoted as $s_i : p_{j_{E_k}}$. In our simple scenario, the event E_2 : FAIL is an anomaly.

We construct a machine learning model using supervised learning, logistic regression, and decision tree (as opposed to our actual experiments, where we use only unsupervised learning methods) with the goal of finding out which service failed to send a packet. We run the simulation for n epochs. The prediction Y of our model is either the service that failed or no service in case of non-anomalous execution. The training set consists of vectors, where each dimension represents a service and the value in each dimension is the event executed by that service in the given epoch.

First, we created a simulation, whose illustration can be found in Figure 4.2. We gave the simulation the following properties:

- Number of services $m = 5$: $s_0 \rightarrow ... \rightarrow s_4$
- Number of epochs $n = 5000$
- Events E_1 and E_2
- Services s_0, s_1, s_3, s_4 have a 1% probability of event FAIL and service s_2 has a 5% probability of a failure event E_2:

$$\forall i \in \{0, 1, 3, 4\} : Pr_i[E_1] = p_i = 0.99$$
$$i = 2 : Pr_i[E_1] = p_i = 0.95$$

The simulation generates a log file with log messages printed out by the services. Below we see some sample log message entries in the simulated log file. The packets are sent sequentially between the services in ascending order. If service i cannot send the packet to the next service, it logs a message `No packet was sent` and the packet does not reach another service j such that $j > i$. These services also do not record a log message.

```
========== Starting new simulation ==========
Epoch: 0 Service 0: Successfully sent a packet
Epoch: 0 Service 1: Successfully sent a packet
Epoch: 0 Service 2: Successfully sent a packet
```

```
Epoch: 0 Service 3: Successfully sent a packet
Epoch: 0 Service 4: Successfully sent a packet
Epoch: 1 Service 0: Successfully sent a packet
Epoch: 1 Service 1: Successfully sent a packet
Epoch: 1 Service 2: No packet was sent
Epoch: 2 Service 0: Successfully sent a packet
Epoch: 2 Service 1: Successfully sent a packet
Epoch: 2 Service 2: Successfully sent a packet
Epoch: 2 Service 3: Successfully sent a packet
Epoch: 2 Service 4: Successfully sent a packet
Epoch: 3 Service 0: Successfully sent a packet
Epoch: 3 Service 1: Successfully sent a packet
Epoch: 3 Service 2: Successfully sent a packet
Epoch: 3 Service 3: Successfully sent a packet
Epoch: 3 Service 4: Successfully sent a packet
Epoch: 4 Service 0: Successfully sent a packet
Epoch: 4 Service 1: Successfully sent a packet
Epoch: 4 Service 2: Successfully sent a packet
Epoch: 4 Service 3: Successfully sent a packet
Epoch: 4 Service 4: Successfully sent a packet
Epoch: 5 Service 0: Successfully sent a packet
Epoch: 5 Service 1: Successfully sent a packet
Epoch: 5 Service 2: Successfully sent a packet
Epoch: 5 Service 3: Successfully sent a packet
Epoch: 5 Service 4: Successfully sent a packet
Epoch: 6 Service 0: Successfully sent a packet
Epoch: 6 Service 1: Successfully sent a packet
Epoch: 6 Service 2: No packet was sent
```

We can convert the set of log messages into a vector embedding, as seen in Table 4.1. Each row represents an epoch and each cell contains an output of the simulation of a corresponding service, where 1 corresponds to event E_1 and 0 corresponds to event E_2. Figure 4.3 shows the number of services that successfully received and sent a packet for each epoch. As expected from the probability distribution of events, it can be seen that the majority of epochs for all 5 services completed successfully. The service numbered 2 was assigned a higher probability of failure, corresponding to the number 2 of successfully completed services on the y-axis.

The obtained simulated dataset can be easily modified to generate labels. If a line contains only ones, the assigned label is `no_service`. If a line contains a zero, its label is the index of the first service that failed to send a packet. This embedding is then used as input to build a logistic regression and random forest model. The toy example is intended to provide a better understanding of how an unstructured text input can be converted into a format suitable for machine learning algorithms.

Epoch	s_0	s_1	s_2	s_3	s_4
0	1	1	1	1	1
1	1	1	0	0	0
2	1	1	1	1	1
3	1	1	1	1	1
4	1	1	1	1	1
5	1	1	1	1	1
6	1	1	1	1	1

Table 4.1: An example of numerical embedding obtained from the first 7 epochs of the simulation of the toy example.

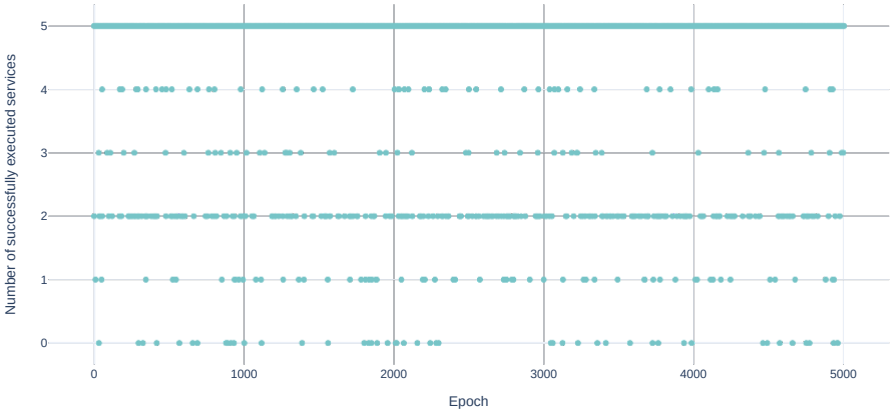


Figure 4.3: The result of running our simulation for 5000 epochs.

4.2 Data Collection

By data collection, we mean the process of collecting log data from the observed system in such a way that the logs can be further parsed, processed, and fed to the machine learning model.

The system we studied follows the microservices architecture pattern as described in Section 2.2 on the architecture of SmartConnect.

4.2.1 Logging

Traditionally, an application performs logging by sending messages to (pseudo) files `stderr` (standard error) and `stdout` (standard output). It can be combined with appending these logs also to a dedicated file with `.log` extension. Both methods are typically used by services running as containerized applications in Pods[1].

[1] `https://kubernetes.io/docs/concepts/cluster-administration/logging/`

In microservices architecture, this method of logging may not be sufficient because an MSA system usually consists of many applications and the logs are scattered in many places. In addition, the individual services are often replaced and the contents of the log files are subsequently removed.

Moreover, for systems that produce a large amount of logs, persisting the logs to files and storing them in the service Pod's file system can cause the node's storage space to be overused. For this reason, techniques such as logrotate[2] are used. Logrotate collects logs, sends them to another location where they can be persisted without affecting the performance of the MSA system, and removes them from the Pod's node. On the other hand, this means that only the most recent (if any) logs can be found in the Pod's `log` file.

SmartConnect MSA takes a similar approach to logrotate. Filebeat[3] from ELK (Elasticsearch, Logstash, Kibana)[4] is used to pass the data collected by the system's Pods to the Elasticsearch engine (introduced in Section 2.2.2). The engine provides an authority for storing logs of the system and is considered as the single source of truth since the log data is persisted there.

4.2.2 Implementation

In summary, the following options are available for log collection in the SmartConnect system:

1. Saving the output of the command `kubectl logs ${POD}` for each relevant Pod
2. Collecting the data stored in `${POD}:/var/log/*.log` for each Pod
3. Requesting logs from the Elasticsearch server from SmartConnect

To gauge the possibilities, the first two items represent the same idea, just executed in a different way - the `logs` command simply prints the output of the desired `.log` file to `stdout`. It is very easy to get the logs of a single Pod, but to gather all the desired information about the whole system, all microservices need to be crawled individually. Moreover, as mentioned earlier, the main drawback of these approaches is that only logs that have not yet been fetched by filebeat are accessible.

Therefore, in order to collect log data from several hours of system operation, services would need to be queried relatively frequently. These methods also introduce complexity into the post-processing of the data once collected, as we cannot guarantee the uniqueness of the entries in these approaches.

On the other hand, the third option is more difficult to automate, but promises a more structured way to obtain the logs. Moreover, given the features of Elasticsearch, it should also provide a faster solution for data collection.

In Motorola, developers review logs stored in their Elasticsearch engine via the Kibana web user interface. The UI makes it easy to visualize and filter log data, but on the down side, its export options are not well suited for a structured way of downloading the data. Regardless of these limitations, a size limit needs to be set for the exports. The limit should not be much larger

[2] `https://linux.die.net/man/8/logrotate`
[3] `https://www.elastic.co/guide/en/beats/filebeat/7.10/filebeat-overview.html`
[4] `https://www.elastic.co/what-is/elk-stack`

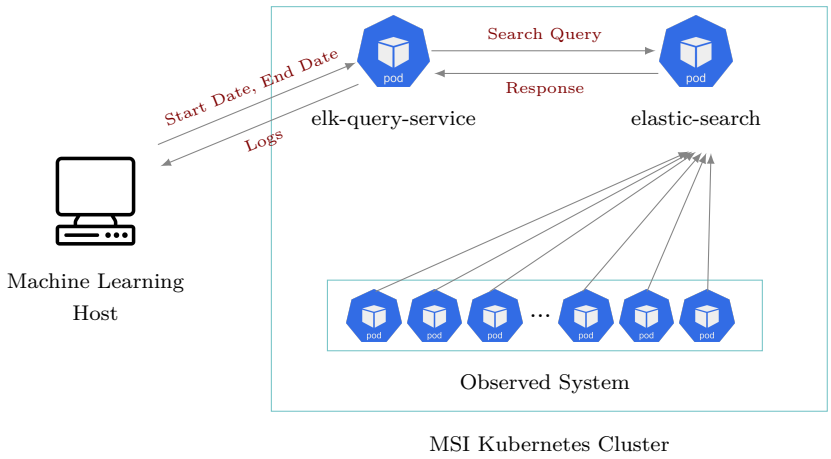


Figure 4.4: The implemented solution for collecting log data. Node with access rights to the Kubernetes cluster calls an orchestration script with two arguments - start date and end date. `elk-query-service` downloads the desired logs that match the date specification. The script stores the response on the machine learning node, which can proceed to the next step - training and evaluating ML models.

than the default 10 MB, as it can potentially have a negative impact on the performance[5]. Based on our requirements, 10 MB of log data would not cover more than a few minutes of logs.

Another option, using ELK stack, would be to make use of the REST APIs of Elasticsearch[6]. However, for security reasons, some companies choose not to expose this API to the Internet - Motorola is one of them. So the only way to access the application programming interface is to make the calls from within the cluster.

We deployed a service called `elk-query-service` within the cluster that handles search queries to the Elasticsearch. Our service provides minimalistic functionality that can be accessed within the company. `elk-query-service` can be invoked with two date arguments indicating the start and end of the time window. Logs between the start and end dates are downloaded to the `elk-query-service` Pod and can be transferred to the node, where they can be further used in a machine learning anomaly detection pipeline, as proposed in Section 8.1.1 for future work. The high-level architecture of our data collection solution is schematically shown in Figure 4.4.

[5] `https://www.elastic.co/guide/en/kibana/current/reporting-settings-kb.html`
[6] `https://www.elastic.co/guide/en/elasticsearch/reference/current/rest-apis.html`

4.3 Log Parsing

This section describes our technique for parsing the raw and unstructured logs obtained from the analyzed system.

As stated in Section 3.2.2, there are many different basic approaches to what techniques should be used when extracting message templates from logs. Therefore, we have developed an additional abstraction layer on top of the actual parser that makes our solution less dependent on which specific template mining algorithm is used.

The experiments therefore work with an object `LogCategorizer` that exposes the functionality to clients. To name a few of the `LogCategorizer` methods, it provides the following, among others:

- `process_log_message(self, log_message: str) -> LogCategory`
- `process_log_messages(self, log_messages: List[str]) -> List[LogCategory]`
- `process_file(self, input_file_name: str)`

The actual reading and creation of templates by these methods is done by an instance of `LogParser`. `LogParser` is an abstraction - an interface used by `LogCategorizer`. Any implementation of the parser must satisfy this contract defined by the interface. The interface defines only the bare minimum of functionality required by a meaningful log template miner, and that is a pair of methods:

- `get_all_templates(self) -> List[str]`
- `process_log_message(self, log_message: str) -> str`

Therefore, an implementation of a log parser should be able to return all templates it has identified up to the call of the `get_all_templates` method. The other method returns a template based on the raw log message entered.

Our implementation follows the adapter design pattern [50], since the goal is to convert interfaces of specific log parsing libraries that may differ drastically in terms of the API they expose.

This allows us to decouple the implementation of the algorithm that identifies templates from logs from our overall goal and opens the way for easy comparison with different template mining strategies.

In Section 3.2.3 we mention that our decision was to proceed with the framework Drain [43], which performs online fixed-depth tree template mining. The current implementation of the algorithm we use is `Drain3` by IBM[7]. The algorithm follows a template of 5 steps as described in more detail in Section 3.2.2:

1. Preprocess by Domain Knowledge
2. Search by Log Message Length

[7] `https://github.com/IBM/Drain3`

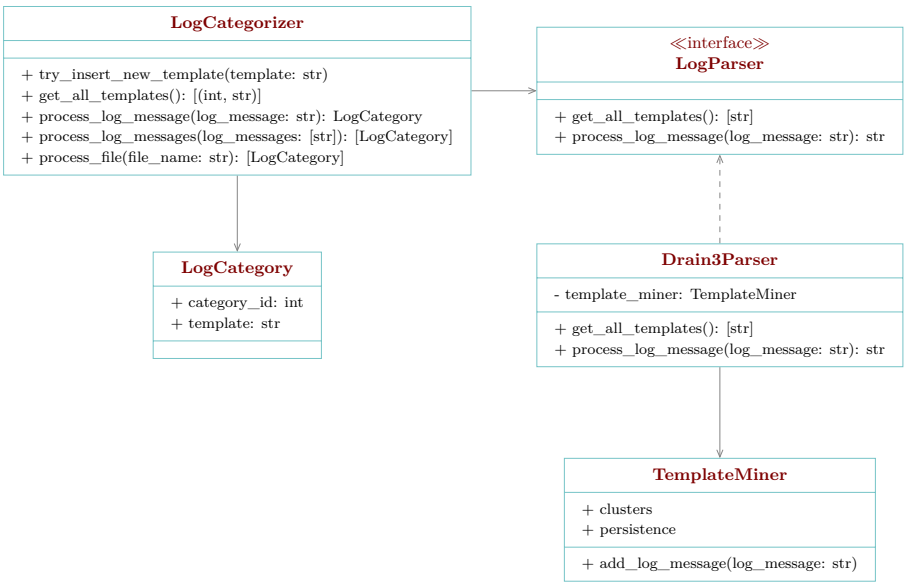


Figure 4.5: The architecture for parsing classes. Application of the adapter design pattern: `LogParser` as adapter, `Drain3Parser` takes the role of concrete adaptee, `TemplateMiner` is adaptee - class defined in another framework with incompatible interface and needs to be converted.

3. Search by Preceding Tokens
4. Search by Token Similarity
5. Update the Parse Tree

Let's pay more attention to the first step. It provides us, the users, with the option to input encoded assumptions and knowledge about the logs that are produced by the system. It is quite intuitive to assume that this domain knowledge, if properly inserted, should improve the performance and precision of mining algorithms. An empirical study [9] confirms this conjecture. However, we argue that this algorithm tuning technique may not be the right choice in all use cases. With our solution, we attempt to analyze a system that is highly fluid and therefore likely to change over time. This, coupled with the fact that many developers from multiple teams and companies (taking into account also external software on which the software system depends) write code that logs in the system, it is not feasible to obtain a single source of truth regarding domain knowledge related to logging. Conversely, we aim for a solution that can learn and adapt to changes as independently as possible.

This goal of adaptability is also reflected in the fact that we ultimately decided to use an online parser. To recall a definition of an online template miner from Section 3.2.1, an online template miner is such a parser that does not need to see the entire dataset to start analyzing message templates. It can perform categorization on the fly while simultaneously updating its internal state, which stores acquired knowledge about the domain.

Drain provides the ability to persist this internal state between multiple runs, giving us the best of both worlds - the parser learns the patterns occurring in the logs over time to improve, while being able to respond to new data in a flexible way.

We did not change steps 2. - 4. as the default parameters [51] produced results that extracted correct templates from the observed logs.

For the last step, updating the parse tree, the Drain3 implementation provides the option to persist the state of the parse tree in the JSON file format. This allows us to reuse the domain knowledge acquired by the template miner across restarts of our solution. This is an extremely useful feature, since we can assume that all logs come from the same system, we can reuse the parse tree already built in previous runs and enrich it. Drain3 provides several ways to save the snapshot of the parse tree, we use the one that saves the state to a file.

4.4 Feature Engineering

Now that we have obtained unique event types from the unstructured log messages, we need to select valuable variables, group them, and encode them into a numeric form that can be further used by machine learning algorithms. This process is called *feature engineering* or *feature extraction*. Feature extraction is the process of generating new features through transformations of the original feature set. In our thesis, we also refer to the generated features as *embeddings*.

The feature engineering step is crucial. The embeddings should be able to summarize valuable information and complex relationships in the dataset, which directly affects the performance of the learning model and the speed of training, and avoids overfitting. The log event types generated in the previous step are used as input to the feature engineering step. The output is a numerical matrix.

4.4.1 Windowing

In time series anomaly detection, a common concept applied in feature extraction is *windowing*. Logs are grouped into smaller blocks called *windows*, where each window represents a sequence of logs. In some previous approaches [8], windowing was applied by grouping events by the same session identifier, so that each window represents a job execution with a unique identifier. This solution is not applicable in our case because we do not have a session or process identifier in our raw dataset.

Log data is also non-uniformly distributed time series data, since there is naturally a time of occurrence of each log. Therefore, another intuitive and commonly used approach to windowing is to generate a set of subsequences of time series. For example, suppose we have log lines ordered by timestamp and the length of the time window is q. Then, log line number 1 to log line number q is enclosed in the same time window 1 and encoded as the first row of the resulting feature matrix. The goal of the next step of anomaly detection is to say for each subsequence whether it contains an anomaly or not. In our work, we use a timestamp based windowing, which is also described in the next part of the text.

A set of time series subsequences can be generated using **sliding windows**. Consider a time series $\mathbf{x} = x_1, x_2, ..., x_n$ of length n. We want to extract subsequences $\mathbf{x_i}$ of length m. In other words, we want to apply sliding windows of length m and move over the time series until we reach the end of the data set. At each move, we shift the window r time steps to the right. As a result, we get p windows, where p is defined as

$$p = \frac{n - m}{r} + 1$$

and the generated windows are of the form

$$\begin{aligned} \mathbf{x_1} &= x_{11}, x_{12}, ..., x_{1m} \\ \mathbf{x_2} &= x_{21}, x_{22}, ..., x_{2m} \\ &\dots \\ \mathbf{x_p} &= x_{p1}, x_{p2}, ..., x_{pm} \end{aligned}$$

The size of the step r is a very important parameter. If one chooses r very small, e.g. $r = 1$, such that the subsequences are obtained by sliding the window one step at a time, one has the advantage that no anomalous sequence can be missed in this approach. However, considering the large dataset that grows with time, computing such partial sequences is inefficient and time consuming. On the contrary, we can choose a high value of r. For example, if we set r to the length of the sliding window m ($r = m$), then there is no overlap between the windows. The advantage of fewer generated subsequences and less processing time comes with a greater risk of missing anomalous subsequences.

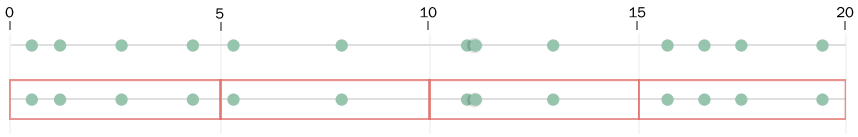
Another parameter to consider is the size of the sliding window m. The ideal size really depends on the application domain and how many consecutive log entries on average can represent an anomaly.

Thus, setting the size of the step parameter r and the length of the sliding window m is a tradeoff between accuracy and processing time. A useful rule is to choose the value of r proportional to the length of the subsequences [52]. Consequently, longer subsequences have a higher value of r, while shorter subsequences have a lower value of r. The number of log lines contained in each window may vary. An example of windowing with different values of m and r parameters is shown in Figure 4.6.

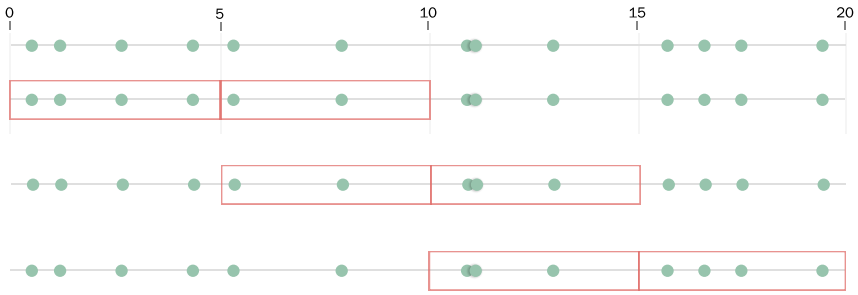
In our work, instead of considering the number of log lines in each window, we consider the size of a subsequence in terms of a time span.

4.4.2 Feature Embedding

After applying the sliding window to generate log sequences, the final step of feature extraction is to construct the feature matrix that can be passed as input to the ML algorithms. In our thesis, we experiment with two different types of embeddings, where both representations describe a set of logs in log sequences: *event count matrix* and *TF-IDF weighted matrix*. Figure 4.7 shows an example of the process of generating a single log sequence into a feature matrix. We assume that logs in sequences are highly correlated. To help the reader understand the differences between the two proposed embeddings, we provide a summary in Table 4.2.



(a) Sliding window with a fixed length of $m = 5$ and step size $r = m = 5$



(b) Sliding window with a fixed length of $m = 10$ and step size $r = 5$

Figure 4.6: Two types of windowing with different step size r. (a) shows sliding windows created without overlap. (b) illustrates sliding windows created by a step size of half the window size, resulting in overlap between subsequent windows.

Matrix type	Row	Column
Event count	log sequence	Frequency of event type in the log sequence
TF-IDF	log sequence	IDF weighted frequency of event type in the log sequence

Table 4.2: A summary of feature matrix types using in our thesis.

Event Count Matrix

A simple approach to obtain such a representation is to create an event count vector for each log sequence. In other words, for each log sequence, we count the number of event type identifiers and store it as a row vector of an event count matrix. The length of the vector corresponds to the number of unique event types detected in the log parsing section, and each component of the vector corresponds to an event type. The value at each position in the vector is the frequency of the event types. For event types that did not occur in the log sequence, the value is zero. Thus, one row of the event count matrix represents a single log sequence. The valuable information that contributes to the machine learning model behind the event count matrix is determining what a normal number of events is and what it means to deviate from the normal

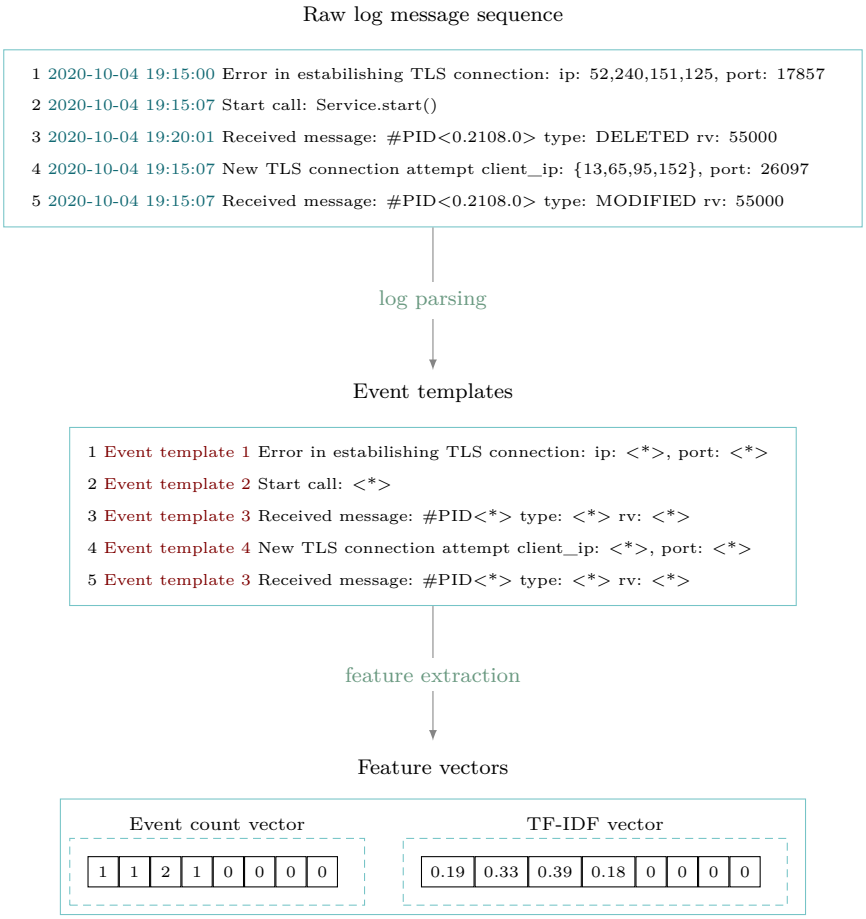


Figure 4.7: An example of encoding log sequence into two types of feature vectors: *Event count vector* and *TF-IDF vector*. We assume that there is 8 event templates in the collection, thus the output vectors are also of length 8.

counts.

This way of constructing the event count matrix is known in the literature as a natural language processing (NLP) model for feature extraction called *Bag-of-Words* (BoW) [53]. BoW treats each log sequence as a *document* and for each *word* (in our case, a word is an event type) in a document, a *weight* is assigned. The weight depends on the number of occurrences of the term in the document. The length of the vector is equal to the number of unique words in the database (analogous to all parsed event types). The weight score is then called *Term Frequency* (TF) and denoted by $tf_{t,d}$, where subscript t refers to the term (event type) and d refers to the document (log sequence) and $f_{t,d}$ is a frequency of term t in document d. The term frequency describes the importance of an event type in a log sequence.

$$tf_{t,d} = \frac{f_{t,d}}{\sum_{t'} f_{t',d}} \tag{4.1}$$

In other words, $tf_{t,d}$ is the probability of t occurring in d.

TF-IDF

Another weighting scheme widely used in Information Retrieval is called *term frequency–inverse document frequency* (TF-IDF).

If we use only the standard term frequency, we have to deal with an important issue. After defining the term frequency, all event types are considered equally important [53]. However, we also want to know how important a term is not only in a document (window), but in the whole dataset. This is where another weighting score, *inverse document frequency* (IDF) comes into play. Let's first define *document frequency* (DF). The document frequency df_t is defined as the number of documents in the collection that contain the term t. Then, the inverse document frequency df_t of the term t is defined as:

$$idf_t = \log \frac{N}{df_t} \tag{4.2}$$

where N is the total number of documents in the collection. It can be observed that if a term is rare in the collection, the idf is high, and if the term is frequent, the idf is low.

By definition, the TF-IDF weight of a term t in document d is a product of two quantities: the term frequency tf_t and the inverse document frequency idf_t:

$$tf\text{-}idf_{t,d} = tf_{t,d} \times idf_t \tag{4.3}$$

The intuition behind the components of TF-IDF weighting is that term frequency gives higher weight to terms that occur frequently in a single document. In contrast, the inverse document frequency gives a lower score to words that occur frequently throughout the collection, since we are more interested in those that occur infrequently. Thus, the weight $tf\text{-}idf_{t,d}$ of the term t in document d is

1. higher, if the term t is occurs frequently in the small number of documents
2. lower, if the term t is rarely occurring in the high number of documents or if the term t is rarely occurring in the document d
3. lowest, if the term t occurs in all documents of the collection

As a result, the feature vector of the log sequence is a vector whose value for each dimension is defined by equation 4.3.

The final vectorization of the log sequence involves normalizing the IDF weight by scaling it between 0 and 1. There are several ways to normalize the IDF, but we chose the common approach of applying the *logistic* (sigmoid) function to normalize IDF:

$$tf\text{-}idf_{t,d} = tf_{t,d} \times \sigma(idf_t)$$
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Log sequence ID	Event Types
l_1	$[1, 4, 1, 3, 7]$
l_2	$[3, 3, 3, 8, 4]$
l_3	$[2, 7]$
l_4	$[3, 3, 6, 8, 4, 6]$

Table 4.3: An example of log sequences that include different sets of log messages. The log message is represented by the event type identifier.

TF-IDF Example

We show an example of computing TF-IDF in our case where we work with log sequences instead of documents and event type identifiers instead of terms. Suppose we have a collection of log sequences and we have 8 event templates extracted from our dataset. Each log message found in the log sequence is identified by the corresponding event type identifier starting from 1 to 8, as shown in Table 4.3.

To compute the term frequency, we find the frequencies of the event types in each log sequence and then normalize the frequencies to sum the row vectors to 1. Continuing with our sample set of log sequences, the TF score calculation is described in Table 4.4.

Log sequence ID \ Event type ID	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
l_1	2	0	1	1	0	0	1	0
l_2	0	0	3	1	0	0	0	1
l_3	0	1	0	0	0	0	1	0
l_4	0	0	2	1	0	2	0	1

(a) Computation of the frequency $f_{t,d}$ of term t in document d. Each value represents the number of occurences of the event type in the respective column in the sequence of the respective row.

Log sequence ID \ Event type ID	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
l_1	2/5	0	1/5	1/5	0	0	1/5	0
l_2	0	0	3/5	1/5	0	0	0	1/5
l_3	0	1/2	0	0	0	0	1/2	0
l_4	0	0	1/3	1/6	0	1/3	0	1/6

(b) Computation of the TF score $tf_{t,d} = \frac{f_{t,d}}{\sum_{t'} f_{t',d}}$.

Table 4.4: Computation of the TF score.

To compute the IDF, we first compute the document frequency df_t of each term and use the formula 4.2 to obtain the inverse document frequencies. For

illustrative purposes, we omit the normalization.

Log sequence ID \ Event type ID	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
l_1	2	0	1	1	0	0	1	0
l_2	0	0	3	1	0	0	0	1
l_3	0	1	0	0	0	0	1	0
l_4	0	0	2	1	0	2	0	1
$\mathbf{n_t}$	1	1	3	3	0	1	2	2

(a) Computation of the document frequency df_t of term t. n_t represents the number of documents (log sequences) that contain an event type represented by the given column.

Log sequence ID \ Event type ID	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
l_1	0.602	0,602	0.125	0.125	0	0.602	0.301	0.301

(b) Calculating the IDF score $idf_t = \log \frac{N}{df_t}$. In our example, $N = 4$ since there are 4 log sequences in the collection. For example, the IDF weight of event type 1 is calculated as $idf_1 = \log \frac{4}{1} = 0,602$.

Table 4.5: Computation of the IDF score.

The final step is to calculate the product $TF \times IDF$ and the values are shown in Table 4.6.

Log sequence ID \ Event type ID	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
l_1	0.2408	0	0.025	0.025	0	0	0.0602	0
l_2	0	0	0.075	0.025	0	0	0	0.0602
l_3	0	0.301	0	0	0	0	0.1501	0
l_4	0	0	0.042	0.021	0	0.201	0	0.05

Table 4.6: TF and IDF scores from the example are multiplied to get TF-IDF.

We assume that by using TF-IDF weighting, we obtain additional information about the event type distribution not only with respect to the log sequences, but also in the whole dataset, which could potentially increase the performance of the ML model.

4.5 Anomaly Detection

Now that we have created the feature matrices, machine learning methods can be applied to detect outliers in the data. For our experiments, we use an open-source machine learning toolkit *Loglizer*.

4.5.1 Loglizer

Loglizer[8] is an open-source machine learning based log analysis toolkit for automated anomaly detection written in Python [9]. Loglizer was developed for automated anomaly detection as a part of LogPAI[9], a collection of AI-based log analysis solutions. These log analysis tools have already been deployed by industrial teams at Microsoft and Huawei. It includes three supervised anomaly detection models (*Logistic Regression, Decision Tree, Support Vector Machine*) and six unsupervised anomaly detection models (*Local Outlier Factor, One-Class SVM, Isolation Forest, Principal Component Analysis, Invariants Mining, Clustering*). At the time of writing, two more unsupervised models were under development (*DeepLog, AutoEncoder*).

We chose to use a third-party toolkit for several reasons. First, since the algorithms we use in our experiments are already implemented, there is no need to completely reinvent the wheel and write the implementation ourselves. This way, we can avoid a time-consuming process and instead focus on the main purpose of our research - to investigate whether applying different anomaly detection techniques to a real-world dataset obtained from SmartConnect system can lead to successful results and how to approach it.

Another advantage is that the Loglizer anomaly detectors work with log sequences. This means that each of the models is trained on a set of log sequences, and the output of the detector classifies whether a single log sequence is an anomaly or not. As described in Section 4.4, we use windowing to generate the log sequences, making the Loglizer toolkit an excellent fit for our research context. Specifically, four methods were used in our research: Invariants Mining, Isolation Forest, and PCA. The underlying algorithms behind these tools are presented in Section 3.1.

The implemented anomaly detection methods are evaluated on a public dataset, HDFS. The logs in the HDFS dataset were collected from the Amazon EC2 platform and contain 11 175 629 entries [8]. Loglizer provides benchmarking results for both supervised and unsupervised methods separately using this dataset, which can be found in Table 4.7. The evaluation metrics are explained in Section 6.2. Benchmarking gives a better idea of the expected performance of the anomaly detection methods we intend to use in our work.

Last but not least, Loglizer makes it easier to reproduce, understand and customize the experiments since the code is open source.

4.5.2 Experiment Workflow

Running machine learning experiments is typically an iterative process, with each iteration requiring different configuration parameters, input training and testing datasets, and a lot of file processing. The same is true for our anomaly detection experiments. We obtained several raw datasets from the SmartConnect system. We selected four anomaly detection methods, with each method requiring a different dataset as well as its own parameter configuration. Each raw dataset must be parsed from its original JSON file into a unified structure, cleaned up, and erroneous log entries removed. Then, a log parser program is applied to obtain an event type identifier for each log message. Finally, we

[8] https://github.com/logpai/loglizer
[9] `http://www.logpai.com`

Model	HDFS Precision	Recall	F1
Supervised Methods			
LR	0.955	0.911	0.933
Decision Tree	0.998	0.998	0.998
SVM	0.959	0.970	0.965
Unsupervised Methods			
LOF	0.967	0.561	0.710
One-Class SVM	0.995	0.222	0.363
Isolation Forest	0.830	0.776	0.802
PCA	0.975	0.635	0.769
Invariants Mining	0.888	0.945	0.915
Clustering	1.000	0.720	0.837

Table 4.7: Benchmarking results of three supervised and six unsupervised anomaly detection methods on HDFS dataset [9]

.

use two different feature matrix representations as input to anomaly detection methods. After we obtain trained models for each dataset, the evaluation follows. With so many dependent processes that need to be performed for each iteration of the experiment, where things can very easily become messy, it is desirable to have an automated workflow system. Although our work is not a software engineering thesis and thus the focus was not on a perfectly extensible workflow, we tried to find a solution that would save the time required to run experiments.

We created a Makefile-based pipeline containing a set of Python scripts to facilitate experiments. The goal was to have a reproducible, modular, and scalable solution to deal with the complexity of running experiments. In this section, we will briefly describe the architecture of this pipeline and the implementation of the experiment scripts.

Makefile

The execution of experiments is orchestrated by the GNU *make* instructions described in the Makefile scripts. A Makefile consists of a set of rules that generate target files when one of their dependencies changes. We chose Makefile because it directly enforces a modular character to our workflow. Every rule must follow the shape:

```
target ... : prerequisities ...
   recipe
   ...
```

Normally, targets and prerequisites are filenames and recipes are commands. A Makefile can be expressed by a directed acyclic dependency graph [54]. The dependency graph of our Makefile is shown in Figure 4.8.

Each target contained in the Makefile operates on the `DATASET` variable assignment. `DATASET` represents the name of the dataset that must exist in the `data/raw/` directory of the project folder before any of the targets are executed, so we call it a *prerequisity* or a *dependency*. The variable can be set from outside the Makefile as part of the command line. Changing the value of `DATASET` directly affects the names of the generated output files. By running `make` in the directory containing the Makefile and assigning `DATASET` variable, we can manage the workflow of the experiment:

- `make` *data* `DATASET=<dataset name>`
  - Parses JSON file with raw data into a panda's DataFrame
  - Flattens columns of the DataFrame that contain nested JSON structures
  - Executes log abstraction by processing log messages, extracting an event type for each log message and appending event type column to the DataFrame
  - Stores DataFrame with preprocessed logs
  - Extracts event count and TF-IDF feature matrices
  - Stores feature matrices
- `make` *train* `DATASET=<dataset name>`
  - Executes training script and generates invariant mining, isolation forest, log clustering and PCA models using event count and TF-IDF feature matrices
  - Stores models
- `make` *evaluate* `DATASET=<dataset name>`
  - Executes evaluation script for invariant mining, isolation forest, log clustering and PCA models trained using event count and TF-IDF feature matrices by detecting anomalies on provided labeled dataset
  - Generates precision, recall and F1 score into CSV files stored in `results/metrics` directory
  - Generates labels predicted by all the models into CSV files stored in `results/predictions` directory
- `make` *all*
  - Runs all modules of the workflow
- `make` *clean* `DATASET=<dataset name>`
  - Cleans up all generated files related to `DATASET`

Models and their configurations that are trained in the pipeline can be specified within the Makefile.

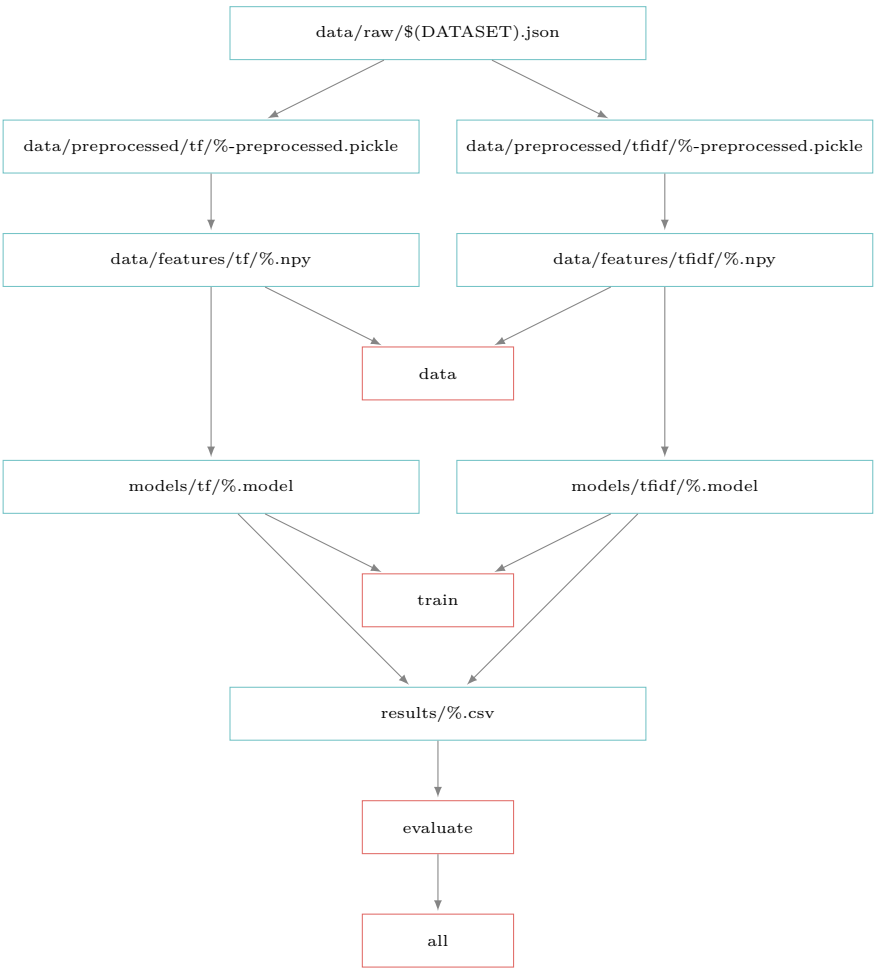


Figure 4.8: A dependency graph for a Makefile to perform experiments in our thesis. Blue rectangles represent *file targets* and red rectangles represent *phony targets*. Phony targets are targets that do not create or update a file, but instead represent a name for a sequence of commands to execute. Phony targets enable the modular execution of our experiments.

Scripts Implementation

A consistent file naming convention and file organization in directories and subdirectories is critical for automated experiments. For an example of the directory structure of our experiments designed for use with Makefile, see the Appendix B.1.2.

Each machine learning method has three Python scripts in the `src/models` directory: *train*, *evaluate*, and the *model* itself.

Each *model* script acts as a wrapper with a common interface that contains an algorithm-providing library (in our case, we use the LogLizer library, as described in Section 4.5.1) and allows calling the following methods:

- `__init__`

- `fit`
- `predict`
- `evaluate`
- `save`
- `load`

The advantage of having a wrapper on top of the actual model is that it can easily be replaced by another library that provides the model implementation, or we can provide our own implementation.

As reader can see from the list above, each model also contains `save` and `load` methods. They are included in the model as a means of serializing and deserializing the trained model. The reason for this is that training a model is usually a very time-consuming task. The ability to load an already trained model saves us the trouble of training the model every time it is needed. A serialized model can be conveniently restored later and used for evaluation or prediction. We use the library `pickle`[10], which converts any Python object into a byte stream. The process of serializing the data is then called "pickling" and the deserialization is called "unpickling".

Train scripts allow optional and model-specific parameters to tune the models. These parameters can be modified in the Makefile. Then, the preprocessed data is simply loaded, the model is instantiated, and the model training process is executed. Finally, the trained model is stored in a pickle.

Evaluate scripts include loading the trained model and executing the `evaluate` method to obtain metrics and predictions.

As shown in the directory structure in the Appendix B.1.2, preprocessed features, trained models, and resulting metrics and predictions are stored in either the `tf` or `tfidf` folder, depending on which feature extraction method the particular file belongs to. `tf`, as an abbreviation for term frequency, is an equivalent to what we call `event count` in our work, as we explained in detail in Section 4.4.2. For brevity, we use `tf` in our file structure.

4.5.3 Mapping from Prediction Back to Log Entry

Once the output is computed, we get a prediction for each window of the input dataset. This is true regardless of the specific machine learning model used for the computation. This type of output, which consists of ones and zeros, is not very helpful if we want to understand why we got these results and whether they are correct. Ideally, we would want more detailed information about each window. Typically, the windows that are identified as anomalous would be of interest, as they are expected to be less frequent and more important. However, there are use cases where we want to examine all windows, especially in the debugging phase.

From a feature vector of a given window, only the ID and the frequency of the event type are given, the template and the order are lost. After discussing with domain experts what would be valuable information about a time window, we concluded that it should be included with every window:

[10] https://docs.python.org/3/library/pickle.html

- *How frequent is each of the event types and what is the template for each event.*
  We create a histogram, ordered in descending order from the most frequent events to the least frequent. This gives a good overview of what the state of the system is in a given time window.

- *Raw trace of logs within the window.*
  The histogram is a good place to start, but in some cases the detail and preserved order matters. In some cases, we also need to know the variable parts of the log messages that are missing from the log templates to get a clear picture of what was going on in the system. Therefore, we also order the traces of the raw logs according to the time in which they occurred.

- *What time range a window corresponds to.*
  It is easy to map from a window index to a human readable time. This way, an expert can tell if something special happened in the system. For example, regular tests may occur daily at certain times, which may explain some behaviour.

5. Dataset

In this chapter we will provide information about the datasets we used in our research. We will describe how we obtained a labeled dataset for training and also mention the weaknesses associated with this dataset. We will take a look at the important properties of the logs. We will also discuss some known anomaly scenarios that can occur in the system logs, and we will go through the dataset in which they are captured. We use the anomalies to validate our models. A validation dataset that contains anomaly-free data will also be described.

All datasets contain log traces collected from multiple microservices comprising the Motorola SmartConnect Cloud Infrastructure Engineering development environments. To obtain the logs, we developed a pipeline (the details of the pipeline are described in Chapter 4.2) that connects to the system, collects logs, and preprocesses them into such a form that the data can be passed to selected machine learning algorithms or plotted. More about the preprocessing step can be found in Chapter 4.

5.1 Datasets

The data we feed machine learning models with makes a big difference. Therefore, we need to look at what data we can collect at a high level from the system and what properties an individual log entry has.

The anomaly detection task can be reformulated as a classification problem with two classes: *positive* (anomaly) class and *negative* (anomaly-free) class.

Negative or *normal* class is a category of a dataset that is free of anomalies and represents the expected (or normal) behaviour of a system that produces these logs. Negative data samples are labeled as 0. The class *Positive*, *anomalous*, or *abnormal* is the target class we are interested in. It is assigned to log sequences that contain anomalies as 1. The goal is to train the classifier to distinguish between positive and negative samples.

Motorola Solution's policy for storing historical data is that we have logs available at any point in time from 30 days ago to this point in time. The stored interval may at times become even smaller if the amount of persisted logs exceeds a limit.

In Motorola, the development environment is further divided into two subsections that we have access to. Let's call them *experimental* and *stable*.

The *stable* environment is synchronized with all the master branches of the source code. It is used to perform extensive testing of the system every night.

On the other hand, the *experimental* environment is used by the developers to deploy and explore side (or feature) branches during the day.

For confidentiality reasons, we do not have logs of the *production* environment available. This is one of the major drawbacks, as it can be assumed that they contain different traffic. It can also be assumed that the distribution of anomalies in *production* environment is different from both *experimental* (more anomalies, some related to ad hoc deployment) and *stable* (fewer anomalies, less traffic). In general, the development environment produces huge amounts of data, thousands of log messages per minute.

Next, we provide a detailed description of the characteristics of each dataset and how they were obtained. See Table 5.1 for a summary of all datasets used in this work.

5.1.1 Dataset Daily

This dataset reflects the *experimental* region of the development environment. In such an environment, it is natural (or even desirable) for anomalies to occur.

We refer to the dataset of system logs produced by the region as the *Daily* dataset, and may also refer to it as *mixed*, since it will contain both positive and negative data samples. The system is active during a working day when developers are testing new features. The amount of logs varies greatly depending on the team's tasks. However, we are talking about tens of thousands of log messages per hour, even when the system is idle.

Unfortunately, such a dataset is not labeled, as experts would have to go through millions of lines of unstructured log messages to obtain the labels.

5.1.2 Dataset Nightly

At night, the system we are experimenting with goes through tests in the *stable* region. This phase of health check of the system involves mimicking the normal behaviour of the telecommunications system. All the necessary infrastructure setup takes place and simulations of calls between groups of push-to-talk radios are performed. These tests are run again each night as part of the nightly test suite. Typically, the test pipeline lasts about 7 hours and produces about 2 million log messages.

If the nightly tests all pass, the data collected should either be completely free of anomalies, or contain only a very small percentage of them. Therefore, we assume that the data collected during the nightly testing phase, if passed successfully, will contain a large majority of negative data samples and represent normal anomaly-free behaviour of the system. This statement relies heavily on the assumption that the functionality of the system is well covered by the tests (the relationship of our approach to anomaly detection and test code coverage is discussed in detail in Section 8.1.2).

Therefore, for our use case, we can also interchangeably refer to the dataset *Nightly* as *negative* or *normal* dataset.

Therefore, the logs collected at night by such a system should be suitable candidates for one-class classification algorithms as well as generate a labeled dataset for testing purposes, which we will explain later in this chapter.

However, the data generated by the system during nightly testing is slightly different from the live production environment. The main difference is that the scenarios generated by the test suite are serialized, meaning that there is at most one call at any given time.

On the other hand, in the live production environment, there is no limit to the number of radio broadcasts at a time. Therefore, the logs of the production environment can contain much higher counts of some events.

Although this could be argued as a weakness of our nightly dataset, as described windowing operation of our preprocessing phase in Chapter 4, we will always consider a sequence of logs and the number of event types in that log sequence in a fixed time window. Therefore, we believe that the final feature

representation passed to the ML algorithm should be very similar, even though the data will look different with respect to individual log entries.

Since it is incredibly difficult to obtain labels in log data, we decided to use the Nightly data, despite its potential drawbacks and slightly artificial nature.

The Nightly data contains logs from the night of January 24, 2021.

5.1.3 Dataset Nightly Test

This dataset was collected in the same way as the Nightly and should meet all the properties, but it was recorded on a different day. Therefore, the timing is different and the tests are examining a different version of the SmartConnect system source code.

We use Nightly Test mainly to show that a similar distribution of arbitrary records collected during any nightly testing can be assumed.

Nightly Test contains logs from the night of January 22, 2021.

5.1.4 Dataset Anomalies

This dataset contains versions of the two anomaly scenarios described later in Section 5.4. The anomalies were achieved by killing Kubernetes Pods, which is responsible for the correct operation of a particular service type. At the same time, users were trying to establish, or were in a call.

This dataset captures a total of 23 instances of anomaly scenarios and was created by merging logs from two days - January 25 and 28, 2021.

5.1.5 Dataset Glostrup Calling

In contrast, the Glostrup Calling dataset consists of anomaly-free calls between Motorola Solutions two-way radios. The *experimental* environment was set so that nothing else interferes with the calls. Therefore, the resulting logs can be considered as *normal* data.

The trace records several calls that were carried out during 8 minutes on February 3, 2021.

5.2 Dataset Test with Labels

Unsupervised learning does not require data to be labeled to detect anomalies, as it assumes that the majority of data points are normal. This is very beneficial in our situation where it is extremely difficult to obtain labeled data. However, the lack of labels poses a difficulty when it comes to the model evaluation phase.

Unlike supervised learning, where one can directly compare predictions with actual labels to assess the quality of the machine learning model, there is no way to do this in unsupervised learning.

In our work, the majority of the data is unlabeled data. To validate the proposed approaches, we had to develop the test dataset ourselves.

To get a better idea of what kind of data constitutes a manually created test dataset, we provide a pie chart of the test set composition in Figure 5.1. It is important to note that the number 131 represents the amount of extracted

log sequences (windows), so the number of underlying log message samples is much higher.

Determining the negative class representation is trivial. As we mentioned earlier in this chapter, we consider the Nightly data from the testing phase as normal data samples. To create the normal part of the dataset, we collect another set of data from the testing phase that was previously unseen in the training phase.

On the other hand, positive samples must be manually labeled. To avoid label assignment errors, we introduce some known anomaly scenarios that are expected to occur in our real dataset. These scenarios can be reproduced, we can collect the data and label them as anomalies.

This is not an ideal test dataset, but not all anomalies are known and can be sourced. It provides a reasonable tool to measure how well our models are performing. We will also test the models with only normal data to detect false positives.

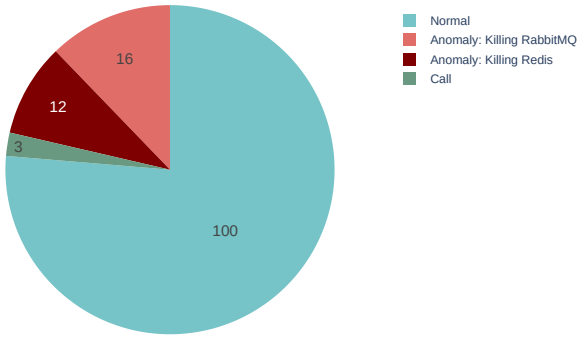


Figure 5.1: Testing dataset composition. A manually collected and labeled dataset consists of 131 log sequences, where each log sequence was collected within a two-minute time window. The value in each section of the pie chart indicates the number of log sequences of each type of log message it contains. The "Normal" and "Call" sections of the pie chart both represent normal, anomaly-free behaviour, while " Killing RabbitMQ" and "Killing Redis" are anomalies.

5.3 Log Properties

The services record 33 properties per single log entry. Of the 33 log properties, we chose to consider only `msg` and `timestamp`, as the others (such as the name of the Kubernetes pod, the process ID of the service generating the log, etc.) are too specific. For the full list of the individual log properties see Appendix A.1. Including fewer predictor variables should generally avoid overfitting, make the

Dataset Name Time Span	Log Entries	Size	Environment
Daily 26/1/2021 6:00 - 26/1/2021 21:00	539 372	2.84 GB	Experimental
Nightly 24/1/2021 21:00 - 25/1/2021 3:50	1 362 004	4.24 GB	Stable
Nightly Test 22/1/2021 21:00 - 23/1/2021 3:43	1 373 919	4.47 GB	Stable
Anomalies 25/1/2021 15:40 - 25/1/2021 17:00, 28/1/2021 15:25 - 28/1/2021 16:00	805 614	3.6 GB	Experimental
Glostrup Calling 3/2/2021 9:40 - 3/2/2021 9:47	1904	15.82 MB	Experimental

Table 5.1: Summary of all the log datasets we worked with in our research on anomaly detection.

final model more transparent, and make debugging easier. Therefore, to obtain as general a solution as possible, we restrict ourselves to these two predictors.

5.3.1 Format of Log Properties

Log messages (property `msg`) are strings and we assume that they follow the logic we described in Section 3.2.1 on log template mining. In other words, we assume that they are generated by code in such a way that they can be thought of as a product of constant and variable parts. Consequently, log messages can be further categorized using event types.

Timestamps (property `timestamp`) have the format of:

```
YYYY-MM-DD'T'HH:mm:SS.sss'Z'
```

The format string indicates that the year (`YYYY`) is a four-digit number padded with zeros, the month (`MM`), the day (`DD`), the hour (`HH`), the minutes (`mm`) and the seconds (`SS`) are two-digit zero padded numbers and the milliseconds (`sss`) is a three-digit number filled with zeros. The time part is separated from the date by a single `T` character and the entire timestamp ends with a letter `Z`.

5.4 Types of Anomalies

In this section, we present a set of known anomalies that we will trace in order to test our solution. We will benchmark our approaches and models against these known anomalies, which can be easily simulated and reproduced. However, the goal is to also detect anomalies that we are not aware of. In other words, we also want to identify anomalous scenarios that appear for the first time in the production environment and do not appear in our training and testing dataset.

5.4.1 Cache Outage

In this anomaly, a cache storage that orchestrates the call logic is taken down and the radios that are in a call at that moment bonk because they cannot communicate with each other. In response, the storage service automatically

restores itself, usually in no more than 2 seconds. After that, the broadcast should continue.

As mentioned in Section 2.2.4 about caching in Motorola SmartConnect, Redis is used for caching, so we can also call this anomaly *Killing Redis* or similar.

5.4.2 Message Broker Out of Service

Similarly, an important entity in the SmartConnect microservices system is the message broker that enables inter-service communication. In this scenario, all brokers are terminated, therefore the microservices network is disrupted and messages cannot be forwarded until the brokers are restored. After such disruption, it normally takes over one minute to restore the operation.

Section 2.2.3 discusses in more detail how messages are forwarded in SmartConnect's microservices architecture. RabbitMQ is the specific message broker that is used, so we will refer to this anomaly in the text also as *Killing RabbitMQ*.

6. Experiments

In this chapter, we will look at the configuration of the experiments, which has a direct impact on the results of our four different approaches to anomaly detection. Before running the experiments, we will also look at the data visually to verify the assumptions we make about the data. We then present the experimental setup, including the datasets, the software and hardware used for the experiments, and the evaluation metrics. Finally, we present the definitions of the tunable hyperparameters and give an overview of the hyperparameter values that performed best and will be used for the final experiments.

6.1 Exploration of Assumptions

To gain better insight into the dataset, it is useful to perform an initial Exploratory Data Analysis (EDA) [55] of our dataset. Exploratory data analysis is used to analyze the underlying structure of the data and summarize the main features, often via visualization methods. Visualization helps us to test our assumptions and formulate hypotheses about the data, which are important to choose the right approaches and parameters for anomaly detection. In addition, visualization of the logs helps us answer **RQ1** (Research Question 1).

For visualization, we use the numerical representation of our dataset obtained by feature extraction. All visualizations in the following sections were obtained on the event count embedding of the data. The embedding of the log sequences contains 3 963 variables. Data with such high dimensions are difficult to interpret. Therefore, it is necessary to use dimensionality reduction techniques. To perform initial exploration as well as visualize the anomaly detection results, we will use two algorithms, PCA and t-SNE, to reduce the dimensionality for data visualization:

- **Principal Component Analysis (PCA)**
  One way to reduce the dimensionality of data without losing too much information is to use Principal Component Analysis (PCA) as an unsupervised tool for exploratory data analysis. In PCA, the data points (in our case, the feature vectors) in our 3 963-dimensional feature space are transformed to a 2-dimensional feature subspace. Moreover, the variables in the new artificial subspace (principal components) are not correlated. The first principal component (PC1) propagates in the x-axis direction and explains most of the variance in the data.

- **t-distributed Stochastic Neighbor Embedding (t-SNE)**
  t-SNE is a recent algorithm for dimensionality reduction into a two or three dimensional space that is particularly well suited for data visualization [56]. PCA is a linear method that keeps the dissimilar data points far apart in the low-dimensional projection. In contrast to PCA, t-SNE is a non-linear method. It focuses on preserving the local structure by keeping the high-dimensional data points that lie on or near a nonlinear manifold in the lower dimensions close together. Preserving local structure while revealing important global structure is the main advantage over linear techniques, where this is usually not possible.

The four assumptions we will explore with respect to the collected data are the following:

- **Assumption 1:** Daily dataset deviates too much from the normal behaviour represented by Nightly to be used for training purposes
- **Assumption 2:** Using data from nightly tests collected in a single night for training is sufficient because the data from all nightly tests are equivalent (following the same distribution)
- **Assumption 3:** Logs generated from calls between two radios are grouped into a call cluster when plotted
- **Assumption 4:** Different anomaly logs are grouped into distinct and clearly separated clusters when plotted

We will describe each assumption in detail in the next sections.

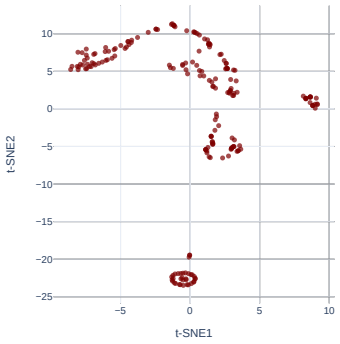
6.1.1 Choosing Data for Training: Daily vs Nightly

As we explained in the earlier sections of this chapter, we train the models on the Nightly dataset under the assumption that its data points are normal. On the other hand, we have no prior assumptions about the Daily dataset and about the distributions in it. Therefore, we want to make a simple visual comparison between the Daily and Nightly datasets. Next, we wanted to determine if the Daily data points fall within the normal range.

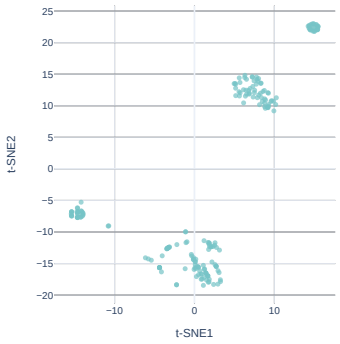
Figure 6.1 shows two visualizations of applying t-SNE to the Daily and Nightly datasets. The t-SNE analysis of the Daily data appears to split the data into four visual clusters. Although it is difficult even for domain experts to interpret what each cluster represents, we assume that the developers tested four different features on the day they obtained the Daily dataset, resulting in the groupings of data points in each cluster.

The visualization of the t-SNE in the plot of the Nightly dataset shows that most of the data points are packed together and form a cluster. However, there is a group of points at the bottom of the plot that appears to be quite distant from the rest of the points and could be considered an outlier. Looking more closely at the raw logs behind the data in this cluster, we found that they contain a high number (about 400 hundred logs every two minutes) of log messages informing about a call, as shown in the following event templates:

- `Call {1586,1} {:call_number, <*> <*> forwarding audio RtpPacket{ssrc: <*> urid: <*> timestamp: <*> seq_num: <*>`
- `Call {1586,1} {:call_number, <*> <*> received RtpPacket{ssrc: <*> urid: <*> timestamp: <*> seq_num: <*> from {228, 28, <*> <*>`
- `Call {1586,1} {:call_number, <*> <*> forward RtpPacket{ssrc: <*> urid: <*> timestamp: <*> seq_num: <*> LMR: [ssrc: <*> urid: <*> to {228, 28, <*> <*>`



(a) Nightly Dataset



(b) Daily Dataset

Figure 6.1: Application of t-SNE to the event count vector embeddings of the Nightly and Daily datasets.

This is further confirmed by visualizing the Glostrup Calling dataset in respect to the Nightly dataset in Assumption 6.1.3.

Since calls are not frequent during testing phase, compared to other processes running in the system, this is exactly what we expect to see reflected in the logs. Nonetheless, as shown in the plot, the cluster is considerably distant from the rest of the points, and we do not want regular calls to be detected as anomalies. It is important to note here that the distances between clusters in t-SNE do not necessarily mean anything. Fine-tuning the *perplexity* hyperparameter of t-SNE is required to obtain clusters that maintain global geometry. Thus, we do not know where the assumed call cluster lies with respect to the rest of the data points.

Finally, we are interested in seeing how PCA compares to t-SNE, and we also want to plot the Daily and Nightly datasets relative to each other. Figure 6.2 shows a plot of the first two principal components after performing principal component analysis on the Nightly and Daily datasets. Figure 6.3 is a t-SNE plot visualizing the Nightly and Daily datasets.

The points produced by PCA appear to be much denser compared to t-SNE, but there is very little overlap between the Daily and Nightly data in either plot. One way to explain this is that the tests performed at night do not fully cover the behaviour of the system. Another explanation, confirmed by the developers at Motorola as the most likely, is that the experiments conducted during the day vary each day depending on the feature being tested, and it may also behave completely opposite to what we consider normal behaviour. Nonetheless, these visualizations confirmed that the Daily dataset should not be used for training the models because it is unpredictable and inconsistent. It is still a good candidate for performing expert validation on an unlabeled dataset, where an expert's understanding of the underlying processes will help us understand the anomaly detection models.

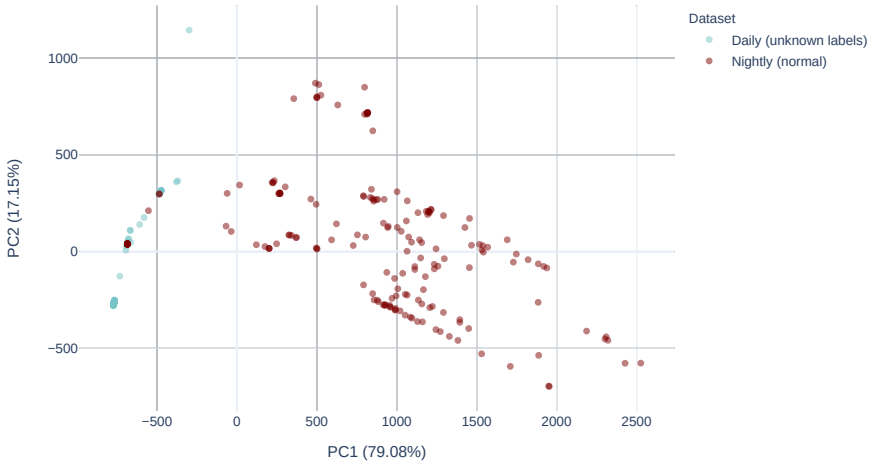


Figure 6.2: Application of PCA to the Daily and Nightly dataset

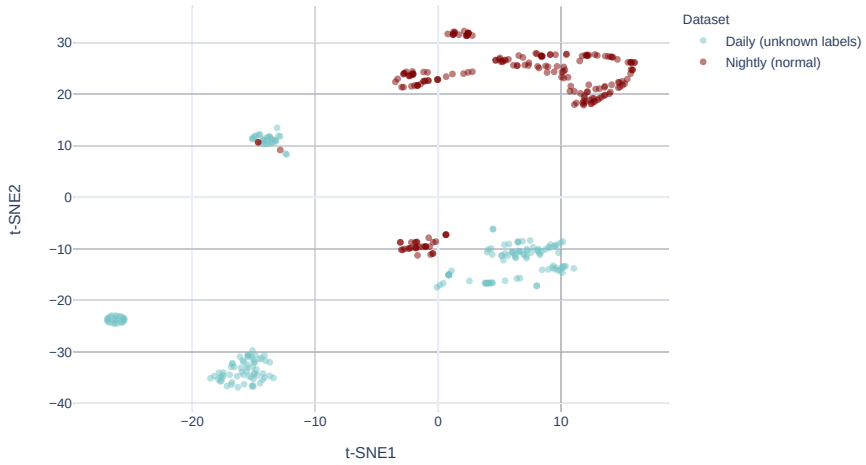
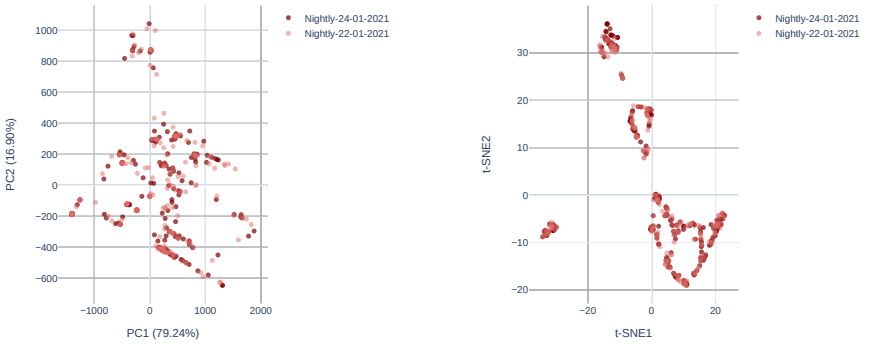


Figure 6.3: Application of t-SNE to the Daily and Nightly dataset

6.1.2 Completeness of the Nightly Dataset

Another important questions we would like to answers by conducting EDA concern the completeness of the Nightly dataset from January 24, 2021, which we would like to use for training the one-class model: *Do the data from the nightly tests differ across dates? Would merging data from multiple nightly test datasets provide additional information about the normal behaviour of the system?*



(a) PCA (b) t-SNE

Figure 6.4: Comparing data from nightly testing obtained from two different dates. The plot on the left (a) represents a PCA plot of the two datasets, whereas (b) is a t-SNE plot.

To analyze the data and answer these questions, we will again use PCA and t-SNE and plot the data from two different nights of testing. Figure 6.4 shows the visualization of the results. It can be seen that the two data sets overlap almost completely in both the PCA and t-SNE plots. Given the considerable overlap, we can safely assume that the logs collected during the nightly tests are equivalent within the different dates.

Through our domain knowledge, we know that the code will differ slightly among the tests as the Motorola SmartConnect software product is under development. The reason for this is that the master branches of the services are being tested and we do not expect them to change dramatically from day to day. However, it is something that needs to be considered when deploying our solution. It must be ensured that the code whose produced logs are analyzed by our anomaly detection algorithm is also the version of the code that runs the tests.

We conclude that using data from only one night is sufficient for training the anomaly detection models, and adding more logs from different nights would not add any additional information to the data. In the longer term, however, the software will deviate from the current state. Therefore, in the future, a proper automated machine learning pipeline that includes retraining models needs to be developed as discussed in Section 8.1.1.

6.1.3 Calls Group Into A Cluster

In the t-SNE of the Nightly dataset, we could see a cluster of points that we assumed were calls between two radios. The assumption stemmed from the fact that we selected several random data points (log sequences) from this cluster and manually checked the histogram of log templates within the log sequences. It turned out that the vast majority of them represented calls. We will now prove that the observed clustering of points is a cluster of call logs and not an unknown anomaly. We will prove this assumption by highlighting the purpose-

built dataset Glostrup Calling in the t-SNE embedding of the Nightly dataset as a central clue to where the call data points cluster.

Figure 6.5 shows the execution of t-SNE on the scenario described above. After applying feature extraction in 2 minute windows on the simulated 8 minutes of calls, we obtain 4 data points (highlighted by green color). The visualization of the plot indicates that the call data points fall very close to the observed cluster and, moreover, all call data points are consistently close to each other.

Therefore, we believe that it is possible to determine the class of a clustered set of data points even without using the actual labels. This confirms our assumption that the data from the Nightly dataset is free of anomalies and that the observed cluster is indeed a cluster of calls.

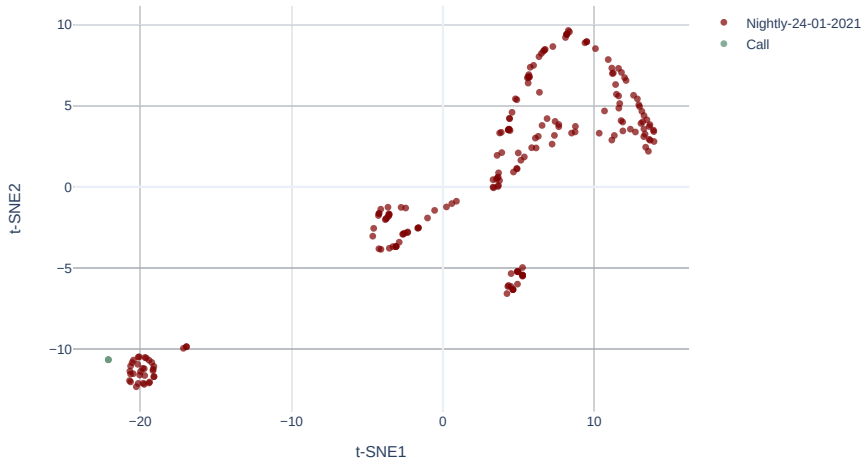


Figure 6.5: Application of t-SNE to the Nightly and Glostrup Calling dataset.

6.1.4 Anomalies Are Separable from Normal Data

The last assumption we want to check before using the data to train the actual machine learning models, and by far the most important, is to see if anomalies have a tendency to cluster in different regions than normal data. This is crucial because if the human eye can detect an anomaly by looking at a plot, then we can assume that anomaly detection algorithms will also be able to detect these anomalies.

To prove this, we will plot our Anomalies dataset generated by collecting anomalies during the simulation of the anomaly scenarios described in Section 5.4. In the PCA and t-SNE plots in Figure 6.6 and 6.7, the anomaly of Killing Redis is represented by green color and Killing RabbitMQ is represented by blue color. Anomalous data points are plotted with respect to normal points, which are highlighted in red.

Both plots give us a clear picture that the data points are divided into different clouds of the same anomaly types. Moreover, the anomalous points are outside the normal range. We consider the fact that different anomalies can be successfully distinguished as clusters in the embedding space as an answer to the **RQ1**. In this case, it is clear that t-SNE gives us a better clustering result than PCA with less points overlapping each other, although both techniques are able to separate the data points into different classes. However, it is interesting to see in the PCA plot, that the Killing Redis anomaly is much closer to the cluster of normal points than Killing RabbitMQ (as mentioned earlier, we should not consider the distances between clusters in t-SNE graphs unless we tune its parameters, so we focus on the PCA graph for cluster proximity analysis). This can be explained as a failure of all brokers in the distributed architecture of the system, which relies on message passing between microservices, causing a major disruption to the entire infrastructure. While the malfunctioning Redis cache is an issue that can be fixed relatively easily, the latter causes a severe problem in the application that propagates to many places. Also, as we observed, the broken broker scenario causes the system to be down for a much longer period of time, which gives the services a longer window to generate a larger amount of logs indicating problems.

We obtained a solid indication that feature embedding is informative enough to identify anomaly clusters. We will attempt to further prove this assumption using machine learning.

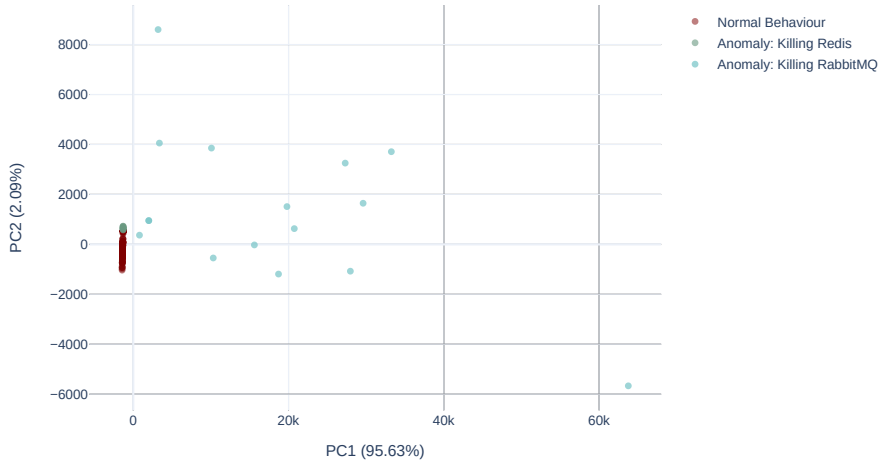


Figure 6.6: Application of PCA to the anomaly data.

6.2 Evaluation Metrics

Evaluation metrics are needed to adequately compare different anomaly detection algorithms and different experiment settings.

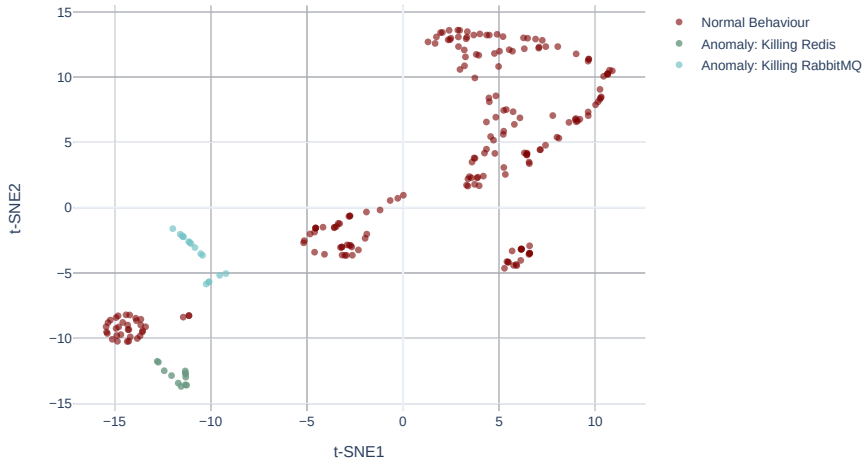


Figure 6.7: Application of t-SNE to the anomaly data.

Let us first introduce four basic metrics: true positive (TP), false positive (FP), true negative (TN), and false negative (FN).

To give an example, let us consider an experimental setting: In a classification task, for each data sample, we assigned a binary label l with values in a set of classes. It is important to note that a positive class is of particular interest for which the evaluation measure is valid. It gives an information about the correctness of the example in the task of anomaly detection. Moreover, an anomaly detector makes a prediction $\hat{l}$, assigning for each data example whether it classifies it as an anomaly or as a normal data instance. The pairwise relationship of the four counts is explained by the confusion matrix in Table 6.1. The confusion matrix is a form of contingency table that provides an interpretation of the classifier's performance on a labeled dataset. It is a 2×2 matrix (in the case of only two classes, but can easily be extended for multiple classes), where the rows represent the actual value of a variable, while the columns represent the predicted value of a variable.

		Predicted $\hat{l}$	
		Anomaly	Normal
Actual l	Anomaly	TP	FN
	Normal	**FP**	TN

Table 6.1: An example of a confusion matrix for binary classification.

Many meaningful measures can be extracted from the confusion matrix. The evaluation measures are computed on the positive class that is assumed to represent the presence of an anomaly.

Accuracy is the most general metric for measuring performance. It simply measures the proportion of correct predictions from all available data points.

Accuracy in binary classification is calculated as shown in the following equation:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{6.1}$$

However, accuracy alone does not exactly reflect the actual performance of a system designed to correctly detect anomalies. It does not distinguish between the number of correct predictions of different classes [57]. The reason why the accuracy measure is not sufficient in the majority of ML applications is explained later in this section.

The most commonly used metrics for performance evaluation in machine learning are *precision, recall* and $F_\beta - measure$. They measure the correct prediction of anomalies within different classes. In our research, we will focus mainly on these three performance measures.

In order to use them, the problem for which we want to estimate performance must be a classification problem. We assume that our dataset represents a normal behaviour of the system, so that an anomaly detection problem can be translated into a one-class classification problem.

Precision, recall, and F_β-measure are based on the ratios of counts TP, FP, TN, and FN.

Precision represents the proportion of correctly detected anomalies out of the total number of anomalies reported. Recall measures the proportion of correctly detected anomalies out of the total number of anomalous data points in the dataset. Finally, the F_β-measure combines these two measures into a single measure. The F_β-measure represents the harmonic mean of precision and recall. If precision and recall are evenly balanced, then $\beta = 1$ and we call it F1-measure. When $\beta > 1$, the score is in favor of precision. On the other hand, it is in favor of recall if $\beta < 1$.

Let's look at the mathematical formulas for calculating the evaluation measures:

$$Precision = \frac{TP}{TP + FP} \tag{6.2}$$

$$Recall = \frac{TP}{TP + FN} \tag{6.3}$$

$$F_\beta - measure = (\beta^2 + 1) \cdot \frac{Precision \times Recall}{\beta^2 \cdot Precision + Recall} \tag{6.4}$$

The reason for the F_β score is that improving either precision or recall is a trivial task in itself, but the goal is to optimize both simultaneously. The precision metric is more useful when true positives and true negatives are more important. However, false positives and false negatives are considered crucial in most classification problems. Therefore, in these cases, the F_β score is a better evaluation metric. False negatives and false positives are given more weight, while a large number of true negatives may not affect the score.

Moreover, since anomalies occur much less frequently than normal data samples, we can easily achieve high accuracy without detecting anomalies. For this reason, even though we will be stating the accuracy metric in the result, we will not use accuracy metrics for actual evaluation in this paper.

6.3 Experimental Setup

In Chapter 3, we described four anomaly detection algorithms: Isolation Forest (3.1.3.1), PCA (3.1.3.2), Invariant Mining (3.1.3.3) and Log Clustering (3.1.3.4). These algorithms are used to perform the unsupervised anomaly detection experiments. In this section, we will describe the data used in these experiments and discuss the choice of different parameters for each of the anomaly detection methods. To tune the parameters, we will use the evaluation metrics described in the previous section.

6.3.1 Dataset

The dataset used for training the models is the Nightly dataset with data instances of the normal class, hence we also refer to this type of learning as one-class learning. Before training, we start preprocessing the data into numerical vector embeddings in all our experiments indistinguishably according to the methods described in Chapter 4. In Table 6.2 we give an overview of the statistics related to the dataset used for training and testing.

We used the sliding window technique to divide the dataset into log sequences. The number of resulting sequences depends on the size of the window and a step size. The window size should be chosen to contain the maximum amount of information. We have found that the anomalies we observe in the experiments span at most two minutes. Therefore, we set the window size to two minutes and the step size also to two minutes so that no two windows overlap. As a result, we obtained 201 log sequences extracted from the Nightly dataset. Each log sequence can be one of two different vector representations: *event count* vectors and *TF-IDF*-weighted vectors. Therefore, each experiment is run twice with both embedding types.

The length of a vector is given by the number of unique event templates extracted during preprocessing of all datasets used in our research and stored in the state of our log template miner. Thus, at the time of model training, both the number of event types and the length of each log sequence is equal to 3 963.

Dataset	Window size	# Log sequences	# Event types	Ratio of anomalies
Nightly	2 minutes	201	3693	0
Testing	2 minutes	131	3693	0.22

Table 6.2: Summary of statistics from the Nightly dataset used for training and the Testing dataset used for model evaluation and hyperparameter tuning.

6.3.2 Software

To develop scripts for data preprocessing and to run our experiments, Python 3.6.9 was used as the primary programming language. In addition, we used the following Python libraries:

- pandas 1.1.3
- numpy 1.19.2

- torch 1.7.1
- plotly 4.12.0
- matplotlib 3.3.2
- jupyterlab 2.2.8

Exploratory analysis, plotting of data, and preparation of scripts for the ML pipeline are performed using Jupyter Notebook with a Jupyter Lab file browsing interface. Jupyter Notebook is a web-based interactive computational environment for writing Python code that works in a web-based environment. The biggest advantage of using Jupyter Notebook instead of writing Python scripts directly is the ability to run inline code and see the result directly below. It is also possible to add formatted text that improves readability and the reasoning behind the analysis steps.

6.3.3 Hardware

All experiments were performed on a HP ZBook computer with the following technical specification:

- Operating System: Linux Mint 19.3 Cinnamon
- CPU: Intel Core i7-8850H
- RAM Size: 32 GB
- CPU Frequency: 2.60 GHz
- Internal Storage: 500 GB

6.3.4 Algorithm Hyperparameters

Hyperparameters are the values of parameters used to configure a machine learning model, and they must be set before the learning process, which is the main difference from parameters found *during* the learning process. Since the performance of the model is unknown for a given combination of dataset and hyperparameters, it is necessary to explore the range of possible values to obtain optimal hyperparameters. This optimization process is called *tuning*.

The strategy we follow to find optimal parameters is a simple grid search: For each combination of hyperparameter values, we train a model and look at the evaluation metrics for comparison.

Below is a list of hyperparameters and their definitions for all algorithms. The list of values for which they are tested can be seen in Table 6.3.

Isolation Forest

1. **Number of estimators**: The number of isolation trees generated in the isolation forest, also called the number of base estimators in the ensemble.
2. **Max samples**: The number of samples drawn from the input matrix for training each of the iTrees. This number cannot be higher than the number of samples in the input dataset.

3. **Contamination**: Percentage of points in the datasets that are anomalous. The value of the contamination parameter is used during fitting when defining the threshold for the decision function.

4. **Max features**: This parameter enables us to control the number of features to pull from the dataset to train each of the iTrees. The maximum value is the number of features in the dataset.

PCA

1. **Number of components**: Number of output features (principal components) to be extracted in the PCA. If a decimal number is specified, it is used to represent the variance ratio of the principal components

2. **Threshold**: The threshold for anomaly detection, which is a value between 0 and 1. When this value is exceeded, the sample is considered an anomaly. The threshold value can be set to be automatically calculated using the Q-statistics using the value of the alpha hyperparameter

3. **Alpha**: The alpha values for the Q-statistic to calculate the threshold for anomaly detection.

Invariants Mining

1. **Percentage**: The support ratio, or the percentage of samples that do not break the invariant. We say that an invariant v_i is not broken if the condition $|X_j v_i| < \epsilon$ is satisfied, where X_i is i-th sample in the dataset and ϵ is another user defined parameter.

2. **Epsilon**: A threshold for estimating the invariant space.

Log Clustering

1. **Max distance**: A value of the distance between the clusters, that is used as a threshold to terminate the clustering process.

2. **Anomaly threshold**: The threshold value for anomaly detection scores, which is a distance to cluster anomalies. If this value is exceeded, the sample is considered an anomaly.

To obtain the optimal hyperparameters, we will run algorithms for each combination of hyperparameters described in the Table 6.3. We will perform these tests using only the basic event count embeddings, and after finding the optimal parameters, we will use them to train the models on the TF-IDF embeddings as well.

We evaluate the results of the experiments using the metrics defined in Section 6.2 to find the optimal hyperparameter values. Since the F1-score combines both precision and recall measures, we select the final set of hyperparameters by the highest F1 score, so we optimize both metrics by optimizing F1. The evaluation is performed on the test dataset.

It is important to note here that we did not perform any tuning of the hyperparameters of the Invariant Mining model and performed the experiments using the default values set by the Loglizer library. The reason for this is that

Algorithm	Hyperparameter	Values
Isolation Forest	*Number of estimators*	50, 100, 150, 201
	Max samples	150, 180, 201
	Contamination	0, 0.02, 0.03, 0.1
	Max features	3693
PCA	*Number of components*	0.85, 0.90, 0.95
	Threshold	auto
	Alpha	0.0001, 0.001, 0.005, 0.01
Invariant Mining	*Percentage*	0.98
	Epsilon	0.5
Log Clustering	*Max distance*	0.3, 0.5
	Anomaly threshold	0.3, 0.5

Table 6.3: Hyperparameters to be tuned and their respective values which are tested for each anomaly detection method.

the process of invariant mining is extremely time consuming and inefficient compared to the other algorithms. It took several hours to train the invariants mining model, while other algorithms were trained in a few seconds. Moreover, we obtained satisfactory results with other algorithms, thereby we decided not to explore further hyperparameter combinations to tune the invariants mining model.

A detailed list of the results of the hyperparameter tuning experiments can be found in the Appendix B.1.1. A summary of the tuned and final values of the hyperparameters of the specified machine learning approaches used for anomaly detection is given in Table 6.4.

Algorithm	Hyperparameter	Value
Isolation Forest	*Number of estimators*	100
	Max samples	150
	Contamination	0.1
	Max features	3693
PCA	*Number of components*	0.95
	Threshold	auto
	Alpha	0.005
Invariant Mining	*Percentage*	0.98
	Epsilon	5
Log Clustering	*Max distance*	0.3
	Anomaly threshold	0.3

Table 6.4: Final hyperparameter values for each anomaly detection method, that are further used for experiments

7. Results

In this chapter, we present the results of anomaly detection experiments performed with two different feature embeddings. We use two methods of evaluation: comparing the metrics of accuracy, precision, recall and $F1$ to evaluate the performance with the labeled test dataset and validating the results using domain experts.

Numerical validation of the models will give us a conclusive answer to **RQ1**. To answer **RQ2**, we will compare the validation metrics computed for both the event count and TF-IDF representations. Finally, we use the test set to evaluate the performance of the selected anomaly detection algorithms in answering **RQ3**. A thorough analysis of these results along with the help of expert validation at the end of this chapter will directly address **RQ4**.

Proper evaluation is a crucial step in machine learning. Supervised methods typically check how the model performs on the training data, as well as on previously unseen datasets called *test* or *validation*, using various performance metrics. If the results are not satisfactory, this information can be used to optimize the model's hyperparameters and repeat the process. However, in the case of unsupervised learning, which we used in our research, it is not immediately obvious how to approach model evaluation.

The only dataset we can consider labeled is the Nightly dataset used for training. All the data collected during the night are *normal* instances without anomalies, but we need to test whether the actual anomalies are detected by our detectors and whether the detected anomalies match actual anomalies.

As described in Chapter 5 when defining our datasets, we manually created a test set of logs containing two types of observed anomaly scenarios that are included in the Anomalies dataset: Killing the Redis server and Killing the RabbitMQ message broker. In our simulations of these scenarios, we know the specific time range in which an anomaly occurs, so we can hand-label certain feature embeddings. Also, a portion of the Nightly Test dataset that contains data not seen during the training phase is used as part of the test dataset. Finally, we include the Glostrup Calling dataset in our test set, which contains a set of log messages generated during calls, as it is very important that our models do not detect calls as anomalies.

The weakness of this approach is that there are only two known types of anomalies in the test dataset. There is no way to tell if the models behave correctly on live and production data. As a compromise, we use the Daily dataset whose labels are unknown to generate predictions. We then perform what we call *expert validation*. We ask domain experts from the Motorola Solutions SmartConnect team to evaluate a small subset of the predictions generated by our best-performing models.

7.1 Test Dataset Validation

The interpretation of the anomaly detection models by the evaluation metrics obtained on the test set (Section 5.2) containing two types of anomalies helped us to verify the **RQ1** proposed in Chapter 1, that the logs generated by Motorola SmartConnect are indeed generated in such a way that they can be

used for anomaly detection. Let us now take a closer look at the results of the machine learning algorithms that allow us to answer the remaining research questions.

Table 7.1 shows the summarized experiment results on the test dataset for the models Isolation Forest, PCA, Invariants Mining and Log Clustering. We first compare the results in terms of feature embeddings and then discuss how the different anomaly detection algorithms performed in comparison to each other.

Algorithm	Embedding	Precision	Recall	F1	Accuracy
Isolation Forest	Event count	100.00%	57.14%	66.67%	87.79%
	TF-IDF	33.33%	14.81%	20.51%	76.15%
PCA	Event count	100.00%	100.00%	100.00%	100.00%
	TF-IDF	100.00%	100.00%	100.00%	100.00%
Invariants Mining	Event count	22.22%	100.00%	36.36%	25.19%
	TF-IDF	21.60%	100.00%	35.53%	24.62%
Log Clustering	Event count	100.00%	100.00%	100.00%	100.00%
	TF-IDF	100.00%	100.00%	100.00%	100.00%

Table 7.1: The precision, recall, F1 score, and accuracy for anomaly detection methods computed on the test dataset with two different embeddings: Event Count and TF-IDF. Both embeddings are generated with a sliding window of two minutes.

7.1.1 Feature Embeddings Comparison

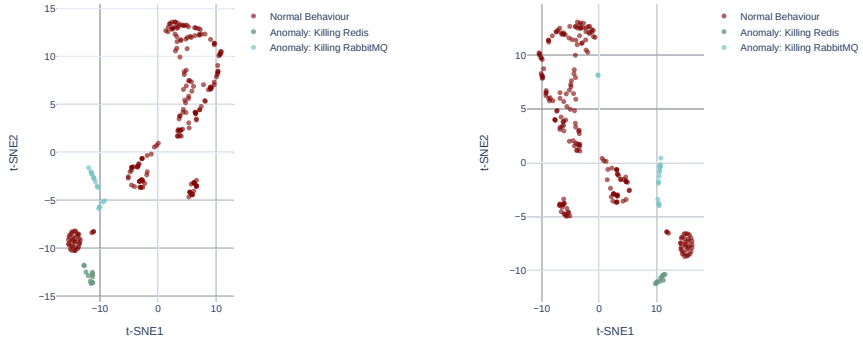
To address **RQ2**, we wanted to see how different feature embeddings in a numeric vector affected the results. As described in previous chapters, we proposed two embeddings: a simple event count vector and a weighted TF-IDF vector. TF-IDF is obtained from event count weight by giving more weight to events that occur less frequently in the whole dataset. We wanted to see if this type of information could bring more knowledge that would be beneficial to the anomaly detection model.

From the results, both embeddings perform very similarly, and it is not possible to tell from the test dataset which one would perform better on unknown anomalies. If we compare in Figure 7.1 the t-SNE visualization applied to normal data points and anomaly data points of different embeddings, we can see that both event count and TF-IDF treat anomalies and normal data in a similar way. Both event count embedding and TF-IDF embedding produced clusters in the embedding space that are visually visible and can distinguish between normal and anomalous data points.

Another means of comparing these two embeddings is by looking at the histogram in Figure 7.2. We expected TF-IDF to assign a higher value to events that occur infrequently, which we hypothesized should include events that are representative of anomalies. The histograms seem to agree in the middle part of the plot, but for the first five hundred event type IDs a much lower weight is observed in the TF-IDF histogram compared to the event count histogram. After manually inspecting the feature vectors, we noticed a pattern in the

frequency of event types with lower IDs. The corresponding event types occur with identical frequency in successive log sequences. This is to be expected and makes sense since the events behind these event type IDs include starting a call, attempting to start a new TLS connection, and other regularly occurring events. Therefore, these event types can be considered as frequently occurring events, which is why the TF-IDF algorithm assigns them a lower weight, which is reflected in the histogram.

All in all, considering that event count embedding is simple and much easier to interpret than TF-IDF and TF-IDF did not perform significantly better in the test dataset, we consider event count embedding as the preferred representation of the dataset for practical use.



(a) Event count embedding (b) TF-IDF embedding

Figure 7.1: Event count embedding (left) and TF-IDF embedding (right) of the normal and anomalous log sequences plotted using PCA.

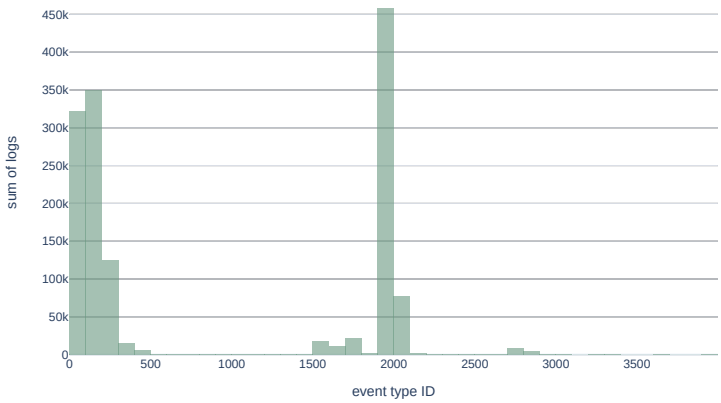
7.1.2 Anomaly Detection Methods Comparison

In this section, we describe how well do the four proposed anomaly detection methods perform on the labeled test dataset. To support our reasoning, we refer to the reported experimental results using evaluation metrics for embedding the test dataset with a two minute sliding window from Table 7.1.

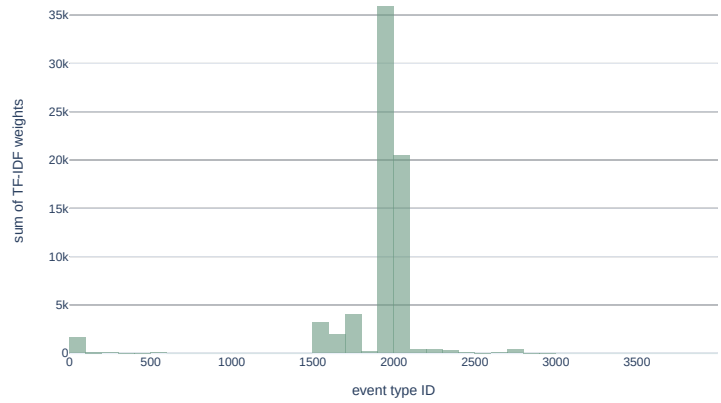
In order of importance, it is more important for a live production system not to miss an anomaly than to correctly detect anomaly-free data points. In other words, false negatives are more alarming than false positives. To analyze the ratio of identified anomalous and non-anomalous log sequences within all datasets, we derived confusion matrices for the Isolation Forest, PCA, Invariants Mining and Log Clustering models, which are shown in Table 7.2, Table 7.3, Table 7.4 and Table 7.5. Figure 7.3 gives a visual interpretation of which points were misclassified in each anomaly detection method.

From the results table, it can be seen that all the anomaly detection methods give a decent result, with Log Clustering and PCA performing significantly better than Invariants Mining and Isolation Forest.

Among the four unsupervised anomaly detection algorithms, the evaluation metrics for Invariants Mining model are the lowest with the values of



(a) Event count



(b) TF-IDF

Figure 7.2: Histogram distribution plot of the event type IDs in the test dataset, where the logs of the same event type ID, shown on the x-axis, are binned together and their cumulative count is shown on the y-axis. (a) is a histogram generated over event count embeddings, and (b) is a histogram generated over TF-IDF embeddings.

22.22% precision, 100.00% recall and 36.36% F1 score. Figure 7.3 (c) shows that Invariants Mining has a tendency to falsely identify normal data points as anomalies. This poor performance is not surprising and we suspect that there are two reasons for this occurrence. First, as explained in Section 6.3, we did not perform hyperparameter exploration because the training process for Invariants Mining on our dataset is extremely ineffective. However, we assume that the main reason for the inability to detect anomalies is the way we per-

formed windowing in log sequences. In similar research that successfully used Invariants Mining, such as He et al. [9], they used *session ID* or *process ID* shared between logs as a key for grouping logs into log sequences. We, on the other hand, focused on the timely relationship between logs and grouped them into log sequences in consecutive two-minute windows. Therefore, there is no guarantee that log events that are part of the invariants revealed by Invariants Mining are in the same log sequence in the resulting embedding. Therefore, it is expected that the Invariants Mining method does not perform well in the feature representation of our log dataset.

Another algorithm that did not perform well is Isolation Forest with 100% precision, but 57.14% recall and 66.67% F1 score. In 7.3 (a), we can observe that Isolation Forest misses a high number of anomalies, resulting in low recall and a high number of false negatives as well as false positives. We concluded that one reason for this finding is that Isolation Forest is not well suited for the one-class learning case (training set containing only negative samples) [34]. The leaves of iTrees provide an anomaly score equal to the length of the average path from the node (sample from the training set) to that leaf. The shorter the path, the smaller the anomaly score and the higher the probability that it is an outlier. Since all samples in the training dataset are normal, assigned anomaly scores may be biased. Therefore, the performance of Isolation Forest does not perform well on our dataset.

Log clustering and PCA achieved superior performance compared to other anomaly detection methods proposed in our research. The final result is 100.00% precision, recall, F1 score and accuracy on the test dataset for both Log Clustering and PCA. It is of our interest to investigate these two algorithms further, and in the next section we will ask the developers of Motorola Solutions to evaluate the results of these two algorithms on a previously unseen dataset (Daily) for which we have no labels and only experts knowledgeable in the field can say whether the predictions are correct.

		Predicted $\hat{l}$	
		Anomaly	Normal
Actual l	Anomaly	**19**	**9**
	Normal	**10**	**93**

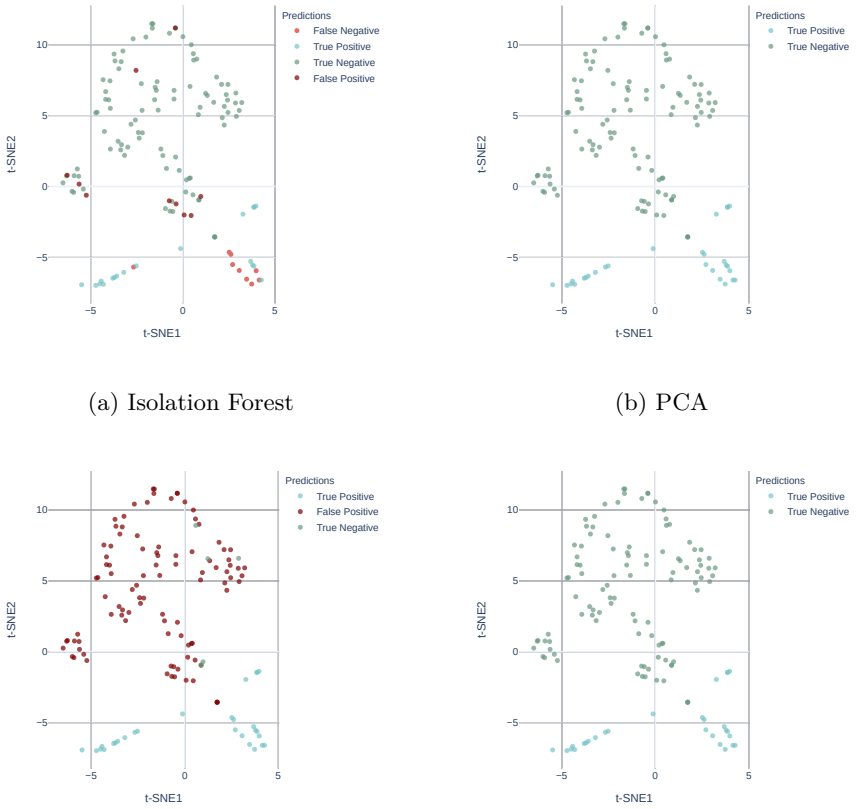
Table 7.2: A confusion matrix for Isolation Forest model on testing dataset.

		Predicted $\hat{l}$	
		Anomaly	Normal
Actual l	Anomaly	**28**	**0**
	Normal	**0**	**103**

Table 7.3: A confusion matrix for PCA model on testing dataset.

7.2 Expert Validation

Thanks to the developers of Motorola Solutions, we were able to obtain external validation of our results. We consulted experts to evaluate the detected



(a) Isolation Forest

(b) PCA

(c) Invariants Mining

(d) Log Clustering

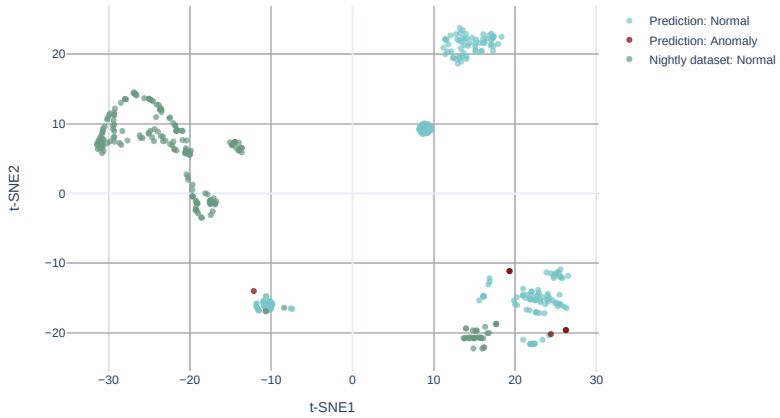
Figure 7.3: Comparison of the predictions of Isolation Forest (a), PCA (b), Invariants Mining (c), and Log Clustering (d) on the labeled test data set. It can be observed from the plots that PCA and Log Clustering algorithms handle testing dataset with a 100% accuracy.

		Predicted $\hat{l}$	
		Anomaly	Normal
Actual l	Anomaly	**28**	**0**
	Normal	**98**	**5**

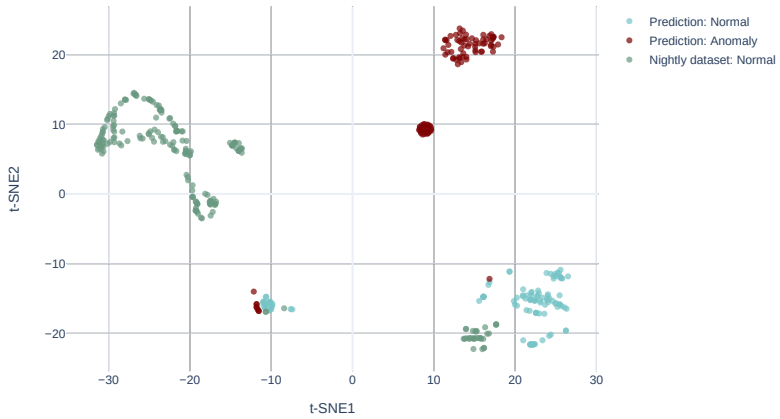
Table 7.4: A confusion matrix for Invariants Mining model on testing dataset.

		Predicted $\hat{l}$	
		Anomaly	Normal
Actual l	Anomaly	**28**	**0**
	Normal	**0**	**103**

Table 7.5: A confusion matrix for Log Clustering model on testing dataset.



(a) Log Clustering



(b) PCA

Figure 7.4: Comparison of Log Clustering (a) and PCA (b) predictions on the unlabeled Daily dataset. Green data points represent the normal data points as a reference point, blue and red highlighted data points represent predictions of a normal and anomalous log sequence, respectively.

anomalies, but also to evaluate samples detected as normal by two of the best performing anomaly detection models: PCA and Log Clustering.

Figure 7.4 shows the t-SNE plots of the predictions for the Daily data obtained by the trained PCA and Log Clustering models. The Daily dataset is plotted on top of the normal data points for a more convenient visual comparison. For t-SNE and PCA plots produced on the predictions of the Daily dataset by the other less performing algorithms studied in our research, see Appendix C.1 and Appendix C.2, respectively.

7.2.1 Analysis

The goal of this step was to expose our machine learning models to anomalies that are difficult to simulate manually, or that we are unaware of and thus could not validate against on a larger scale. However, since our models are designed to be able to identify anomalies not seen before, we studied a random sample from the experimental environment during the day. It should follow a very different distribution compared to both the Nightly dataset and the dataset with known anomalies.

The experts were given a small sample of time windows extracted from the Daily dataset. The sample of 12 windows was given to them in the form of both a *histogram* of event templates and a sequence of *raw logs*. The specific time windows were selected based on their predicted label, so that each predicted class had multiple representative windows in the sample for the two of the best performing anomaly detection models, Log Clustering and PCA. We attempted to select time windows whose predicted labels differed between the models as well as those that were the same.

In these windows, the domain experts managed to identify an anomaly where information from UDP packets is discarded. The packets usually carry the audio information of the calls and their loss can be caused by poor connectivity, which can reduce the overall quality of the calls sent or received. This is certainly a valid anomaly to be detected.

Upon closer inspection, we noticed that the cluster in the upper right corner corresponds to the anomaly of a discarded packet. As can be seen in Figure 7.4, this behaviour was only detected as an anomaly by PCA. On the other hand, Log Clustering is less sensitive to the cluster containing calls with larger pocket loss. When this specific anomaly occurs, it does not trigger any other processes in the system and it continues to run as usual trying to deliver them based on best-effort delivery, therefore it can also be argued that a UDP dropout of small size is not an anomaly that needs to raise alarm. The fact that Log Clustering did not recognize these points as anomalies could be fixed by adjusting hyperparameters.

Additionally, the experts did not report any false negatives (anomalies labeled as anomaly free by our models). This information on its own however doesn't prove that there are no instances of those. Clearly, it is more difficult to identify a false negative by hand. In general, there would be a majority of windows labeled as anomaly-free and we cannot expect human to crawl through all of them effectively verifying that no false negative occurred in the analyzed time span. It is understandable, that a bare eye is not able to spot these just based on the histogram and logs assigned to each of the windows, that are meant for troubleshooting the predictions.

8. Conclusions and Future Work

In our work, we have attempted to apply anomaly detection to a real-world dataset obtained in collaboration with Motorola Solution's Smart Connect. We have formulated a methodology for analyzing unstructured system log data.

Our methodology workflow is initiated by *data collection*. The implemented solution to collect log data produced by SmartConnect's microservices queries Elasticsearch within the cluster. We collected data instances representing the normal behaviour of the system, as well as some known anomalies that we simulated on site.

Since logs typically contain variable parts that result in different log messages for the same log event type, a *log parsing* technique is used to extract unique log event types. For this purpose, an online parser of the Drain3 library is used.

We have proposed two different approaches for *feature extraction*, namely the event count vector and the weighted TF-IDF vector. Both representations are generated from two minute windows. In each time window, the occurrence of unique event types is counted and in the case of TF-IDF, the counts are weighted. It has been shown that both representations perform similarly well.

Finally, we examined and compared four different *anomaly detection* models, along with an analysis of their architecture and hyperparameter selection. Model evaluation was a problematic part of our research, due to the lack of known anomaly scenarios to test and thus a lack of data labels as well. We performed an evaluation of the selected algorithms on a task of unsupervised one-class anomaly detection on log data on a small, manually generated test dataset with two known anomalies. In a second step of the evaluation, we asked experts to assess the predicted labels for unknown anomalies. We concluded that simple algorithms, such as Log Clustering and PCA, performed very well on the test dataset.

During our research, we made the following main contributions:

- Performed a detailed review of anomaly detection methods
- Demonstrated the applicability of our methodology to a real-world case analysis
- Designed core anomaly detection components, that can be used for future anomaly detection pipeline in production dataset in SmartConnect
- Presented future extensions for the work to be production ready and proposed possible new directions of the research

8.1 Future Work

In the time frame of six months, we were able to prove that applying machine learning based monitoring tools to detect anomalies in Motorola SmartConnect is well possible. Along the way, however, many interesting questions arose that we could not address within the scope of this project. Also, our solution should not be considered a finished product that can be packaged and shipped. Definitely, more software engineering work needs to be done before we can

claim that. For now, our solution is mainly for experimental purposes. Here we describe the questions whose answers need to to be explored in future research.

8.1.1 Anomaly Detection Pipeline

As described earlier in Section 5.1, so far our anomaly detection tool only observes the experimental environment intended for developers to try out new features.

This is only a proof-of-concept for our solution. To take full advantage of an automated anomaly detection product, it needs be deployed in the production environment. However, in a large company like Motorola Solution, this takes more than just a few months, which we had available for this work.

Let us illustrate what processes are required to improve our solution so that it can be useful in production.

First, we need to develop a proper pipeline for continuous training of the model(s). We have the foundation in place, but, what remains to be done is to automate the process that checks if the nightly tests passed correctly, and if so, the log samples from that night are downloaded, labeled, and added to the training dataset. The model needs to be re-trained and updated in production. At this stage, the question of how much training is too much also needs to be addressed.

At the same time, our research has not addressed what the specific ways would be to trigger an alarm each time an anomaly is found. Then, a mechanism must be triggered to deal with the detected problem, whether it is a program that attempts to heal the problem or the notification of a person responsible for observing the state of the system.

8.1.2 Code Coverage

After becoming familiar with the Motorola SmartConnect system, we decided to use one-class learning (classification) methods to detect anomalies. Domain knowledge proved critical as it allowed us to take advantage of the fact that we could collect data from nightly tests. We then selected machine learning algorithms that we fed only negative data.

We based our solution on some assumptions about the logs we collect from passing tests:

1. If a test passes, it contains no anomalies.
2. The tests cover the code “sufficiently” and largely examine the data paths that can be executed within the system.

The first assumption basically says that even if the tests mimic scenarios where a failure is simulated and the system can recover from that failure, we do not want to consider that an anomaly. In this case, no alarm is raised because the system does not require human intervention and is prepared to handle the problem.

The last point brings us to an interesting feature of our solution, which opens a whole new perspective that can be explored in future research.

By the nature of the embedding chosen and the machine learning methods studied, the models run into trouble in case when there is an anomaly free scenario that was not sufficiently present in the training dataset (or better, could not be inferred from the dataset). If this non-anomalous scenario then shows up in the system monitored by our anomaly detection tool, false positives will result.
In our case, however, we can rephrase the statement into the claim that if the model frequently reports anomalies that are not in fact anomalies (false positives), this can be turned into valuable information about the tests themselves. More specifically, it means that the tests are not examining some properties of the system and are prone to introducing errors into the product. This observation, along with our anomaly detection solution, can be used as the basis for a tool that could be plugged into a system monitoring process.

8.2 Weaknesses and Limitations

Although we have been able to achieve some great results and discoveries during our methodological process, there are a number of possible improvements that could be made. In this section we will address some of the weaknesses and limitations that we encountered during our research.

8.2.1 Evaluation

First and foremost, the most difficult part in anomaly detection is actually verifying the results because our labeled dataset and the set of known anomalies have a very limited size. For this reason, we were unable to draw conclusions about performance on a real-world dataset. However, due to the complexity of the log data, this was not possible, as labeling the data is extremely time consuming. We asked experts to manually validate of the predictions on an experimental dataset. They approved of the reported anomalies and did not report any false negatives.

However, relying on a human being as the one to validate the actual results and performance makes the whole process more error-prone and even more time consuming.

8.2.2 Dataset

The major limitation of our research is that we were not able to verify our results using datasets from the live production environment. Instead, we used log data generated in a test and experimental environment. This is due to the fact that log data generated in the production environment is highly confidential as it may contain information that can be easily exploited if not handled properly.

To overcome this, a strategy for anonymous handling and storage of data in our program needs to be developed, which is beyond the scope of this paper.

8.2.3 Optimal Window Size

Another issue that has not been fully addressed in this paper is further experimentation with the relationship between performance and window size. In our

study, we only worked with embedding in time windows with a fixed time of two minutes and the windows were non overlapping. It can be argued, that since the downtime caused by anomalies, which we primarily studied, is less than two minutes, our solutions should perform reasonably well. However, it remains to be explored how this embedding choice would cope with particularly longer lasting anomalies.

8.2.4 Variant Length of Feature Vector

In the feature engineering step, we embed features into an event count matrix, where we mark how many times an event occurred in a given time window. However, when embedding previously unseen records, it will eventually happen that a previously unseen event occurs and thus the length of the feature vector increases. In the current solution, when this happens, we have to go back and re-train our models to match the new vector size.

To make our solution more robust, an additional column should be introduced in the feature vector. The frequency of this event type would be incremented whenever an event occurs that was not represented in the training data. After that, in the model re-training step (which is suggested as a possible extension in Section 8.1.1) the new event types integrated and the event count would be properly increased.

8.3 Conclusion

In our research, we reviewed and tried four unsupervised anomaly detection methods to prove that outlier detection using machine learning techniques can be applied to log data in the domain of MSI's SmartConnect product.

Based on the conducted results of the experiments on the feature extraction methods proposed in our work on Motorola Solutions data, we can answer the research questions outlined in Chapter 1. In the remainder of the chapter, we will provide a short answer to each of the research questions.

We focused on detecting anomalies that were known to us before conducting the experiments. We attempted to prove that our algorithms could detect them while not triggering error when normal events occured. However, the desired application of our solution it to be able to detect anomalies by monitoring the system in real time. In the section above, we elaborated the rough workflow of how the alarms would be raised in the future anomaly detection pipeline. For this use-case, it would be worthwhile to experiment with smaller windows and possibly overlapping windows, as opposed to the strict fixed-size embedding we are doing at the moment. The reason for looking at smaller windows is that a finer granularity of problems could be detected this way. If an anomaly occurs, the system breaks down, and users suffer, it is no longer as useful to detect an anomaly because the system has been severely impacted. However, if the window is small enough to capture an event that leads to an anomaly, something can be done about the initial problem at some point to prevent the entire anomaly (consisting of $n, n > 1$ windows) from occurring.

RQ1: *Can anomaly detection be applied on the log data that are at the moment produced by Motorola Solutions SmartConnect system?*

Before the actual process of active anomaly detection, we performed an initial exploration of the dataset. After we plotted the two types of observed anomalies, killing Redis and killing RabbitMQ, on top of the Nightly dataset of normal instances (Figure 6.7), the plot showed three distinct clusters. Each cluster contained similar data points (log sequences) grouped into clusters, and the cluster of normal instances was separated from the cluster of anomalous instances, based on the behaviour of the underlying data points. In addition, visual interpretation also allowed us to distinguish between two different types of anomalies.

Being able to see anomalies with the naked eye gives us the "why" of a particular decision made by the anomaly detection algorithm. Interpretability and explainability in machine learning is extremely important. It helps us to understand what the reasons are for the results, and it also helps us trust the final models.

Therefore, we argue that visual analysis is direct evidence that the dataset of logs produced by Motorola Solutions SmartConnect contains enough information for anomaly detection and the answer to the first research question is yes.

RQ2: *Does a weighted event vector representation of raw log messages logged by Motorola Solution's SmartConnect production system provide sufficient information for spotting anomalous outliers in the data?*

As part of our research, we designed a preprocessing stage that transfers unstructured log messages into a structured format, or *embedding*. We proposed two different types of feature embeddings: an event count vector and TF-IDF weighted vector.

To answer this question, we conducted machine learning experiments and discovered that our best performing models can successfully detect all types of known anomalies, with an F1 score of 100% on the test dataset.

Both types of feature embeddings performed similarly well, however we choose to prefer the event count embedding due to its simplicity in terms of generation and interpretation. It seems that a simple representation in the context of the frequency of different event counts is detailed enough to detect outliers within the normal data.

These results confirm that this type of feature extraction, which is commonly used in anomaly detection, successfully reflects the characteristics of SmartConnect log messages for further outlier analysis.

RQ3: *Which anomaly detection techniques and approaches are applicable on time series data produced by Motorola SmartConnect system?*

To answer the third research question, we conducted a thorough investigation of anomaly detection methods, approaches, and similar research papers, in which they have been successfully used. The architecture of the anomaly detection process consists of many more steps than just the selection and execution of an ML algorithm.

We proposed a workflow of our research that consists of four important steps: Data collection, Log parsing, Feature Engineering and Anomaly Detection.

As a result of our research, we have compiled a short list of four representative state-of-the-art algorithms for unsupervised anomaly detection, including Isolation Forest, PCA, Invariants Mining and Log Clustering. Based on the research results, we investigated several log parsing algorithms, that are also a critical component of successful anomaly detection. We chose to use an online log parser called Drain3.

RQ4: *How are the techniques performing on the given log dataset and why they do or do not yield satisfying results? If not, what could be improved about the log that would increase the performance of anomaly detection?*

With the accumulated knowledge to convert data into a structured format and having verified our assumptions using data visualization techniques, we were ready to perform the final part of the research - conducting experiments. To find the ideal parameters for anomaly detection algorithms, we performed a simple grid search over the hyperparameter space. We did not care to explore the hyperparameters in detail, as achieving optimal parameters and results was not the main goal of this work.

Anomaly detection algorithms are judged on their ability to correctly identify anomalies. To measure this ability, we used four types of evaluation metrics: Precision, Recall, F1 Score and Accuracy. On the test set, the log clustering and PCA algorithms are able to detect 100% of killing Redis and killing RabbitMQ anomalies and 100% of normal data instances. A special type of normal samples are calls between two radios, which appeared as outliers on some plots, are also correctly detected as anomaly-free.

The killing Redis and RabbitMQ anomalies are realtively easy to detect due to the logs generated and the high volumes generated on these occasions. However, reproducing anomalies in the SmartConnect system is not a trivial task. The data from these two cases of anomalies were obtained on site physically making the calls while stopping the connection to the Redis and RabbitMQ services, when the experimental environment was not interrupted by other experiments performed by developers. We also argue that being able to recognize these two anomalies and distinguish them from each other is helpful information that it is possible to detect an occurrence of anomalies in this dataset.

To support this claim, we leveraged the knowledge of domain experts - developers in the SmartConnect team to evaluate the discovered results on dataset with unknown number and type of anomalies. In this way, we obtained an unbiased view of the performance of the trained models. Due to the high number of logs in time windows that the experts were asked to examine, they could not confidently confirm the predicted labels of all time windows. However, they managed to find another anomaly cluster, that was correctly identified as an anomaly by PCA. The anomaly represented by this cluster was unknown before the experiment.

Bibliography

[1] Motorola Solutions. APX NEXT Fact Sheet SmartConnect. https://www.daywireless.com/downloads/motorola/motorola-apx-next-smartconnect-fact-sheet.pdf.

[2] Q. Fu, J. Lou, Y. Wang, and J. Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *2009 Ninth IEEE International Conference on Data Mining*, pages 149–158, 2009.

[3] Qingwei Lin, Hongyu Zhang, Jian-Guang Lou, Yu Zhang, and Xuewei Chen. Log clustering based problem identification for online service systems. pages 102–111, 05 2016.

[4] Ricardo Dunia and Joe Qin. Multi-dimensional fault diagnosis using a subspace approach. 04 1997.

[5] Jian-Guang Lou, Qiang Fu, Shengqi Yang, Ye Xu, and Jiang Li. Mining invariants from console logs for system problem detection. 01 2010.

[6] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. Isolation-based anomaly detection. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 6(1):1–39, 2012.

[7] Zhilong Wang, Min Zhang, Danshi Wang, Chuang Song, Min Liu, Jin Li, Liqi Lou, and Zhuo Liu. Failure prediction using machine learning and time series in optical network. *Optics Express*, 25(16):18553–18565, 2017.

[8] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 117–132, New York, NY, USA, 2009. Association for Computing Machinery.

[9] S. He, J. Zhu, P. He, and M. R. Lyu. Experience report: System log analysis for anomaly detection. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, pages 207–218, 2016.

[10] Raghavendra Chalapathy University of Sydney, Capital Markets Cooperative Research Centre, Sanjay Chawla Qatar Computing Research Institute, and Hbku. Deep learning for anomaly detection: A survey. 2019.

[11] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 1285–1298, New York, NY, USA, 2017. Association for Computing Machinery.

[12] K. Zhang, J. Xu, M. R. Min, G. Jiang, K. Pelechrinis, and H. Zhang. Automated it system failure prediction: A deep learning approach. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 1291–1300, 2016.

[13] Motorola Solutions. What is a two way radio? https://www.motorolasolutions.com/en_xu/solutions/what-is-two-way-radio.html.

[14] Motorola Solutions. APX Series P25 Two-Way Radios. https://www.motorolasolutions.com/en_xa/products/p25-products/apx-story.html.

[15] APX Next SmartConnect Presentation. 2021.

[16] Erik Dahlman, Stefan Parkvall, and Johan Skold. *4G: LTE/LTE-advanced for mobile broadband*. Academic press, 2013.

[17] VIAVI Solutions Inc. Project 25. https://www.viavisolutions.com/en-us/project-25.

[18] Business Wire. Motorola Solutions' Next Generation APX NEXT Smart Radio Brings New Intelligence and Technology to Public Safety.

[19] Irakli Nadareishvili, Ronnie Mitra, Matt McLarty, and Mike Amundsen. *Microservice architecture: aligning principles, practices, and culture*. "O'Reilly Media, Inc.", 2016.

[20] Martin Fowler and James Lewis. Microservices a definition of this new architectural term. *http://martinfowler.com/articles/microservices.html*, 2014.

[21] Kasun Indrasiri and Prabath Siriwardena. Microservices for the enterprise. *Apress, Berkeley*, 2018.

[22] Clinton Gormley and Zachary Tong. *Elasticsearch: the definitive guide: a distributed real-time search and analytics engine*. "O'Reilly Media, Inc.", 2015.

[23] Klaus Elhardt and Rudolf Bayer. A database cache for high performance and fast restart in database systems. *ACM Transactions on Database Systems (TODS)*, 9(4):503–525, 1984.

[24] Keith Ord. Outliers in statistical data: V. Barnett and T. Lewis, 1994, (John Wiley & Sons, Chichester), 584 pp.,[UK pound] 55.00, ISBN 0-471-93094-6. *International Journal of Forecasting*, 12(1):175–176, 1996.

[25] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM Comput. Surv.*, 41(3), July 2009.

[26] Diego García-Gil, Julián Luengo, Salvador García, and Francisco Herrera. Enabling smart data: Noise filtering in big data classification, 2017.

[27] X. Wu. Mining with noise knowledge: Error aware data mining. In *2007 International Conference on Computational Intelligence and Security (CIS 2007)*, pages xxix–xxix, 2007.

[28] M. Gupta, J. Gao, C. C. Aggarwal, and J. Han. Outlier detection for temporal data: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 26(9):2250–2267, 2014.

[29] Xiaojin Zhu and Andrew B Goldberg. Introduction to semi-supervised learning. *Synthesis lectures on artificial intelligence and machine learning*, 3(1):1–130, 2009.

[30] N. Goernitz, M. Kloft, K. Rieck, and U. Brefeld. Toward supervised anomaly detection. *Journal of Artificial Intelligence Research*, 46:235–262, Feb 2013.

[31] David Martinus Johannes Tax. One-class classification: Concept learning in the absence of counter-examples. 2002.

[32] Shehroz S. Khan and Michael G. Madden. One-class classification: taxonomy of study and review of techniques. *The Knowledge Engineering Review*, 29(3):345–374, 2014.

[33] Tan, Pang-Ning, Michael Steinbach, Michael Adeyeye Oshin, Vipin Kumar, and Vipin. *Introduction to Data Mining*. 05 2005.

[34] Jakob Sternby, Erik Thormarker, and Michael Liljenstam. Anomaly detection forest. In Giuseppe De Giacomo, Alejandro Catalá, Bistra Dilkina, Michela Milano, Senén Barro, Alberto Bugarín, and Jérôme Lang, editors, *ECAI 2020 - 24th European Conference on Artificial Intelligence, 29 August-8 September 2020, Santiago de Compostela, Spain, August 29 - September 8, 2020 - Including 10th Conference on Prestigious Applications of Artificial Intelligence (PAIS 2020)*, volume 325 of *Frontiers in Artificial Intelligence and Applications*, pages 1507–1514. IOS Press, 2020.

[35] Kevin P. Murphy. *Machine learning: a probabilistic perspective*. MIT Press, Cambridge, Mass. [u.a.], 2013.

[36] J. Jackson and Govind Mudholkar. Control procedures for residuals associated with principal component analysis. *Technometrics*, 21:341–349, 08 1979.

[37] J. C. Gower and G. J. S. Ross. Minimum spanning trees and single linkage cluster analysis. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 18(1):54–64, 1969.

[38] Log4j 2 API. `https://logging.apache.org/log4j/2.x/manual/api.html`. Online, Accessed: 2020-01-11.

[39] Serilog. `https://serilog.net/`. Online, Accessed: 2020-01-11.

[40] logging — Logging facility for Python. `https://docs.python.org/3/library/logging.html`. Online, Accessed: 2020-01-12.

[41] Vladimir Legeza, Anton Golubtsov, and Betsy Beyer. Structured logging: Crafting useful message content. 2019.

[42] P. He, J. Zhu, S. He, J. Li, and M. R. Lyu. An evaluation study on log parsing and its use in log mining. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 654–661, 2016.

[43] P. He, J. Zhu, Z. Zheng, and M. R. Lyu. Drain: An online log parsing approach with fixed depth tree. In *2017 IEEE International Conference on Web Services (ICWS)*, pages 33–40, 2017.

[44] R. Vaarandi. 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)*, pages 119–126, 2003.

[45] Liang Tang, Tao Li, and Chang-Shing Perng. Logsig: Generating system events from raw textual logs. pages 785–794, 10 2011.

[46] A. Makanju, A. N. Zincir-Heywood, and E. Milios. A lightweight algorithm for message type extraction in event logs . 2009.

[47] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael Jordan. Mining console logs for large-scale system problem detection. 01 2008.

[48] Jieming Zhu, Shilin He, Jinyang Liu, Pinjia He, Qi Xie, Zibin Zheng, and Michael R. Lyu. Tools and benchmarks for automated log parsing, 2018.

[49] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules. In *Proc. of 20th Intl. Conf. on VLDB*, pages 487–499, 1994.

[50] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Elements of reusable object-oriented software*. 1995.

[51] Drain3. `https://github.com/IBM/Drain3`. Online, Accessed: 2020-01-13.

[52] H. Izakian and W. Pedrycz. Anomaly detection in time series data using a fuzzy c-means clustering. In *2013 Joint IFSA World Congress and NAFIPS Annual Meeting (IFSA/NAFIPS)*, pages 1513–1518, 2013.

[53] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, USA, 2008.

[54] Stuart I Feldman. Make—a program for maintaining computer programs. *Software: Practice and experience*, 9(4):255–265, 1979.

[55] John W. Tukey. *Exploratory Data Analysis*. Addison-Wesley, 1977.

[56] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-SNE. *Journal of Machine Learning Research*, 9:2579–2605, 2008.

[57] Marina Sokolova, Nathalie Japkowicz, and Stan Szpakowicz. Beyond accuracy, f-score and roc: A family of discriminant measures for performance evaluation. volume Vol. 4304, pages 1015–1021, 01 2006.

List of Figures

A. Dataset

A.1 Log Properties List

1. @timestamp
2. _id
3. _index
4. _score
5. _type
6. agent.ephemeral_id
7. agent.hostname
8. agent.id
9. agent.type
10. agent.version
11. cloud.instance.id
12. cloud.instance.name
13. cloud.machine.type
14. cloud.provider
15. cloud.region
16. ecs.version
17. host.name
18. input.type
19. kubernetes.container.image
20. kubernetes.container.name
21. kubernetes.labels.app
22. kubernetes.labels.ignore-drain
23. kubernetes.labels.pdb
24. kubernetes.labels.pod-template-hash
25. kubernetes.namespace
26. kubernetes.node.name
27. kubernetes.pod.name

28. kubernetes.pod.uid
29. kubernetes.replicaset.name
30. log.file.path
31. log.offset
32. message
33. stream

B. Experiments

B.1 Experimental Setup

B.1.1 Hyperparameter Tuning

B.1.1.1 Log Clustering

Max distance	Anomaly threshold	**Precision**	**Recall**	**F1**	**Accuracy**
0.3	0.3	100.00%	100.00%	100.00%	100.00%
0.3	0.5	100.00%	100.00%	100.00%	100.00%
0.5	0.3	100.00%	100.00%	100.00%	100.00%
0.5	0.5	100.00%	100.00%	100.00%	100.00%

Table B.1: Precision, Recall, F1-score and Accuracy of the test dataset on Log Clustering model with different hyperparameter values.

B.1.1.2 Invariants Mining

Percentage	Epsilon	**Precision**	**Recall**	**F1**	**Accuracy**
0.98	5	22.22%	100.00%	36.36%	25.19%

Table B.2: Precision, Recall, F1-score and Accuracy of the test dataset on Invariants Mining model and its hyperparameter values.

B.1.1.3 PCA

Number of components	Threshold	Alpha	**Precision**	**Recall**	**F1**	**Accuracy**
0.85	auto	0.0001	100.00%	75.00%	85.71%	94.66%
0.85	auto	0.001	100.00%	85.71%	92.30%	96.95%
0.85	auto	0.005	100.00%	92.86%	96.30%	98.47%
0.85	auto	0.01	100.00%	92.86%	96.30%	98.47%
0.90	auto	0.0001	100.00%	75.00%	85.71%	94.66%
0.90	auto	0.001	100.00%	85.71%	92.31%	96.94%
0.90	auto	0.005	100.00%	92.86%	96.30%	98.47%
0.90	auto	0.01	100.00%	92.86%	96.30%	98.47%
0.95	auto	0.0001	100.00%	92.86%	96.30%	98.47%
0.95	auto	0.001	100.00%	96.43%	98.18%	99.23%
0.95	**auto**	**0.005**	**100.00%**	**100.00%**	**100.00%**	**100.00%**
0.95	**auto**	**0.01**	**100.00%**	**100.00%**	**100.00%**	**100.00%**

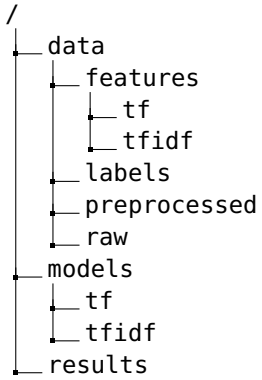
Table B.3: Precision, Recall, F1-score and Accuracy of the test dataset on PCA model with different hyperparameter values.

B.1.1.4 Isolation Forest

Number of estimators	Max samples	Contamination	Max features	**Precision**	**Recall**	**F1**	**Accuracy**
50	150	0	3693	100.00%	10.71%	19.35%	80.92%
50	150	0.02	3693	66.67%	7.14%	12.90%	79.34%
50	150	0.03	3693	50.00%	14.29%	22.22%	78.63%
50	150	0.1	3693	71.42%	3.57%	4.76%	69.47%
50	180	0	3693	0.00%	0.00%	0.00%	78.63%
50	180	0.02	3693	0.00%	0.00%	0.00%	76.33%
50	180	0.03	3693	33.33%	7.14%	11.76%	77.10%
50	**180**	**0.1**	**3693**	**80.00%**	**57.14%**	**66.67%**	**87.79%**
50	201	0	3693	78.63%	0.00%	0.00%	0.00%
50	201	0.02	3693	66.67%	14.23%	23.53%	80.15%
50	201	0.03	3693	0.00%	0.00%	0.00%	75.57%
50	201	0.1	3693	0.00%	0.00%	0.00%	67.94%
100	150	0	3693	0.00%	0.00%	0.00%	78.63%
100	150	0.02	3693	57.14%	14.23%	22.86%	79.39%
100	150	0.03	3693	40.00%	7.14%	12.12%	77.86%
100	**150**	**0.1**	**3693**	**65.51%**	**67.86%**	**66.67%**	**85.50%**
100	180	0	3693	0.00%	0.00%	0.00%	78.63%
100	180	0.02	3693	0.00%	0.00%	0.00%	77.86%
100	180	0.03	3693	0.00%	0.00%	0.00%	76.34%
100	180	0.1	3693	0.00%	0.00%	0.00%	71.76%
100	201	0	3693	100.00%	3.57%	6.90%	79.39%
100	201	0.02	3693	90.00%	32.14%	47.37%	84.73%
100	201	0.03	3693	50.00%	7.14%	12.50%	78.63%
100	201	0.1	3693	66.67%	50.00%	57.14%	83.97%
150	150	0	3693	0.00%	0.00%	0.00%	78.63%
150	150	0.02	3693	50.00%	7.14%	12.50%	78.63%
150	150	0.03	3693	33.33%	7.14%	11.76%	77.10%
150	150	0.1	3693	33.33%	21.42%	26.10%	74.04%
150	180	0	3693	0.00%	0.00%	0.00%	78.63%
150	180	0.02	3693	66.67%	7.14%	12.90%	79.38%
150	180	0.03	3693	50.00%	14.29%	22.22%	78.63%
150	180	0.1	3693	43.75%	25%	31.82%	77.10%
150	201	0	3693	0.00%	0.00%	0.00%	78.63%
150	201	0.02	3693	50.00%	7.14%	12.50%	78.63%
150	201	0.03	3693	0.00%	0.00%	0.00%	76.34%
150	201	0.1	3693	20.00%	10.71%	13.95%	71.76%
201	150	0	3693	0.00%	0.00%	0.00%	78.63%
201	150	0.02	3693	77.78%	25.00%	37.84%	82.44%
201	150	0.03	3693	71.43%	17.43%	28.57%	80.92%
201	150	0.1	3693	35.30%	21.43%	26.67%	74.81%
201	180	0	3693	0.00%	0.00%	0.00%	78.63%
201	180	0.02	3693	50.00%	7.14%	12.50%	78.63%
201	180	0.03	3693	40.00%	7.14%	12.12%	77.86%
201	180	0.1	3693	38.89%	25.00%	30.43%	75.57%
201	201	0	3693	0.00%	0.00%	0.00%	78.63%
201	201	0.02	3693	66.67%	14.29%	23.53%	80.15%
201	201	0.03	3693	42.86%	10.71%	17.14%	77.86%
201	201	0.1	3693	57.69%	53.57%	55.56%	81.68%

Table B.4: Precision, Recall, F1-score and Accuracy of the test dataset on Isolation Forest model with different hyperparameter values.

B.1.2 Directory structure used with Makefile



```
│   ├── metrics
│   │   ├── tf
│   │   └── tfidf
│   └── predictions
│       ├── tf
│       └── tfidf
├── src
│   ├── data
│   │   ├── categorization.py
│   │   ├── feature_extraction.py
│   │   └── template_mine.py
│   └── models
│       ├── invariant_mining.py
│       ├── isolation_forest.py
│       ├── log_clustering.py
│       ├── train.py
│       ├── evaluate.py
│       └── pca.py
└── Makefile
```

B.1.3 Makefile for running experiments

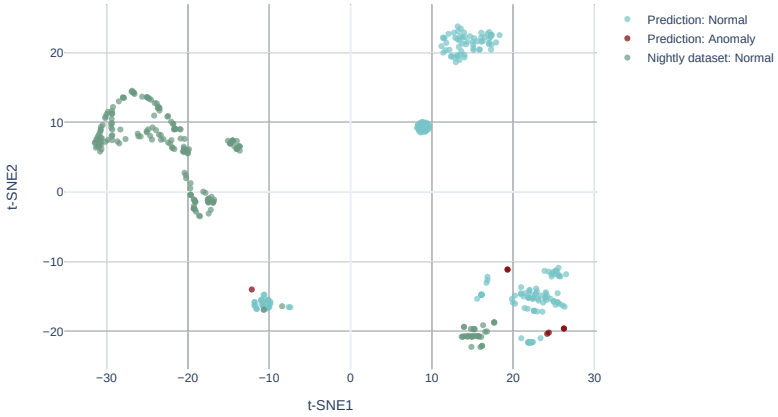
```
.PHONY: all clean data train evaluate

################################################
# GLOBALS                                      #
################################################

RAW_DATA_DIR=data/raw/
PREPROCESSED_DATA_DIR=data/preprocessed/
FEATURES_DATA_DIR=data/features/
LABELS_DATA_DIR=data/labels/

METRICS_DIR=results/metrics/
PREDICTIONS_DIR=results/predictions/

MODELS=isolation_forest log_clustering pca invariant_mining

TF_DIR=tf/
TFIDF_DIR=tfidf/

MODELS_DIR=models/

################################################
# COMMANDS                                     #
################################################

clean:
```

```
27          rm -f data/preprocessed/$(DATASET)-preprocessed.
                pickle
28          rm -f data/preprocessed/$(DATASET)-preprocessed.csv
29          rm -f data/features/tf/$(DATASET).npy
30          rm -f data/features/tf/$(DATASET).csv
31          rm -f data/features/tfidf/$(DATASET).npy
32          rm -f data/features/tfidf/$(DATASET).csv
33          rm -f models/tf/*.model
34          rm -f models/tfidf/*.model
35
36  all: evaluate
37
38  data: data/features/%.npy
39
40  train: models/tf/%.model models/tfidf/%.model
41
42  evaluate: results/%.csv
43
44  data/preprocessed/%.pickle: $(RAW_DATA_DIR)$(DATASET)-raw.
        json
45          python3 src/data/template_mine.py $< $(
                PREPROCESSED_DATA_DIR)$(DATASET)-preprocessed.
                pickle --csv $(PREPROCESSED_DATA_DIR)$(DATASET)-
                preprocessed.csv
46
47  data/features/%.npy: data/preprocessed/%-preprocessed.
        pickle
48          python3 src/data/feature_extract.py $(
                PREPROCESSED_DATA_DIR)$(DATASET)-preprocessed.
                pickle 5S --csv --tf $(FEATURES_DATA_DIR)$(
                TF_DIR)$(DATASET).npy --tf_idf $(
                FEATURES_DATA_DIR)$(TFIDF_DIR)$(DATASET).npy
49
50  models/tf/%.model: data/features/tf/%.npy
51          $(foreach model,$(MODELS),python3 src/models/
                train_$(model)_model.py $(FEATURES_DATA_DIR)$(
                TF_DIR)$(DATASET).npy models/tf/$(model).model;)
52
53  models/tfidf/%.model: data/features/tfidf/%.npy
54          $(foreach model,$(MODELS),python3 src/models/
                train_$(model)_model.py $(FEATURES_DATA_DIR)$(
                TFIDF_DIR)$(DATASET).npy models/tfidf/$(model).
                model;)
55
56  results/%.csv: models/tf/%.model models/tfidf/%.model
57          $(foreach model,$(MODELS),python3 src/models/
                evaluate_$(model)_model.py models/tf/$(model).
                model $(FEATURES_DATA_DIR)$(TF_DIR)$(DATASET)-
                test.npy $(LABELS_DATA_DIR)$(DATASET)-test-
                labels.npy $(METRICS_DIR)$(TF_DIR)$(DATASET)-$(
```

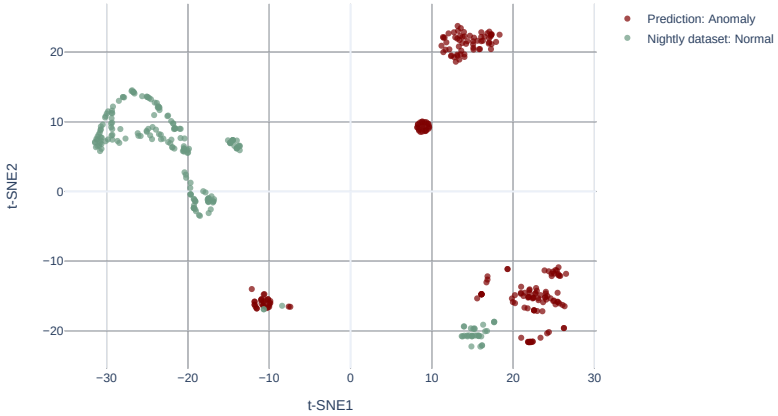
```
            model)-metrics.csv $(PREDICTIONS_DIR)$(TF_DIR)$(
            DATASET)-$(model)-predictions.csv;)
58      $(foreach model,$(MODELS),python3 src/models/
            evaluate_$(model)_model.py models/tf/$(model).
            model $(FEATURES_DATA_DIR)$(TFIDF_DIR)$(DATASET)
            -test.npy $(LABELS_DATA_DIR)$(DATASET)-test-
            labels.npy $(METRICS_DIR)$(TF_DIR)$(DATASET)-$(
            model)-metrics.csv $(PREDICTIONS_DIR)$(TFIDF_DIR
            )$(DATASET)-$(model)-predictions.csv;)
```

C. Results

C.1 Unsupervised Testing

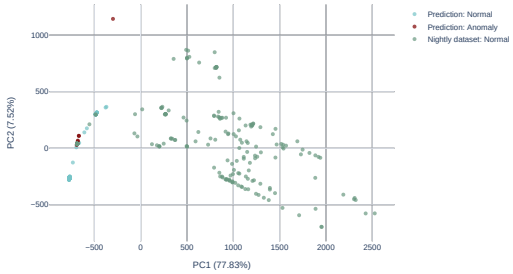


(a) Isolation Forest

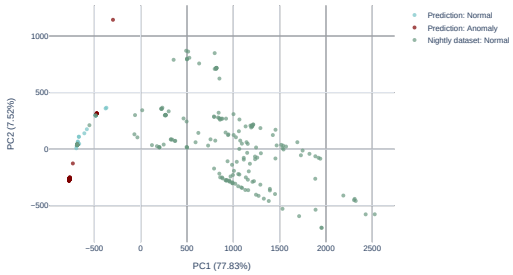


(b) Invariants Mining

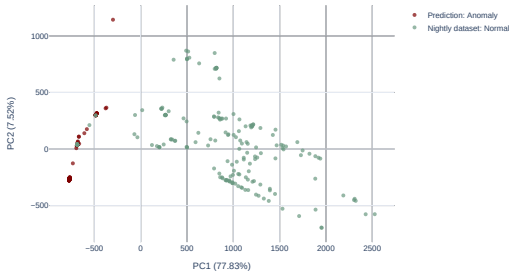
Figure C.1: Comparison of t-SNE applied on Isolation Forest (a) and Invariants Mining (b) predictions on the unlabeled Daily dataset. Green data points represent the normal datapoints as a point of reference, data points highlighted in blue and red represent predictions of normal or anomalous log sequence respectively.



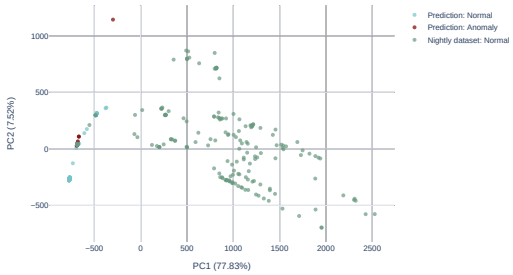
(a) Isolation Forest



(b) PCA



(c) Invariants Mining



(d) Log Clustering

Figure C.2: Comparison of PCA applied on Isolation Forest (a), PCA (b), Invariants Mining (c) and Log Clustering (d) predictions on the unlabeled Daily dataset. Green data points represent the normal datapoints as a point of reference, data points highlighted in blue and red represent predictions of normal or anomalous log sequence respectively.