# Log Anomaly Detection of Structured Logs in a Distributed Cloud System

David Nilsson

Albin Olsson

LUND UNIVERSITY

Department of Automatic Control

MSc Thesis
TFRT-6176
ISSN 0280-5316

Department of Automatic Control
Lund University
Box 118
SE-221 00 LUND
Sweden

# Abstract

As computer systems grow larger and more complex, the task of maintaining the system and finding potential security threats or other malfunctions become increasingly hard. Traditionally, this has had to be done by manually examining the logs. In modern systems, this can become infeasible due to either the large amount of logs or the complexity of the system. By using machine learning based anomaly detection to analyze system logs, this can be done automatically.

In this thesis the authors have researched the area of anomaly detection, and implemented an anomaly detection pipeline for a specific system. Three different machine learning based anomaly detection models were implemented, namely a clustering algorithm, PCA, and a neural network in the form of an autoencoder. These models were compared and evaluated with regards to a baseline error detection system, which was already in place for the target system. They were also compared against each other to find which models performed best, and in which circumstances. To compare the models, six different types of known anomalies were injected into the data.

When comparing the performances of the different methods, all of them were found to outperform the baseline system. In the first experiment, where the models were trained and tested using data from the same time period, PCA achieved the highest F1-score of 0.990. In the second experiment the models were trained and tested using data from separate time periods. In this scenario, the clustering algorithm outperformed the others, with an F1-score of 0.879. Both PCA and the autoencoder found many false positives, reducing their precision and thereby their F1-score.

# Acknowledgements

Thank you to all the people at Ingka Group Digital. Especially the Customer Web and Customer Admin development teams for supporting us throughout this project. A special thanks to our supervisor Gustav Hochbergs for always being available and helping us with the issues at hand. We would also like to thank Magnus Pettersson, the responsible manager at Ingka, for providing us with the opportunity of writing this thesis. Also, we would like to thank the other thesis workers at Ingka for providing interesting and insightful discussions about our theses.

Furthermore, we would like to thank our supervisor at LTH, Johan Eker, for providing great input and discussions about the project and the report. Finally, we would like to thank Karl-Erik Årzén for being the examiner of this thesis.

# Contents

# 1 Introduction

## 1.1 Background

In recent years there has been a dramatic increase in complexity in industrial computer systems. They are more advanced than ever, especially since cloud computing has become a popular concept in the industry. The inherent complexity of these systems not only makes them more prone to errors and incidents, but also makes issues harder to detect and repair. For this purpose, keeping system logs is a widely adopted practise for monitoring the state of applications and has become a key metric for determining the health status of a system. They record the state of the system by logging critical events during system execution. Traditionally, they have also helped developers find system bugs and fix potential issues in a system.

Chandola et al. [1] describes anomaly detection as the art of finding abnormal behavior in a set of data. Entries are considered anomalous if they do not behave according to the definition of normal, and the general concept has been used across many domains for over a century.

As system logs contain information about the state of a system, analyzing logs may help developers fix issues and detect abnormal behavior. This is a key activity in the aspect of computer security, where log entries may capture potential data leaks or similar issues. Furthermore, it may be used as an actual health metric for the underlying system. Finding potential errors in log files are conventionally done by manual analysis, where developers conduct manual searches for keywords indicating an error has occurred. It may also be done by matching rules to the log entries to find certain undesired patterns in the flow of logs. Because of the highly complex systems in industrial settings and the vast amount of logs the systems produce, manual analysis of this kind often becomes infeasible to perform in the long run. This method of analysis is also error-prone, making an automatic method for analysis preferable.

Automatic log analysis would mostly mitigate the need for manual inspection of logs. It could be done by rule based algorithms, which search for certain behaviors throughout the system execution. However, as an application is developed, its execution flow may change, along with the information available in the logs and the distribution between different kinds of messages. Using statistical machine learning based methods, one can analyze the execution flow of an application without explicit development of the anomaly detection component.

This thesis is done for Ingka Group Digital, which is part of the IKEA franchise. They are currently using rule based error detection system for various applications in their environment. Inherently, this system is limited in what kinds of errors it can find, as well as being hard to maintain and expand. For this reason, they are interested in exploring the possibilities of machine learning based anomaly detection for their systems.

## 1.2 The System

The system being used in this thesis is an opt-out system, which enables customers to opt-out from receiving various messages from Ingka. It covers various platforms, such as SMS and e-mail communication.

The system consists of a front end with graphical interfaces allowing users to pass the necessary information to successfully opt-out from receiving messages. The back end side of the system updates various databases throughout Ingka's infrastructure. This part is distributed in a cloud environment with multiple cloud services running simultaneously, and each interaction between the back end and other services are constantly logged for debugging and system monitoring. The overall system architecture can be seen in Figure 1.1.

The notion of *anomalies* and *errors* are important in this setting. The main concern would be if an error would prevent a customer from opting out from receiving messages from Ingka, as this would be a breach of EU law. However, a user being unable to opt-out could be caused by a plethora of reasons. For instance, the user themselves may have entered faulty information in the provided input fields. This in itself is not an error or an anomaly, but would still result in an error response from some of the API:s. Instead, an anomaly could be a sequence of similar error responses, which could indicate some kind of attack from the user, or an internal error regarding an API or the communication between services. It is important to distinguish that an error is not necessarily an anomaly, and, likewise, an anomaly does not necessarily consist of errors. As previously mentioned, anomalies are log messages or sequences of log messages that do not conform to the normal flow of logs in the system.

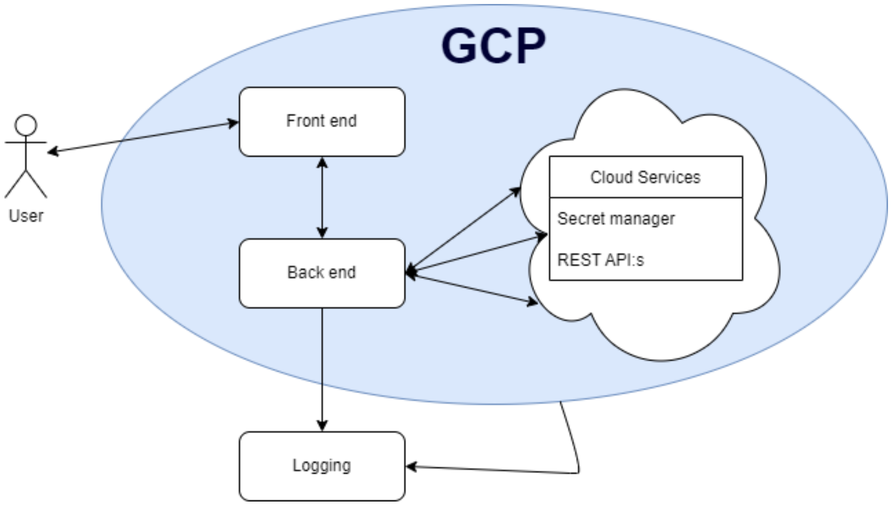


**Figure 1.1** The overall architecture of the opt-out system.

## Available data

Log data from the services deployed are generally stored for one month before being deleted. Throughout this time it is accessible for debugging potential issues within the system. However, for the sake of this project, logs will be kept in a separate storage, allowing multiple one month periods worth of logs to be temporarily stored and used to train the machine learning models. In the stored data, sensitive information such as IP and mail-addresses will be omitted to conform to GDPR laws. A cleaned log entry may look as follows:

```
{
    "insertId": "09jxvn",
    "jsonPayload": {
        "logMeta": {
            "endPoint": "CMA-OPTOUT",
            "result": "SUCCESS"
        },
        "message": "cma-optout response successful",
        "name": "opt-out-be",
        "pid": 3,
    },
    "severity": "INFO",
    "timestamp": "2022-03-01T00:00:11.938999891Z"
}
```

In the training phase, the data will be stored in a file provided to the models for easy access. If the models are actually deployed in production, the stored data will be discarded and the models will rely solely on streamed data instead.

### Baseline

The concept of testing is generally difficult in an anomaly detection application. The data sets are often large and unlabeled, meaning manual evaluation is infeasible and the algorithms have no baseline to base their decisions on. This project differs from this normal setting, as there is currently a hard coded and rule based anomaly detection in place which should be improved upon. This system analyses fixed windows of log entries from the past hour, considering specific static fields within the logs and determines if an anomaly has occurred based on the amount of messages of this kind from a certain user. This means there are windows labeled as anomalous according to this system which can be used to compare and evaluate the different methods.

## 1.3 Project objectives

Robust anomaly detection systems are key to maintaining a secure and healthy computer environment. Key aspects in industrial anomaly detection include time efficiency and accuracy. Time efficiency means the logs have to be processed in a timely manner, while accuracy refers to the ability to correctly categorize log sequences as anomalous or normal. There is a trade-off between a sensitive system with too many false positives and a system which neglects sequences that are in fact anomalous.

The objective of this thesis is to improve upon the existing system by exploring, evaluating, and comparing different machine learning based anomaly detection algorithms. The comparison will be conducted with regards to the standard machine learning evaluation metrics precision, recall and F1-score. The frequency of false positives, i.e, how often a model falsely labels a normal sequence as anomalous, will also be considered.

## 1.4 Thesis outline

The thesis begins with an introduction where the background and the objectives of the project are presented. It then moves on to Chapter 2 about cloud computing where information that will be relevant later in the thesis is presented. The information is relevant since the training and testing of the models will be conducted in a cloud environment. It is also relevant because the system whose logs are being analyzed is also run in a cloud environment. The thesis then moves on to Chapter 3 about machine learning, which provides a theoretical background of the machine learning concepts which will be used later in the thesis. Chapter 4, 5 and 6 (Log Parsing, Feature Extraction and Anomaly Detection) provides the background for the different steps in the anomaly detection pipeline. In Chapter 7 the method of collecting logs and performing the different steps in the anomaly detection pipeline as well as the implementation and testing of the models is presented. In Chapter 8 the results from the different tests are presented and in Chapter 9 these results are discussed. Finally, the conclusions drawn from the research are presented in Chapter 10.

## 1.5 Individual contributions

Most of this thesis was done in cooperation between both authors. Generally, David focused more on researching and implementing the clustering and PCA algorithms, as well as the testing pipeline used. Albin focused more on researching different parsing techniques, as well as implementing the autoencoder. There was an equal contribution regarding the literature study and the writing of the report itself.

## 1.6 Related work

Anomaly detection has been used for decades across various domains, and cloud computing is no exception for this type of application. There has been a lot of research done for anomaly detection in cloud computing systems. One example of this is in [7], where the authors propose a method for both finding and explaining anomalies in cloud environments. Their method uses a hybrid single/multiple-threshold anomaly detection method relying on an LSTM-Autoencoder. These concepts will be further explained in Chapter 3. Their proposed method also includes an anomaly explanation module which helps identifying the root cause of the anomaly.

Continuous integration and continuous deployment (*CI/CD*) is vital in large scale software development for maintaining a healthy software environment. *Cross functional collaboration between development and operations (DevOps)* is a frequent term when talking about continuous styles of development. Hrusto [15] presents a solution for handling feedback and alerts from operations to development using a smart filter, which utilizes data from system operations to detect anomalous system behaviors through machine learning based approaches. They especially explore the possibilities of using deep learning to analyze multivariate time series data from a distributed computer system.

There has also been a lot of research where the performance of different anomaly detection methods are compared. In [14] the authors compare both unsupervised and supervised anomaly detection methods using publicly available production log datasets. This thesis will only focus on the unsupervised methods since those are much more relevant to this thesis. The methods *Clustering*, *PCA* and *Invariants Mining* were compared, with *Invariants Mining* performing best among those three. These methods are further explained in Chapter 6. In another paper, [2], other anomaly detection methods were compared, focusing on deep learning methods.

These comparisons were done on the same datasets as the comparisons in [14]. The unsupervised methods compared were *LSTM*, *Autoencoder* and a *Transformer*. All of these methods were compared both with and without the use of semantics in the log messages keys. Both with, and without, the use of semantics the *LSTM* model performed better than the other methods.

# 2 Cloud Computing

In recent years, cloud computing has become increasingly common, and important, in the industry. It helps deliver computing services, such as servers, storage, networking, and software, in an on-demand manner over the internet [23]. Ingka is no exception to this phenomenon, and they have a great fraction of their systems running in Google's cloud service *Google Cloud*. This is one out of many cloud providers, such as Amazon, Microsoft, and IBM. This thesis will focus on Google Cloud since that is what Ingka is using for their applications.

*National Institute of Standards and Technology* (NIST) defines cloud computing as "*a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction*" [21]. There are five main characteristics of cloud computing, namely on-demand self service, broad network access, resource pooling, rapid elasticity, and measured service.

**On-demand self service** refers to a customer in need being able to acquire resources, such as CPU time, without human interaction.

**Broad network access** means these resources should be delivered over the Internet and be used by platforms at the user's end.

**Resource pooling** means multiple customers can be served simultaneously through dynamically scaled resources depending on customer demand.

**Rapid elasticity** means a customer should be able to rapidly scale their use of resources in accordance with the previous point.

**Measured service** refers to the problem of measuring the resources used by each individual customer.

## 2.1 Cloud deployment

Simply put, cloud computing means transforming IT infrastructure into a utility. It allows customers to access servers and operating systems maintained by a cloud provider [16]. There are four main service models that categorize the utilities that are provided [3].

***Software as a Service (SaaS)*** SaaS means cloud customers deploy their applications in an environment that is hosted and maintained by the cloud provider. The customers do not have control of the cloud infrastructure and various customers' applications are commonly organized in shared virtual environments. An example of a SaaS application would be Gmail.

***Platform as a Service (PaaS)*** PaaS allows customers to utilize the cloud platform for application development, whereas SaaS only allows customers to deploy already finished applications. This means PaaS has to include various tools for software development, which is not accessible through SaaS. A PaaS example from Google would be Cloud Run, which is described in Section 2.3.

***Infrastructure as a Service (IaaS)*** IaaS allows customers to utilize the infrastructure of the system they are provided with. To this end, cloud providers commonly utilize virtual machines to isolate the machines from both the hardware of the physical machine and other virtual machines (VMs) running on the same physical computer. Google's compute engine, which is also described further in Section 2.3, is an example if IaaS.

***Data storage as a Service (DaaS)*** DaaS means customers are presented with the possibility of utilizing virtual storage within the cloud. This can be seen as a special case of IaaS, and an example could be Google Drive.

## 2.2 Google Cloud

Google, as several other IT companies, offer cloud services to both industrial and private customers. They offer virtual resources on physical hardware, such as computers and hard drives, as well as various virtual machines. The resources are contained within Google's data centers across various regions all over the world. The platform offers and utilizes many services developed by Google to enable the infrastructure that suits the need of the project to be deployed. The available services cover many areas, such as API management, AI and machine learning, and the base for running custom applications within the cloud infrastructure.

The foundation of service deployment in Google Cloud lies in creating projects. A project acts as the main control unit which specifies what will be achieved with the deployment, as various metadata describe the applications. Many resources can be deployed and allocated within the same project, and they can easily communicate internally. Once a resource is deployed in Google Cloud, it can be monitored and accessed by either the Google Cloud Console, which is a graphical interface for managing projects. There is also a command line interface, allowing the customer to manage their projects from a terminal window[11].

## 2.3 Virtualization

As previously seen, virtualization is an important tool to manage cloud infrastructure. It allows deployment of various machines and applications to the same physical computer. These virtual machines do not have explicit access to physical hardware, and they are also isolated from other resources being allocated on the same device.

Google Cloud offers various alternatives for virtualization, the main two being *Compute Engine* and *Cloud Run*. Compute Engine is a service that lets the customer deploy a custom VM to Google's data center. There are a number of presets specializing in various aspects of computing, such as general purpose, high memory, and compute-intensive workloads [10]. An alternative to this would be Cloud Run, which allows customers to deploy containerized applications (e.g., through the use of *Docker*) to a platform that automatically scales the resource usage depending on the needs of the application [9].

### Terraform

Virtual resources are commonly managed through automatic orchestration software, such as Google's Cloud Run. *Terraform* serves as an alternative to this. It is a tool for managing resource infrastructure with regards to, for instance, virtual machines, storage, and networking. The main way of interacting with Terraform is through its scripting language, which tells it what resources to use, what plugins to install, and what initialization steps are required to perform [12].

### Docker

Docker is a platform for packaging and running applications in *containers*, which are isolated environments which runs completely independently from the operating system of the host machine. The containers contain all the resources necessary to run the applications being developed, meaning the underlying file system and host machine dependencies do not matter. The base unit for creating containers are Docker *images*, which acts as an instruction for creating the container. It is often built from a base image, e.g, containing an operating system, in which the application and dependencies are added. Once an image is created, it can be instantiated in the form of a *container*, which can run the programs loaded into the image [4].

# 3

# Machine Learning

Machine learning is an area of computer algorithms that learns, adapts and improves upon itself through experience. By analyzing data the algorithms builds models that can make decisions or predictions based on patterns in the data. These algorithms are widely used where implementing a fixed, conventional algorithm would either be too complex, time consuming or simply not possible [8]. Applications of machine learning are used in a lot of different fields, such as, speech recognition, medicine, credit card fraud, or computer vision [29]. Depending on the goal of the algorithm and the nature of the data available, there are some different approaches to machine learning. The basic approaches are:

**Supervised learning** is when the algorithm has access to a labeled data set in the training stage. A labeled data set is data that contains the desired output of the model, mapped to each data entry. The goal of the algorithm is to be able to create mappings between the data entries and their respective labels so that when presented with a new entry, the model can predict the corresponding label [8].

**Unsupervised learning** is when the algorithm does not have access to a labeled data set in the training stage. Instead of predicting a label associated with each entry, unsupervised models try to find structures in the data, often based on similarities between entries. One way of doing this is clustering. By grouping similar entries together, clusters are formed. Information about a new data point can then be acquired by how it is placed in relation to the already defined clusters [29].

**Reinforcement learning** is a common field of machine learning for, i.e., game theory, simulations, and information theory. It focuses on how a virtual agent should act in certain scenarios in an environment to maximize some reward metric [29]. E.g, in an arbitrary game against an opponent, losing may yield a negative reward, a tie may yield a neutral reward (0), and winning a positive one. This reward is given as feedback to the machine learning agent to indicate whether it performed well or not.

## 3.1 Metrics

### Loss functions

Training machine learning models, especially deep learning models, generally means minimizing the error measured between the automated prediction from the model and a ground truth value. This error is often called *loss*, and is the difference in performance of the model compared to the expected values. If the model predicts something that differs greatly from the expected value, the loss will be big, indicating that the model needs to alter its weights to counter this effect. Ravindra Parmar [26] describes various types of loss functions and their appropriate applications

Loss functions can roughly be divided into *regression* and *classification* losses.

**Regression losses** are used in tasks where regression is used to predict continuous values. Thus, distance based loss functions are common. For instance, the *Mean Square Error* (MSE), measures the average squared distance between the prediction and the expected value in accordance with Equation (3.1). $L$ is the loss itself, $n$ is the number of data points considered, $i$ is the $i:th$ data point in the data set, $y_i$ is the expected value for the $i:th$ data point, and $\hat{y}_i$ is the predicted value for the $i:th$ data point. The fact that the distance is squared means that large deviations from the ground truth will be penalized.

$$L = \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{n} \tag{3.1}$$

Other common loss functions include *Mean Absolute Error* (MAE), as seen in Equation (3.2), and *Mean Bias Error* (MBE), as seen in Equation (3.3).

$$L = \frac{\sum_{i=1}^{n}|y_i - \hat{y}_i|}{n} \tag{3.2}$$

$$L = \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)}{n} \tag{3.3}$$

MAE roughly measures the same concepts as MSE, as it does not take the direction of the loss into account. However, it is more reliable when it comes to occasional data set anomalies, since it does not use the square as MSE. In extension, the less common MBE loss function performs the same mathematical operations as MSE, but does not square the error. In practice, this means the direction of the error will matter, and could affect the weights of the machine learning model.

**Classification losses** are used in classification problems which involve predicting values from a finite set of options. For instance, this could be classifying images of cats and dogs, or handwritten digits between $0-9$. *Cross entropy* might be the most common loss function for classification problems, and is characterized by the fact that it penalizes predictions that are confident, i.e., close or equal to zero or one, but wrong. The formula for cross entropy can be seen in Equation (3.4). From this equation it becomes clear that when the actual label is one, the second half of the equation vanishes. When the ground truth label is zero, the first half does.

$$L = -(y_i log(\hat{y}_i) + (1-y_i) log(1-\hat{y}_i)) \tag{3.4}$$

## Performance measures

Goodfellow et al. [8] describes various performance metrics which are explained in this section. To determine the performance of the model being trained and, thereby, how useful the model actually is, a performance measure is required. There are various performance metrics, although some are more common than others. *Accuracy* is a common performance metric, although it is not useful in all scenarios. It effectively measures the fraction of correct classifications from a system, and can be written as seen in Equation (3.5). $TP$ stands for *True Positives*, $TN$ for *True Negatives*, $FP$ for *False Positives*, and $FN$ for *False Negatives*. It lacks in the aspect of reliability. If the model tries to find rare entries, the model can achieve high accuracy by hard coding that this specific type of entry is never present in a data set. Of course, this would not be a useful model, but the accuracy metric would still show high values.

$$a = \frac{TP+TN}{TP+TN+FP+FN} \tag{3.5}$$

To solve the problems of the accuracy metric, there are alternative metrics such as *precision* and *recall*. Precision is the fraction of predictions that were correct, according to Equation (3.6). Recall, on the other hand, is the fraction of true data points that were detected, according to Equation (3.7). These metrics measure the opposite aspect, meaning a model predicting all values as *false* would achieve 100% precision but 0% recall.

$$p = \frac{TP}{TP+FP} \tag{3.6}$$

$$r = \frac{TP}{TP+FN} \tag{3.7}$$

Precision and recall are commonly plotted against each other as they tell the user a great deal about the performance of the model. However, it is not always practical to use plots to describe how these metrics cooperate. Instead, they can be combined into a single value representing the performance of the model, in the form of an *F1-score*, which can be seen in Equation (3.8), where $p$ is the precision and $r$ is the recall.

$$F1 = \frac{2pr}{p+r} \tag{3.8}$$

## 3.2 Machine learning approaches

This chapter will mainly focus on unsupervised methods since those are most relevant to the project.

### Clustering

As mentioned in the last section, clustering refers to grouping similar entries together based on their similarities, which lays a foundation for analyzing the data further. There are many different approaches to clustering, both in regards to how these clusters are defined as well as how you use the information gained to draw conclusions [33].

**K-means Clustering** is a clustering algorithm that tries to minimize the mean squared error (MSE), described in Section 3.1, of clusters. The MSE between the mean vector of a cluster and a data point is regarded as the distance between them. When building the clusters, the algorithm starts with picking $K$ random samples from the training data as initial cluster mean vectors. The algorithm then, for each data point, calculates the MSE in regards to each cluster and adds the data point to the closest cluster. When all data points belong to a cluster, new cluster mean vectors are calculated and the algorithm is run again until the new mean vectors are identical to the old ones. Since this can result in very long execution times, a threshold of either minimum change or maximum iterations is often used [33] .

One of the drawbacks of k-means clustering is that you have to pre-define the amount of clusters within the data. **Hierarchical clustering** is an approach where the number of clusters does not have to be pre-defined. Hierarchical clustering methods build clusters by recursively either combining or splitting clusters until there is a certain minimum distance between clusters. Figure 3.1 shows an example of agglomerative hierarchical clustering where each data point is initially considered a cluster. Clusters that are close to each other are then combined until there exists a certain minimum distance between clusters. This distance can be computed in a multitude of ways. One of these ways is to use MSE just as in K-means clustering. The other way of doing it is called divisive hierarchical clustering, then you start with all of the data points in the same cluster. The clusters are then split until there exists a certain minimum distance between each cluster [19].

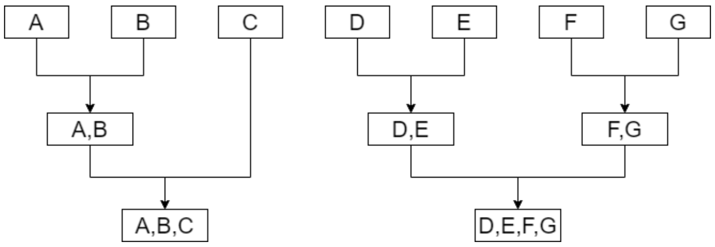


**Figure 3.1** An example of agglomerative hierarchical clustering where each letter corresponds to a data point and each box corresponds to a cluster

### Principal Component Analysis

Principal Component Analysis (PCA) is an algorithm used for dimensionality reduction and it aims to represent higher dimensional data in lower dimensions while still preserving as much of the characteristics of the data as possible. It builds a new coordinate system based on $K$ principal components, i.e, the coordinate system is represented in $K$ dimensions. By calculating the variance of the original components the algorithm finds which components have the most integral part of representing the data [14]. This new coordinate system can be used in a variety of ways to determine properties of a new data entry. It can for example be used in outlier detection by analyzing the properties of a new entry with regards to the found principal components. PCA is closely related to *Singular Value Decomposition* (SVD) which is commonly used to decompose matrices. Furthermore, it is common to use SVD for PCA, and it is possible to use PCA for SVD [35].

Figure 3.2 shows a simplified example of how PCA could work in practise. In this example there are two dimensional data points, represented by the light blue dots. In this case, they should be represented in one dimension along the dark blue line.

## 3.3 Deep learning

Deep learning is a field of computer science that tries to mimic concepts in biological brains to allow computers to learn from examples of data [6]. This is mainly done through the use of Artificial Neural Networks (ANNs). They consist of layers of artificial neurons, or nodes, which are each connected to at least one node in the next layer of the network. A node also contains information about how to interpret each of its inputs and at what threshold its output value should be passed along to the next node in the network. Each network consists of an *input layer*, which takes input values and passes it along in the network, and an *output layer*, which receives data processed from the earlier stages of a network and processes it with regular neurons. Between the input and output layer there might be intermediate layers, or *hidden layers*. An example of a neural network can be seen in Figure 3.3. A neural

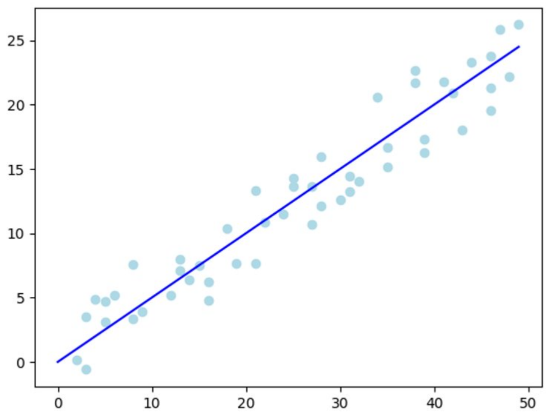


**Figure 3.2** Visualization of PCA from two dimensions to one.

network with three or more layers (including the input and output layer) can make out an algorithm in the realm of *deep learning* [17].

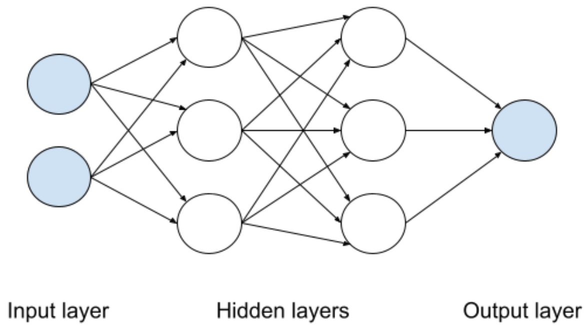


**Figure 3.3** An example of a neural network with an input layer, two hidden layers, and an output layer.

[6] explains that deep learning is part of the machine learning realm, but that it differs greatly from traditional methods. Many ordinary machine learning algorithms utilize labeled and structured data. This means there are predefined fields embedded in the data for the algorithms to learn, and that there are labels present, acting as a ground truth for the models to compare with their predictions. Deep learning mitigates some preprocessing steps that are typically necessary in machine learning.

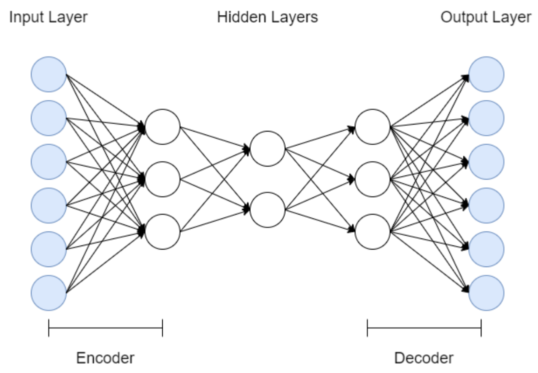


**Figure 3.4** An autoencoder architecture with three hidden layers. The input layer and the first hidden layer make out the encoder part of the network. The last hidden layer and the output layer make out the decoder part of the network.

Deep learning models also require less explicit knowledge from human experts as they are able to automatically identify the most important features in the data.

Some common neural network architectures are:

**Feed forward** is a network architecture where the information is passed from the input layers, through each hidden layer, and finally to the output layer. The information does not pass through any of these layers more than once, and it is only passed *forward* in the network. In Figure 3.3 the information would flow from left to right [25].

**Autoencoders** are special neural networks designed to learn efficient encodings of the input data. The training data is passed as in a normal neural networks, but are passed along to one ore more hidden layers with smaller dimensions than the input layer. Once the data is represented in fewer dimensions, the network then reproduces the data in the same dimensional space as for the input layer, with the goal of minimizing the reconstruction error [27]. An example of an autoencoder architecture can be seen in Figure 3.4

**Recurrent neural networks** work on sequential data, and it is able to remember what it has encountered previously to help improve future predictions. The states in the network get progressively updated each time data is read, and each time step also takes the network states of the previous time step into account. The architecture of a recurrent neural network can be seen in Figure 3.5. Regular recurrent networks have a very short term memory, meaning it can not remember very long sequences of data. This problem is mitigated using a *Long Short Term Memory* (LSTM) network, which is a special kind of recurrent network with extra components for handling longer input sequences. These components are called *gates*, and they result in a

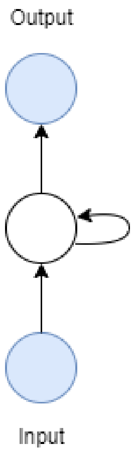


**Figure 3.5** A visualization of a recurrent neural network architecture. The input is in the form of sequential data, which results in a prediction for each time step in the output step.

longer memory for the neural network. The first gate, *cell state*, is the effective long term memory of the network. The *forget gate* erases information in the cell state that is no longer needed based on the input from the current timestamp and the cell state from the previous timestamp. The *Input gate* determines what information is added to the cell state based on the input from the current timestamp and the cell state from the previous one. Finally, the *output gate* extracts the information from the cell state [25].

# 4

# Log Parsing

Recording system events is a crucial act to maintain a healthy computer system in an industrial environment. It is common to store vital system information in logs for developers and engineers to debug potential problems. Although there is much information embedded within logs, how to effectively analyze them is not a trivial question. Manual analysis will quickly prove tedious and infeasible considering the vast amount of data generated each hour. Thus, machine learning techniques have recently been on the rise in log file analysis. For this to work, the initial step is to prepare the log data at hand for such algorithms through log parsing.

It is common for log messages to be generated through log-statements from the source code. Generally, they are informational strings of characters, often mixing constant strings with variable runtime values. The goal of log parsing is to construct a log template consisting of the constant string, along with a list of parameters consisting of the various variable information.

Zhu et al. [34] presents, explains, and evaluates a multitude of log parsing algorithms. They walk through each algorithm thoroughly, explaining their characteristics, techniques, and how they perform compared to other techniques on various data sets. The evaluations are mainly based on *accuracy*, *robustness*, and *efficiency*. The top performing log parsing algorithm from their research was *Drain*, *IPLoM*, and *AEL*, which will be described in this chapter. Their experiments will be described further in Section 4.4.

## 4.1 Drain

Drain was proposed by He et al. [13] and aimed to reduce the manual work in traditional log parsing strategies. It does not require any access to source code or any other resources, other than raw logs, and is a *fixed depth tree based online log parsing method*.

### Drain pipeline

Drain is an online parser, meaning it handles log entries one by one in a streaming manner. When new messages arrive, the algorithm either assigns it to an existing log group or creates a new one. To speed up the process of searching through existing log groups, Drain effectively utilizes a fixed depth search tree which reduces the amount of log groups to explore. Furthermore, the fixed depth means all *leaf* nodes in the tree are found at the same depth.

The tree itself consists of a *root* node, which is at the top of the parsing tree. At the bottom, there are *leaf* nodes, which contain lists of log groups. The log groups are described by a log event, which is the constant part in a log entry, and a log ID, which records the IDs of all the log messages in that group. In between the root node and leaf nodes there are *internal* nodes, which encode specific rules which guide the search through the tree. These rules could be the length of log messages stored further down that path, or specific parameters found in those log messages.

**Preprocessing** is the first step for incoming logs. The preprocessing is done in the form of domain-specific regular expressions. The users provide regular expressions which Drain will use to remove matching fields from the log entry. This is mainly done to reduce redundant information and potentially improve parsing accuracy.

**Traversing the parsing tree** is the second step of the pipeline. Drain starts its search from the root node of the tree. The first step in traversing the tree is to search through the first layer of internal nodes, which holds information about message length (measured by the number of tokens in the entry). It will select the path matching the number of tokens in the incoming log.

The next layer in the tree consists of significant constants within log entries, which Drain assumes occur within the first few positions of a message. Each constant is represented as an internal node. If any of the initial tokens contain digits, they will be represented by "$*$" in the tree to restrict the size of the tree.

Finally, Drain has found a leaf node containing a list of log groups. It adds the incoming log message to the most similar log event of each group. The similarity is determined by *simSeq*:

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

where $seq_1$ and $seq_2$ denote the incoming log message and the log event respectively, n is the number of tokens, and *equ* is defined as:

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

where $t_1$ and $t_2$ are tokens in $seq_1$ and $seq_2$ respectively.

***Updating the parsing tree*** Finally, the log group found in the previous step will be appended with the log ID of the incoming log entry and the group log event will be updated. If no suitable group is found, Drain will create a new log group containing only the log ID of the incoming log and the log event being exactly the incoming message itself.

## 4.2 IPLoM

*Iterative Partitioning Log Mining*, or *IPLoM*, was proposed by Makanju et al. [20] and excels at finding event type patterns in logs. It operates through iteratively partitioning a batch of log messages, converging towards monotone clusters with regards to the message format with each iteration.

### Partitioning

The partitioning goes through three separate steps to find all possible log lines.

1. The first step is to partition the set of log messages with regards to *token count*, meaning the log messages are grouped based on the number of tokens in the message. It is assumed that log messages that derive from the same print-statement in the source code have the same format and, inherently, the same number of tokens.

2. Next, the algorithm further partitions the log messages based on *token position*. It creates clusters based on the token index $i$ with the fewest number of unique tokens. These clusters are populated with the different log variants where the tokens at index $i$ are the same.

3. Finally, the partitioning is based on *bijective relationships* meaning there is a $1:1$ ratio of adjacent tokens. This typically means there is a strong relation between these tokens and the messages containing them are therefore grouped together. However, before passing through this partitioning step, the clusters are evaluated. If they are considered good already, this step is mitigated entirely.

The final step of the algorithm is to find the cluster descriptions, or line format of the log entries within that cluster. Each cluster is assumed to contain log entries derived using the same log format. This is done by counting the number of unique tokens in each token index of every entry within the cluster. If this count is one, the token is considered a constant and represents itself in the description. If it is greater than one, however, it is considered a variable and the index is shown as a wildcard token, "$*$".

## 4.3 AEL

The task of abstracting execution logs (*AEL*) means transforming log lines into execution events which can be used for analysis. Log lines are generally generated through output statements in the source code of an application, and log entries from the same output statement are therefore generated by the same execution event. Furthermore, log entries with identical static parts and similar structure in their varying parts most likely derive from the same line of source code.

*AEL* operates by detecting similarities in various log lines and parameterizing them. It performs three main steps:

1. The first step is to anonymize the log entries by finding tokens corresponding to varying parts of the message. This is done by looking for token pairs indicating an assignment using the equals sign or phrases indicating assignments with words like *is* or *are*. Variable tokens are replaced with a wildcard token, in this case "*$v*".

2. The second step is to tokenize the resulting log lines from the previous step by separating them into groups based on the number of tokens in each line. The group names are tuples consisting of the number of static words and the number of variable tokens of the messages in the group.

3. The third and final step is to categorize each group and abstract them to the correct execution event. This means matching the tokenized log messages in each bin, labeling similar ones with the same execution event. Entries that do not match any previous formats get a new and unique label.

## 4.4 Performance of log parsing techniques

Zhu et al. [34] defines the parsing accuracy as the ratio of correctly parsed log entries in relation to the total number of messages. They show that, on average, the most accurate log parser technique they experimented on was *Drain*, which reaches a high accuracy ($> 90\%$) on a majority of the data sets. Other high performing algorithms were *IPLoM* and *AEL*.

The robustness of the parser is determined by its consistency across different data sets. A robust method should be consistent when exposed to various types of logs from different domains, as well as for different amounts of logs to process. They conclude that *Drain*, *IPLoM*, and *AEL* are robust since they all have high average accuracies. However, the two latter methods have great variances in their performances. *Drain* and *AEL* show the best robustness when it comes to variance in volume of logs, although *Drain* has higher overall robustness than all other methods.

The efficiency is determined by the time required to finish parsing the entire batch of logs. Out of the parsers tested, *Drain* and *IPLoM* are the most efficient and their execution times roughly scales linearly to the volume of logs being handled.

There are three recurring methods throughout these tests, namely *Drain*, *IPLoM*, and *AEL*, which are all among the top performing methods in the aspects tested. Furthermore, Meng et al. [22] propose *LogAnomaly*, a state-of-the-art anomaly detection framework for log files. They use a log template representation called *Template2Vec* which allows them to capture semantic information in the logs. *LogAnomaly* itself is discussed further in Chapter 6.

## 4.5 This project

There are many similarities between the algorithms described so far. They all try to group similar log messages together by using trees, clusters, or both. In this project, fast and reliable methods are of interested for processing the logs. Since Drain is the top performer in most categories, this is the parser used in this thesis.

# 5

# Feature Extraction

Traditional log messages consist of strings of characters, meaning they need to be converted to numerical features before they can be used by an anomaly detection algorithm. The message is typically converted into a template and features via a log parser, which is described in the previous chapter. Using this template, the act of feature extraction is performed in two main steps, namely *partitioning* and the *feature extraction* itself. This is described in [2].

## 5.1 Partitioning

After having generated a log template from a parser, log entries are commonly grouped in various ways. For instance, they can be grouped by some ID, such as for tasks or sessions, present within the data or by time. Chen et al. [2] describes the main ways to partition chronologically sorted logs into sequences.

**Fixed partitioning** means partitioning the logs into groups of a predetermined size, usually determined by the timestamp in the logs. Figure 5.1 shows how a sequence of numbers can be split using fixed partitioning. The window samples the entries it can fit within its length, before jumping to the first entry it has not covered in previous window positions. In practise, this means there is no overlap between partitions.

**Sliding partitioning** Sliding partitioning means grouping logs based on a sliding window defined by a window size and a step distance, which is generally smaller than the window size. Figure 5.2 shows this principle. The window samples entries within its length, before sliding to a new position, specified by the step distance. This means the groups are overlapping, meaning the same entries are present in multiple groups, effectively resulting in more data than in the original data set.

**Partitioning based on identifiers**, or *session windows* [14], means partitioning logs based on a common field contained within a sequence of logs. Figure 5.3 shows this type of partitioning. The concept of windows is not relevant in this type of

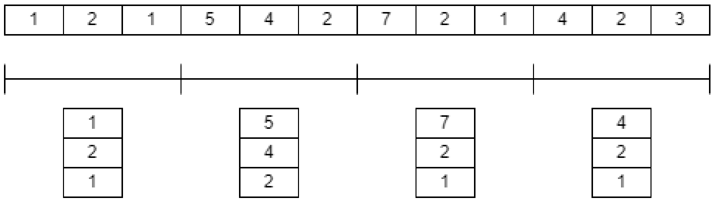


**Figure 5.1** Illustration of how fixed partitioning divides a dataset into windows

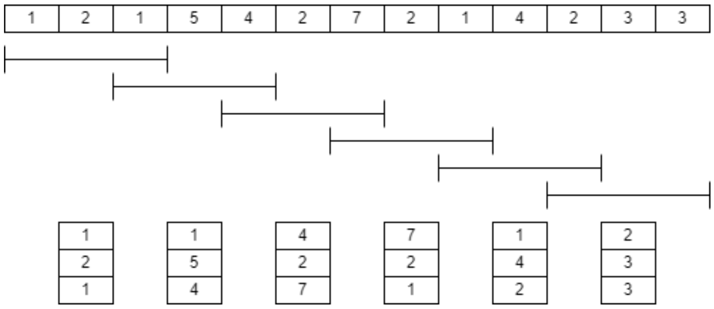


**Figure 5.2** Illustration of how sliding partitioning divides a dataset into windows

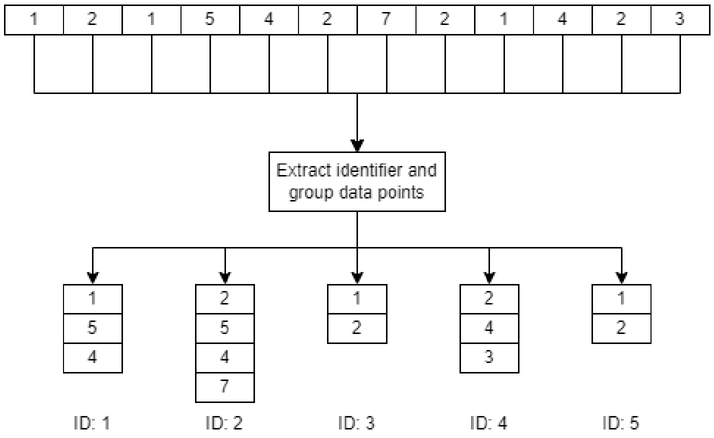


**Figure 5.3** Illustration of how feature based partitioning divides a dataset into windows

partitioning. Instead, each unique value for a specified field is grouped together. This often indicates they belong to the same task or were created in the same context. A key characteristic of these groups are their varying lengths.

## 5.2 Extracting features

After having grouped the log entries, many machine learning based anomaly detection techniques use an *event count vector* for each sequence, resulting in a matrix for all sequences [14]. The rows in this matrix are the event count vectors, in which each dimension is represented by a specific log message. The number in that dimension is the number of occurrences of that message within the sequence.

Another, more elaborate, method is described in [2] and is based on deep learning methods aimed to learn the semantics of the log messages. Here, the words in a message are transformed to a numerical representation using *Word2Vec* methods such as *FastText* or *GloVe*. The output can be aggregated to represent the semantic vector of the message.

## 5.3 This project

Over the course of this project, multiple anomaly detection algorithms have been compared. These algorithms, which are discussed in detail in Chapter 6, require different methods for data partitioning depending on what they analyse.

# 6

# Anomaly Detection

Detecting anomalies or outliers in data can be done in a variety of different ways, and to decide which one is best for the problem at hand is not always trivial. Which method that is best suited depends on factors such as the nature of the input data, type of anomalies, the availability of labels and the desired output of the model [1]. Defining what is and what is not an anomaly is not an easy task, the authors of [1] uses the following definition: "Anomalies are patterns in data that do not conform to a well defined notion of normal behavior" and classifies anomalies into three different categories.

**Point anomalies** are the simplest kind of anomalies. They refer to individual data instances which by themselves can be considered anomalous. These are usually the easiest to detect since they usually stand out from the rest of the data points.

**Contextual anomalies** are data instances which are considered anomalous in the context of the instance, but do not have to be considered anomalies when compared to the rest of the data.

**Collective anomalies** are not anomalies in the same sense as the two previous types. Collective anomalies are collections of data instances where individual data points do not have to be anomalies. But when evaluated as a group the whole collection is deemed anomalous.

Even though these different types of abnormal behaviors can be categorized, what is, and what is not, considered normal behavior might change over time. This might be because of new features added to the system or other system updates. Since there will be no available labeled data in this thesis, the focus will mainly be on unlabeled anomaly detection techniques. Acquiring labeled data is often hard and expensive when it comes to anomaly detection. Labeling has to be done manually by a human who has to be an expert at the system at hand. In some cases it can be hard, even for an expert, to find all anomalies in a data set since there may exist anomalies that are not known beforehand. According to the authors of [14] unsupervised techniques are therefore often far more applicable in real world environments.

## 6.1 Traditional unsupervised anomaly detection techniques

**Clustering based techniques** groups similar data points into clusters and evaluates new data point entries by placing them inside or outside already created clusters. There are a few different approaches to how you distinguish anomalies from normal data. You can assume that all data points that do not belong to a cluster is an anomaly, but this could result in a situation where anomalies form a cluster and therefore would not be caught by the model. To address this issue you could instead calculate the density or size of the cluster where it is placed. If the density or size is under a certain threshold the cluster is considered anomalous. An implementation of a clustering based technique proposed by [18] called *LogCluster* is divided into two different training phases. In the first phase event count vectors are clustered using agglomerative hierarchical clustering and a representative event count vector is calculated for each cluster. The second phase is used to further adjust these clusters and possibly create new ones by adding new event count vectors one by one. The model is then ready for use. By calculating the distances to nearby clusters, a new event count vector can be considered normal or anomalous.

**Nearest Neighbor based techniques** based techniques work similarly to the clustering techniques. Instead of creating and calculating distances to the closest cluster, nearest neighbor techniques calculate the distance to the nearest neighbor or, alternatively, the density around a data point. This means that it requires less computing power in the training phase but it is quite computationally costly in the testing phase. The performance heavily relies on how the distance is defined for the problem [1].

**Spectral anomaly detection techniques** are techniques where the data is transformed into a lower dimensional space where the characteristics of the normal instances, and therefore also the anomalous instances, are more clearly defined. One frequently used approach is to use Principal Component Analysis (PCA) described in Section 3.2 [1]. He et al. [14] uses the PCA algorithm to construct two different sub-spaces, one normal space and one anomalous space where the normal space is built by the first $K$ principal components and the subspace built by the rest of the components is considered anomalous. By projecting a new entry onto the normal subspace and checking the length of the projection into the anomalous subspace an entry can be reported as anomalous. It is anomalous if the length in the anomalous subspace is larger than a certain threshold. The authors use squared prediction error (SPE) to represent the length of projections.

**Invariants Mining** is a technique where program invariants are mined. A program invariant is a linear relationship that always hold true during program execution.

Logs that can be grouped together into sessions, e.g, by the use of session IDs, can represent the execution flow of that session. The invariants mined represent the normal execution flow of the system under different circumstances, such as different workloads or different parameter values. This method uses event count vectors to generate an event count matrix. Using singular value decomposition the number of different invariants to be mined can be calculated. These invariants are then mined using a brute force search algorithm. When a new log sequence arrives it will be checked against the mined invariants and will be reported as anomalous if it breaks the rules of the invariants [14].

As mentioned in [2] traditional and statistical models, even though they have improved over the years, still have some limitations. They offer *insufficient interpretability* which means that they often only flag a sequence as anomalous without providing any more information about the anomaly. They also have *Weak adaptability* which means that they do not respond well to changes in system behavior. Some models even have to be retrained from scratch when new system functions and features are added. They also rely heavily on *handcrafted features* which means that the model often has to be specifically tailored to the specific system.

## 6.2 Deep learning based anomaly detection

To combat the limitations of traditional and statistical methods [2] proposes to use deep learning to detect anomalies. Deep learning often applies neural networks such as recurrent neural networks (RNNs) or convolutional neural networks (CNNs). A lot of research has been done in recent years regarding the application of deep learning in anomaly detection, and in [2] the authors compare some of the deep learning methods for anomaly detection.

### Autoencoder

One way to perform anomaly detection using deep learning is to use autoencoders. Chen et al. [2] deploy an autoencoder to learn the representation of normal log sequences. As mentioned in Section 3.3, the autoencoder first encodes the data by using neural network layers of smaller dimensions than the original data. It then decodes and tries to represent the data again in its original dimension. This is visualized in Figure 3.4. By comparing the input and the output of the model, i.e, the original log sequence and its encoded and decoded version, the data can be considered an anomaly if the difference is too big.

### DeepLog

DeepLog is a deep learning neural network proposed by Du et al. [5], which is an approach for anomaly detection that takes inspiration from natural language processing (NLP). Just like a language, system logs follow certain logical patterns and control flows. By viewing log sequences as sentences and entries as words the log data behave very much like a language. A deep learning network is used to model this. A Long Short-Term Memory (LSTM) network allows DeepLog to learn the patterns of normal execution and flag anomalous events. By analyzing the log sequence DeepLog creates a probability distribution of the next log entry to appear. If the new log entry does not meet the probability threshold, it will be considered anomalous.

DeepLog first parses free text log entries into structured, sequential data. They extract a log key from each data entry. A log key is the type of the message and refers to the constant string of the log message with variables removed. Variables will be abstracted with an asterix (*) in the log key. The parameter values for each log is stored in a separate vector which will be used in addition to the log key

DeepLog contains three main components: log key anomaly detection model, parameter value anomaly detection model and the workflow model. In the training stage the log files are parsed into log keys and parameter vectors. Sequences of log keys are used to train the log key anomaly detection model and the system workflow model. For each distinct log key there is also a parameter value model trained on the parameter vectors of a specific log key.

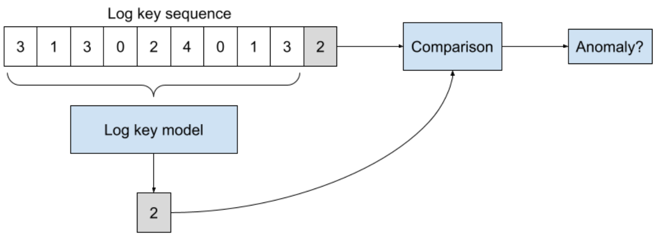


**Figure 6.1** An illustration of how DeepLog's log key model determines if an anomaly has occurred.

In the detection stage, a new log entry is parsed into a log key and parameter vector. Firstly the log key is checked with the log key model. This is done by viewing a history of log keys for the $n$ latest log messages. The last message is this sequence is extracted as a label, and the remaining $n-1$ log keys are passed to the log key model. The output of the model is the top $k$ predictions for the final log key in the sequence. The output of the model is compared to the ground truth value, and if the ground truth is among the top $k$ predicted values, the sequence is considered as normal. If not, it is considered as anomalous. The values of $n$ and $k$ are not predetermined can be tuned for optimal performance. The log key model pipeline is visualized in Figure 6.1.

If the log key is normal the parameter vector model will check if the parameter vector is normal for that log key. This is done in a similar way as for the log key anomaly detection. However, instead of comparing the output of the model with the ground truth directly, the *Mean Squared Error* (MSE) between the ground truth and the prediction is compared. If the MSE is outside a confidence interval, based on the validation MSE from the training phase, the parameter vector is considered as anomalous. If either of the log key or parameter vectors is abnormal, the workflow model will provide semantic information to help diagnose the anomaly. An analyst can report anomalies as false positives. If this happens, DeepLog can use this data to update the models.

## LogAnomaly

LogAnomaly, proposed in [22], builds upon the same principles as DeepLog, using LSTM networks to learn sequential log patterns and create probability distributions of the next log entry. The key difference is that LogAnomaly also takes the semantic information in the logs into account when creating and indexing log templates. Meng et al. [22] uses the term log template instead of log key, these terms will be used interchangeably. DeepLog simply indexes the different log templates and all of them are considered equally different. LogAnomaly uses the semantic information in the logs to create semantic relationships between log templates. Instead of only indexing the different log templates, LogAnomaly creates template vectors which contain semantic information about the log entry. This vector is created using the *Template2Vec* method mentioned in Chapter 4. Before *Template2Vec* can be applied, the logs have to be parsed and the templates have to be created. This is done using the, at the time, state-of-the-art logparser *FT-Tree* [32].

By using the methods *FT-Tree* and *Template2Vec* sequences of log messages are converted to sequences of log template vectors. An LSTM network is then used to extract sequential and quantitative features from these sequences in the offline training component of the model. In the online detection component, the log entry is compared and matched with the existing log templates. If there is a match the log is converted to the log template vector. Otherwise the log template will be approximated by analyzing the similarities between the new log entry and existing log template vectors and matching the new template vector to the most similar existing template. This is done with the assumption that new types of log templates will only differ by a small margin from previously known log templates. This allows the model to handle new log messages and provide results even though a log message has never been seen before. The trained LSTM model then determines if a log sequence is normal or anomalous. Meng et al. [22] recommends that the offline training is done periodically, e.g. weekly, to incorporate new system behaviors and log messages.

### Transformer models

Both LogAnomaly and DeepLog rely on LSTM networks to learn patterns in log data. One alternative to using LSTMs or other types of RNNs is to use a transformer. The first use of a transformer model in log anomaly detection was done by Nedelkoski et al. [24] who proposes *Logsy*. Nedelkoski et al. utilizes Tokenization of raw log messages instead of using a log parser. However, Chen et al. [2] used Logsy with both the log preprocessing of DeepLog and LogAnomaly by only using log template indices and adding semantic information. The *Logsy* model has two phases. During the offline phase the model is trained and parameters are tuned by using log messages. In the online testing phase each log message is run through the model and an anomaly score for each message is generated. The transformer model proposed by Vaswani et al. [28], is a model architecture that relies on an attention mechanism to draw global relationships between input and output. An attention mechanism maps queries and key-value pairs, in the form of vectors, to an output. The transformer itself maps input embeddings to a sequence of intermediate symbol representations, before generating an output sequence. This is done by dividing the model into an encoder and a decoder. Both parts consists of six layers with output normalization between each one. Each encoder layer passes the data through a multi-head self-attention function (see [28]) and a fully connected neural network. The decoder layers perform the same steps, but also adds an additional attention function on the output from the encoder.

Another implementation of a transformer model is done by [31] who proposes *LSADNET*. *LSADNET* utilizes a one-dimensional convolutional network (Conv1D) in combination with a globally sparse transformer model to find both global and local dependencies in the data. *LSADNET* utilizes log parsers to perform template extraction, different log parsers were tested by Zhang et al. with Drain, FT-Tree and Logsig being the top performing ones. The log templates are then vectorized using three different log vectorization components: log template semantic embedding (*LSE*), log key embedding (*LKE*) and log template transfer value (*LTF*). Where *LSE* represents the semantic information of the log message and *LKE* represents the log template. *LTF* represents the characteristics of the system, meaning it contains information about the likelihood of the next log template. These vectors are then combined into one vector, representing the whole log template. This is the vector that is then used by the model to create a probability distribution of the next log message.

## 6.3 This project

This section has presented various methods for anomaly detection used in the industry. For this thesis, the focus will be on finding collective anomalies and the methods Clustering, PCA and autoencoder will be compared and evaluated.

# 7

# Method

## 7.1 Project process

To start up the project, a literature study was conducted with the aim of finding best practices and state-of-the-art methods for log anomaly detection. The literature was found through the use of academic databases such as Google scholar and *LUB-search* (Lund University libraries). This resulted in various articles being read and summarized, and were to be referenced at later stages to determine suitable methods for the specific applications of this project.

With the knowledge gained during the literature study, the opt-out system was studied and log files were collected. Understanding the back end communication of the system was crucial for determining suitable methods. Because the system is distributed, involving various independent systems, there is no good way of following a sequence of messages, it was determined that a frequency analysis method probably would be the best choice. However, sequence analysis would still be of interest to explore how useful it would be in this context and if it could still yield any useful results.

To further analyze the logs and prepare them for the anomaly detection itself, the log messages needed to be categorized using a log parser. All of the state-of-the-art parsers that were identified during the literature study were created for unstructured log messages with the purpose of structuring the data within the message. Since the logs from the opt-out system already were structured, a new way of parsing and categorizing these messages, utilizing their structured nature, was explored. The method implemented was inspired by the Drain-method. The method was later deemed too static, hard coded for the specific system, and it did not handle new log messages or small message changes very well. Instead, another approach was taken where unstructured messages, containing the important information in the structured ones, were created and parsed by the state-of-the-art log parser Drain. An existing implementation of Drain by the LogPai team was used, which had to be slightly tweaked to accommodate the needs in this thesis.

To further prepare the log messages for analysis, window partitioning methods were implemented, which allowed partitioning logs according to fixed and sliding windows, based on either the number of messages or the timestamp embedded within each message. This means that a window either contains a fixed number of messages or all the messages within a certain time window.

The algorithms chosen for further experiments were *LogCluster*, *PCA analysis*, and an *autoencoder*. For *LogCluster* and *PCA analysis*, open source implementations by the *LogPai* team were used [14] . There was also an implementation of an autoencoder, but it relied on a specific data structure that could not be replicated in this setting. For this reason, a dense neural network was implemented and used in its place.

To evaluate the implemented methods, anomalies had to be injected into the data. Examples of various types of anomalies were presented by Ingka. These anomalies were then replicated and randomly distributed in the test data set. By analyzing which, and how many, of these anomalies each model finds, conclusions about their performances could be drawn.

When using machine learning methods you have to provide input parameters which can alter their performance. To fairly evaluate these methods, a parameter search was conducted. Since this requires a lot of runs, an environment was set up in GCP where simultaneous virtual machines could be deployed with different parameter setups.

The models were trained and tested on the data collected during the month of March 2022 where both a window and hyper-parameter search was conducted. The best performing methods and parameter choices were then validated with data from the month of April 2022, to see how the performance changes with alternative data sets.

## 7.2 Environment setup

To test the capabilities of each anomaly detection method a parameter search is conducted. Without testing different parameters, it is hard to know the full capability of the method. This means that a lot of training and testing runs need to be performed for each method. Running this locally and sequentially, with one run after each other, could take a tremendous amount of time. Instead of doing this locally, an environment in GCP was set up, where simultaneous runs could be conducted.

To be able to run the models in virtual machines, Docker images were created for each of the models. These images were pushed to a repository in GCP where they could be accessed by virtual machines with the required permissions. The environment was set up using terraform, which creates a Compute Engine in GCP based on a VM image which contains all of the required permissions needed to access the docker images. When the Compute Engine is created a startup-script is run. This startup-script pulls one of the docker images and starts it with specified parameters. Before terminating, the results of the test are put in a json-file and pushed to a bucket in GCP where it can be accessed. After terminating the test, the Compute Engine shuts itself off. A visual representation of what happens when the terraform script is run is provided in Figure 7.1. This terraform-script is started by a shell-script which can specify both the number of runs, i.e., VM:s created, as well as the parameters of the run.

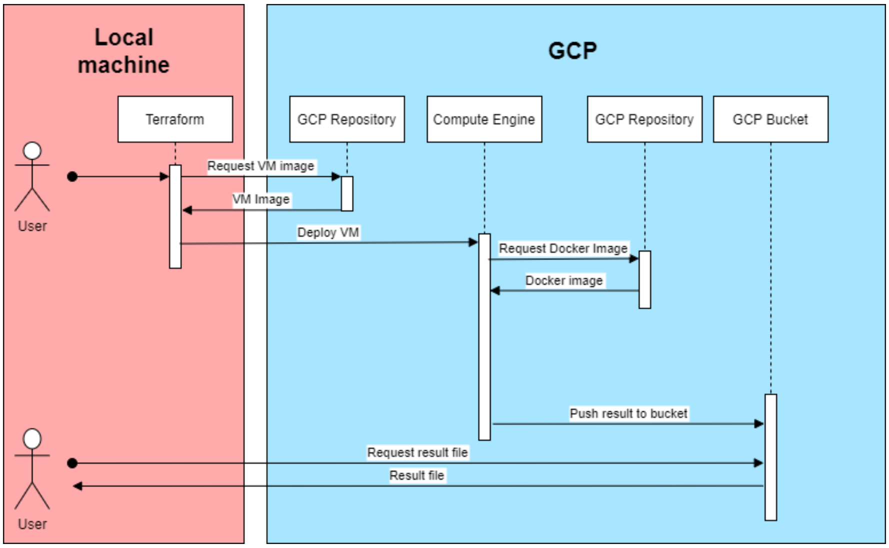


**Figure 7.1** Sequence diagram of a single run of the testing environment.

## 7.3 Log collection

### Collection of production logs

To perform the experiments, several months of log messages created by the opt-out system in production will be available. However, due to the vast amount of data, the virtual machines utilized are not able to handle all the data at once. Instead, the models are trained on one month's worth of logs, saving the remaining logs for possible future evaluations of the models being trained. The log messages come from three different sources within the system, namely *HTTP* logs, *application* logs and *stderr* logs. Logs are accessed through files containing the messages in JSON format. Since the messages from different sources are provided in different files, they need to be aggregated and sorted by their timestamps to be able to create the correct sequences.

### Collection of anomalies

To test the models there is a need of examples of anomalies that the models can be tested on. INGKA presented six different types of anomalies that can be used during the testing stage. These anomalies are:

**decryption-failed-v1** and **decryption-failed-v2**. These errors happen when the wrong key is used for encryption or decryption. This could mean that someone is trying to guess or use old encryption keys to try to get access to data.

**email-not-found** can happen when there is something wrong with the system that leads to the data about a user being inaccessible.

**request-body-validation-failed** can happen either because the link sent to the user was wrongly generated or because a user is trying to do something malicious.

**secret-error** means that there is something wrong with the handling of secrets in the system. Either because of the secrets being inaccessible or removed.

**api-fail** means that something has gone wrong in regards to a request sent to another api. This could either be due to the other api malfunctioning or some kind of communication problem , e.g,. network problems.

## 7.4 Parsing and vectorizing data

The vast majority of the prior research in log anomaly detection have studied the case of unstructured logs in production systems. This thesis tackles a different scenario, where the logs are generated in a structured manner from multiple production systems in a distributed cloud environment. This means the methods encountered in the literature study needed to be adapted to the problems of this thesis to work correctly. The parsing and vectorization steps of the anomaly detection pipeline have been experimented with to find a suitable solution for this setting.

Most of the parsers mentioned earlier only apply to unstructured log messages. Since this project instead are dealing with structured messages, there are a few alternatives on how to categorize log messages. One of the problems is that the log messages from the different systems do not have the same structure. All of them are JSON messages but not all of them contain the same fields. This means that the information in individual fields cannot be used to fully categorize these messages.

### Initial parsing approach

Log parsing generally means transforming unstructured logs to a structured format for easy data access and log template extraction. However, the cloud services in this project output log messages that are already structured, in the form of JSON objects. These objects do not need to be parsed for easy data access, since there are already fields defined containing the available data. Instead, they need to be parsed to allow extraction of log templates and a template id. To this end, the characteristics of a unique log type needed to be determined, which was initially done by recursively utilizing the object keys within each log. Some further knowledge of the logs at hand may be required to fully extend the scope of the log templates. For instance, HTTP log messages contain a *method* field with values such as *POST* or *GET*, which should result in to different log templates. Thus, for some data fields, the log templates should be determined by a combination of the key and its corresponding value, separated by a colon.

The idea for parsing the structured log messages comes from unstructured log parser techniques such as *AEL* and, most importantly, *Drain*. A parse tree of depth two was defined. The first layer in the tree was the number of key-value pairs in the top level of a log message. The next layer was a list of all the log messages with the same length as the number presented in the previous layer. This list contained a log template described by the object keys presented as a recursive list of key names. For each log template a unique identifier is also present.

### Final parsing approach

The initial parsing approach was valid, and could categorize each type of structured log message it was presented with. However, the solution required lots of prior knowledge about the logs being handled, since the programmer had to manually specify which fields were of interest for the system, and that some explicit fields should be handled differently. This resulted in the majority of the fields in the messages being removed, possibly discarding useful information.

A different approach was taken to mitigate these problems. Instead of a custom parser implementation, an existing, open source, implementation of *Drain* was used. However, to use this implementation, the logs needed to be converted to a common unstructured format before being passed through the parser. The common format looked as follows:

```
<Timestamp> <Id> <Severity>: <Content>
```

*Timestamp*, *Id*, and *Severity* are fields fetched from the log objects directly. They represent the time of creation, the id of the log message being handled, and the severity level of the log message, respectively. These values can be seen as general information about the log event, but do not affect the log templates being output from the algorithm. The *Content* part, however, determines the resulting log templates. It consists of a concatenation of multiple unstructured fields, along with fields such as *severity*, which are deemed important for detecting different log types.

A log entry may look as follows:

```
{
    "insertId": "09jxvn",
    "jsonPayload": {
        "logMeta": {
            "endPoint": "CMA-OPTOUT",
            "result": "SUCCESS"
        },
        "message": "cma-optout response successful",
        "name": "opt-out-be",
        "pid": 3,
    },
    "severity": "INFO",
    "timestamp": "2022-03-01T00:00:11.938999891Z"
}
```

Before parsing this log, its unstructured parts, along with a few other fields deemed useful, are concatenated into the format previously described. The resulting unstructured message would look like:

```
2022-03-01T00:00:11.938999891Z, 09jxvn, INFO, INFO CMA-OPTOUT SUCCESS
     cma-optout response successful
```

This unstructured message can then be passed through the parser itself, which extracts the following template:

```
INFO <*> SUCCESS <*> response successful
```

As seen in the log template above, some words are replaced with <*>, which indicates a variable part of the message. These are extracted into a separate feature vector. In this case, this feature vector would look like this:

```
['CMA-OPTOUT', 'cma-optout']
```

This results in a more dynamic solution, which can handle more types of log messages, and which also requires less prior knowledge about the logs and the system producing them.

## Feature extraction

To further prepare these sequences of log messages for anomaly detection, the log messages are split into windows of log messages. Since the chosen approaches, at least LogCluster and PCA, uses event count vectors, sliding windows over time were used, where, for instance, a window can contain ten minutes of log messages. This is because it was thought that time based windows would give a better representation of the program state than using a fixed number of logs in each window. The resulting sequences of log messages are then turned into event count vectors.

Before these event count vectors are used in anomaly detection they are first normalized and weighted. The normalization used is *zero-mean* and the weighting is done by the *tf-idf* (term frequency–inverse document frequency) method. In zero-mean normalization all values are changed the same amount so that the mean of all values in the dataset is zero. Tf-idf is a weighting factor which is common in text processing. It is intended to describe how important a word is in a document or collection of documents [30]. In the case of this thesis, it helps with discovering which message types are more important than other. For example a change from 300 to 350 messages of one type in an hour might not mean as much as a change from 0 to 50 of another type.

## 7.5 Model implementations

The three models used in this thesis are Clustering, PCA and Autoencoder. PCA and Clustering were used since they are two commonly used frequency analysis methods. To incorporate deep learning into the thesis, an autoencoder was also implemented. Most deep learning methods focus on sequence analysis, but the autoencoder also has the possibility of using frequency vectors. These methods were implemented and adapted to suit the setting of this project. Furthermore, the baseline system, which is described in Section 1.2, was converted from query lines to Python code. This allowed evaluation of the system using the same pipeline as for the other methods. The evaluation is described in detail in Section 7.7.

The PCA and LogCluster algorithms were evaluated by utilizing the open source implementations from LogPai. Their toolkit *LogLizer* [14] was utilized and adopted to the setting of this project.

The autoencoder, however, was implemented using the *Keras* deep learning API for Python. The same preprocessing as for the other methods was used, and the resulting data representation was passed to a dense neural network in the form of an autoencoder. The architecture consisted of an input layer with one neuron for each type of message in the data set. There was one hidden layer with a number of neurons that is smaller than the number of neurons in the input layer. The exact number of neurons in this layer is one of the parameters that will be tuned and experimented with in the upcoming steps. Finally, there was an output layer with the same dimensions as for the input layer. The model was trained using *MSE* as the loss function, and in each training step the validation loss is stored for later reference. Then, for each prediction in the test data, the test MSE is compared to the average validation MSE from the training step. If the reconstructed prediction differs from the average validation MSE by more than one standard deviation, the data point will be classified as an anomaly. This prediction routine is inspired by *DeepLog* [5] and serves as an alternative to a hard coded threshold value.

## 7.6 Hyperparameters

### Window settings

The different window setting parameters are **window_size** and **step_size**. These parameters are either set in seconds or number of messages depending on if a window based on time or number of messages is used. The PCA and LogCluster algorithms use the time based windows since they use event count vectors which automatically are the same length even if the number of messages are different.

### LogCluster

For the LogCluster algorithm there are three different parameters that can be tweaked to ensure the best possible performance of the model to specific cases. These are **nr_bootstrap_samples**, **anomaly_threshold** and **max_dist**. **nr_bootstrap_samples** are the number of samples used for the offline training of the model. The authors of [14] mention that a number larger than 10000 is infeasible due to memory consumption. 10000 is therefore used, since it should provide the best result while still being usable on normal machines in a timely manner. The **anomaly_threshold** parameter dictates the threshold for what should be considered an anomaly, and the **max_dist** parameter sets the threshold for when the clustering process should be stopped. For the window settings-scan, the parameters were set to their standard value, set by the authors. These values are **max_dist**=0.5 and **anomaly_threshold**=0.3.

### PCA

The PCA algorithm has two different parameters. The **n_components** parameter which dictates the number of principal components. This parameter is used to control the variance which is covered by the principal components. The other parameter is the **threshold** parameter which dictates the threshold for anomalies. This parameter is calculated automatically using Q-statistics if a number is not entered. The automatically calculated Q-statistics threshold was used in the experiments. For the window settings-scan, the **n_components** parameter was set to cover a variance ratio of 95%.

### Autoencoder

There are two main parameters which could impact the performance of the autoencoder. There is the number of nodes in the hidden layer, and there is the dropout rate. The number of hidden neurons basically determines how big of a dimensional reduction the network will perform. The dropout rate indicates at which rate the output from a neuron will be ignored, and is common measure for counteracting over fitting a neural network. There are other parameters which could have an impact on the performance of the network, such as the number of hidden layers and the loss function used, but these were not altered throughout the experiments conducted in this project. For the scan for window settings, the number of hidden nodes were set to ten, and the dropout rate to 0.1.

## 7.7 Model evaluation

All the implemented models were evaluated using a common testing pipeline, which is added on top of the ordinary anomaly detection pipeline. An anomaly injection step is added to the preprocessing, which injects log messages labeled as anomalous into the testing data. The training data are kept clean, and no anomalies are injected in this data set. The models are then trained as normal, using the training data set which represents the normal behavior of the system. After training, the models are evaluated. This is done by passing the test data to the models, which output a prediction, anomaly or not anomaly, for each window in the data. The predictions are then compared with the ground truth labels, which allows computation of the various metrics described in Chapter 3. The metrics recall, precision and f1-score are used since they are common metrics in classification problems.

First a window parameter search was conducted using standard hyperparameters for the models. In this search different combinations of step and window sizes was tested to find which parameters was best for the different models. The best performing window parameters were then used in a hyperparameter search where different hyperparameter combinations were tested. The best performing parameter setups where then tested on the dataset of logs collected during the month of April 2022 while still being trained on the data from March. Lastly the models were also both trained and tested with the data from April.

### Injecting anomalies

To allow injecting anomalies, Ingka provided examples of known anomalies which could be replicated and injected into the data set. The injection pipeline starts with sampling a normal data set from an input file to the system. This normal data set is divided into windows of a certain type, size, and with a specific step length. Each of these windows acts as a data point for the model to train on, and are split into training and test data sets.

The training data set will be kept as is, whereas the test data set will be altered to include a uniform distribution, i.e the same amount of each of the known anomalies mentioned above. A number of windows are then sampled from the normal data and anomalies are injected into these windows. The resulting anomalous windows are then added to the original set of test data and their labels are added to the ground truth vector.

When injecting anomalies it is assumed that the number of each type of anomaly is evenly distributed. It is also assumed that each type of anomaly is rare in relation to the total amount of log entries in the data set. For these reasons, the number of anomalies injected is set to 10% of the number of normal entries in the test set. These anomalies are evenly distributed between the different kinds of anomalies.

# 8

# Results

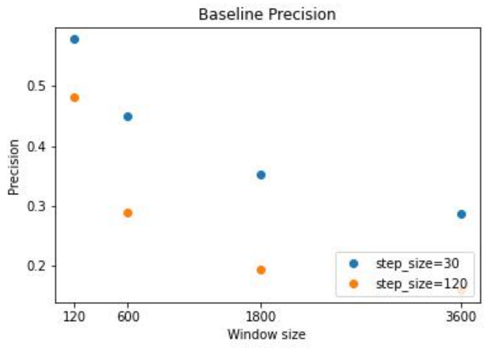
## 8.1 Baseline System

The results of the Baseline system can be seen in Table 8.1, Table 8.2 and Figure 8.1. Table 8.1 and 8.2 shows the baseline algorithm's performance when catching the different types of errors as well as the average false positive rate, i.e., the number of normal instances labeled as anomalies. Table 8.1 shows the results from using a step size of 30 seconds and Table 8.2 shows the results from a step size of 120 seconds. Overall, the baseline system is good at catching the *api-fail* error but not the rest of them, at least not when using a step size of 120 seconds. When using a step size of 120 seconds, the baseline algorithm catches more anomalies the greater the window size is, but this comes in conjunction with a higher false positive rate. When using a step size of 30 seconds a significant improvement can be seen, especially when using larger window sizes. The false positive rate still increases with larger window sizes but is almost identical to the false positive rate when using a step size of 120 seconds.

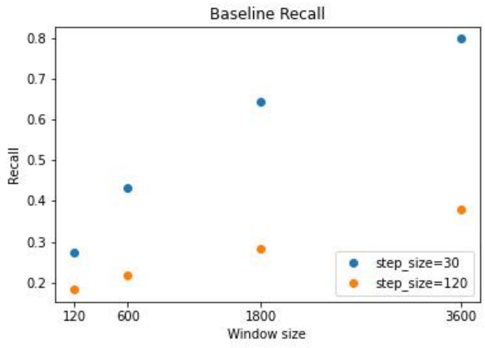
This can also be seen when looking at Figure 8.1 which shows the precision, recall and F1-score for the different step and window sizes. Higher numbers in these categories implies better performance of the method. As can be seen from these figures, a step size of 30 seconds performs significantly better than a step size of 120 seconds for all metrics. A rise in false positive rates can be seen for greater window sizes by looking at the precision graph and the recall graph shows that the larger the window size is, the more anomalies the algorithm catches.

## 8.2 Window Parameter search

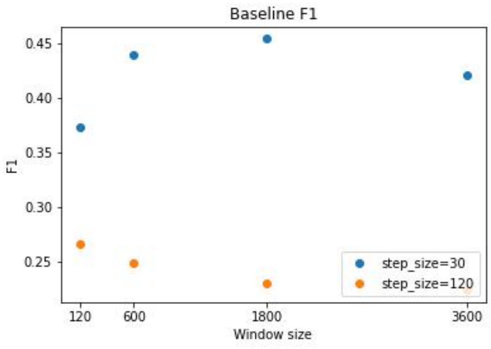
In this section the results from the window parameter search is described. For this experiment the default parameters, mentioned in Section 7.6, were used.



**(a)** Baseline Precision



**(b)** Baseline Recall



**(c)** Baseline F1-Score

**Figure 8.1** The evaluation metrics precision, recall and F1-score for the baseline algorithm for the window settings parameter-search

**Baseline with 30 second step size**

| | **120** | **600** | **1800** | **3600** |
|---|---|---|---|---|
| api-fail | 1.0 | 1.0 | 1.0 | 1.0 |
| decryption-failed-v1 | 0.12 | 0.27 | 0.58 | 0.81 |
| decryption-failed-v2 | 0.15 | 0.28 | 0.58 | 0.76 |
| email-not-found | 0.14 | 0.35 | 0.57 | 0.77 |
| secret-error | 0.10 | 0.32 | 0.56 | 0.70 |
| request-body-validation-failed | 0.13 | 0.35 | 0.57 | 0.75 |
| Average False Positive rate | 0.020 | 0.052 | 0.118 | 0.199 |

**Table 8.1** Table of the fractions of different anomaly types found using the Baseline system with a step size of 30 seconds and varying window sizes. The average false positive rate is also presented in the bottom row

**Baseline with 120 second step size**

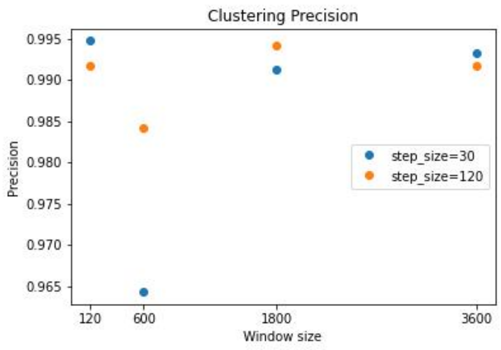
| | **120** | **600** | **1800** | **3600** |
|---|---|---|---|---|
| api-fail | 1.0 | 1.0 | 1.0 | 1.0 |
| decryption-failed-v1 | 0.02 | 0.05 | 0.13 | 0.24 |
| decryption-failed-v2 | 0.03 | 0.06 | 0.14 | 0.23 |
| email-not-found | 0.02 | 0.10 | 0.12 | 0.28 |
| secret-error | 0.01 | 0.07 | 0.16 | 0.25 |
| request-body-validation-failed | 0.02 | 0.04 | 0.15 | 0.26 |
| Average False Positive rate | 0.020 | 0.050 | 0.117 | 0.198 |

**Table 8.2** Table of the fractions of different anomaly types found using the Baseline system with a step size of 120 seconds and varying window sizes. The average false positive rate is also presented in the bottom row

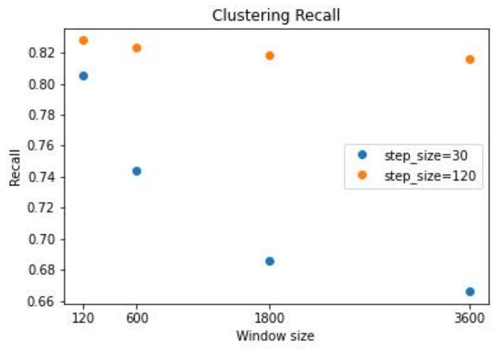
## Clustering

The results of the window parameter search for the clustering method can be seen in Table 8.3 and 8.4, as well as Figure 8.2. These tables show that the clustering method, as opposed to the baseline method, has a hard time successfully categorizing the api-fail anomaly as anomalous. It does however perform much better on the other anomalies. It can also be seen that the model has similar performance with regards to average false positive rate for both step sizes of 30 and 120 seconds.

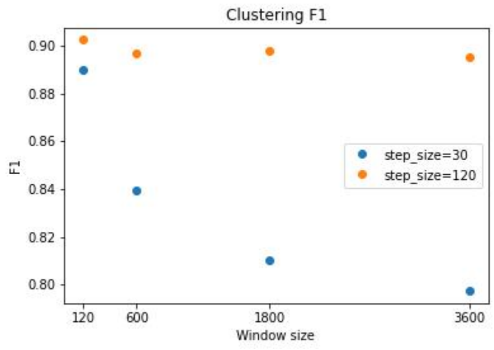
Figure 8.4a shows the average precision performances for various window sizes and window step lengths. It shows that the precision varies heavily between each set up. The best performance is found for a window size of 120 and a step size of 30. Figure 8.2b shows that the recall metric is heavily dependent on the window step length, and a length of 120 seconds outperforms step lengths of 30 seconds for all tests conducted. The best recall value was found for window size = 120 and step size = 120. The same holds true for the F1-score in Figure 8.2c.



**(a)** Clustering Precision



**(b)** Clustering Recall



**(c)** Clustering F1-Score

**Figure 8.2** The evaluation metrics precision, recall and F1-score for the Clustering algorithm for the window settings parameter-search

**Clustering with step size 30**

| | **120** | **600** | **1800** | **3600** |
|---|---|---|---|---|
| api-fail | 0.03 | 0.01 | 0.00 | 0.0 |
| decryption-failed-v1 | 0.95 | 0.84 | 0.81 | 0.75 |
| decryption-failed-v2 | 0.95 | 0.91 | 0.80 | 0.79 |
| email-not-found | 0.99 | 0.93 | 0.85 | 0.78 |
| secret-error | 0.98 | 0.95 | 0.87 | 0.90 |
| request-body-validation-failed | 0.98 | 0.98 | 0.78 | 0.77 |
| Average False Positive rate | 0.001 | 0.003 | 0.001 | 0.001 |

**Table 8.3** Table of the fractions of different anomaly types found using the Clustering model with a step size of 30 seconds and varying window sizes. The average false positive rate is also presented in the bottom row

**Clustering with step size 120**

| | **120** | **600** | **1800** | **3600** |
|---|---|---|---|---|
| api-fail | 0.00 | 0.00 | 0.00 | 0.00 |
| decryption-failed-v1 | 0.98 | 0.99 | 0.99 | 0.98 |
| decryption-failed-v2 | 0.99 | 0.98 | 0.98 | 0.98 |
| email-not-found | 0.99 | 0.97 | 0.97 | 0.98 |
| secret-error | 1.00 | 0.99 | 0.99 | 0.98 |
| request-body-validation-failed | 0.99 | 0.99 | 0.99 | 0.96 |
| Average False Positive rate | 0.001 | 0.002 | 0.001 | 0.001 |

**Table 8.4** Table of the fractions of different anomaly types found using the Clustering model with a step size of 120 seconds and varying window sizes. The average false positive rate is also presented in the bottom row

## PCA

The results of the PCA window parameter search can be seen in Table 8.5, Table 8.6, and Figure 8.3. Figure 8.3a shows an increasing trend in precision as the window size increases. The precision is generally slightly better for smaller step sizes, but they converge towards one, along with the larger step size, for larger window sizes. Furthermore, Figure 8.3b shows that the algorithm finds most, or all, of the anomalies that are injected into the test data. One of the recall measurements for **step size** = 30 and **window size** = 120 show an ever so slightly worse result. The F1-score also reaches its peak for large window sizes, for which the step size does not seem to matter. This can be seen in Figure 8.3c.

Table 8.5 and Table 8.6 shows the fraction of found anomalies for each type of anomaly, along with the average fraction of false positives for step sizes 30 and 120, respectively. PCA manages to find all instances of injected anomalies, except for one *api-fail* entry, which was also indicated in Figure 8.3a. It can also be seen that there is a downwards trend in the average number of false positives as the window size increases. This holds true for both window step lengths. On average, however, a shorter step size yields fewer false positives, especially for smaller sized windows.

**PCA with step size 30**

| | **120** | **600** | **1800** | **3600** |
|---|---|---|---|---|
| api-fail | 0.99 | 1.0 | 1.0 | 1.0 |
| decryption-failed-v1 | 1.0 | 1.0 | 1.0 | 1.0 |
| decryption-failed-v2 | 1.0 | 1.0 | 1.0 | 1.0 |
| email-not-found | 1.0 | 1.0 | 1.0 | 1.0 |
| secret-error | 1.0 | 1.0 | 1.0 | 1.0 |
| request-body-validation-failed | 1.0 | 1.0 | 1.0 | 1.0 |
| Average False Positives | 0.023 | 0.012 | 0.006 | 0.002 |

**Table 8.5** Table of the fractions of different anomaly types found using the PCA model with a step size of 30 seconds and varying window sizes. The average false positive rate is also presented in the bottom row

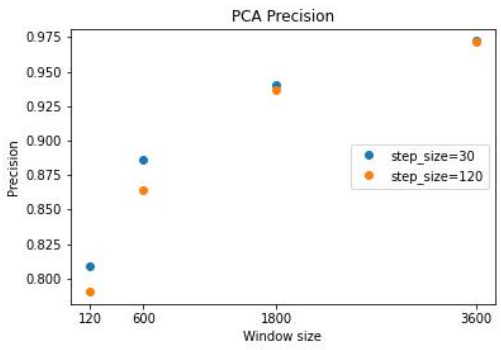
**PCA with step size 120**

| | **120** | **600** | **1800** | **3600** |
|---|---|---|---|---|
| api-fail | 1.0 | 1.0 | 1.0 | 1.0 |
| decryption-failed-v1 | 1.0 | 1.0 | 1.0 | 1.0 |
| decryption-failed-v2 | 1.0 | 1.0 | 1.0 | 1.0 |
| email-not-found | 1.0 | 1.0 | 1.0 | 1.0 |
| secret-error | 1.0 | 1.0 | 1.0 | 1.0 |
| request-body-validation-failed | 1.0 | 1.0 | 1.0 | 1.0 |
| Average False Positive rate | 0.026 | 0.015 | 0.006 | 0.002 |

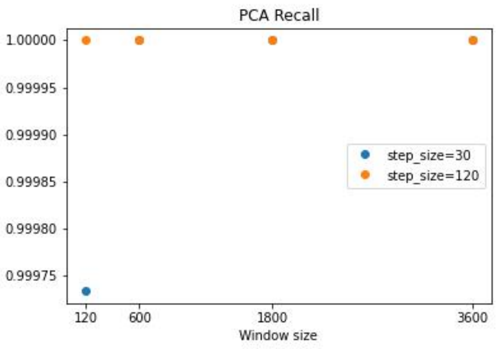
**Table 8.6** Table of the fractions of different anomaly types found using the PCA model with a step size of 120 seconds and varying window sizes. The average false positive rate is also presented in the bottom row

### Autoencoder

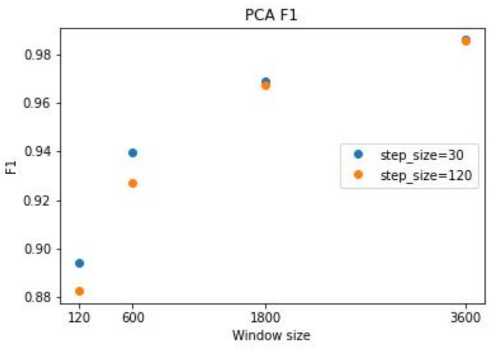
The window parameter search results for the autoencoder can be seen in Table 8.7, Table 8.8, and Figure 8.4. Figure 8.4a shows how the precision of the model increases as the windows get larger. The precision performance peaks at a window size of 1800 seconds, and then slightly decreases for 3600 seconds. It is also clear that the performance is significantly better for a step size of 120 than that of 30 seconds. Figure 8.4b shows that the recall for the model is 100% for all window sizes and step sizes tested. Table 8.7 and 8.8 confirms this, as all injected anomalies are found in each test. Figure 8.4c shows the F1-score of the autoencoder, which reaches its highest value at a window size of 1800 seconds with a step size of 120 seconds.



**(a)** PCA Precision



**(b)** PCA Recall



**(c)** PCA F1-Score

**Figure 8.3** The evaluation metrics precision, recall and F1-score for the PCA algorithm for the window settings parameter-search

Apart from the average fraction of found anomalies, for each type of anomaly, Table 8.7 and 8.8 also show the average false positive rates for each window size. They represent the window step lengths 30 and 120 seconds respectively. By comparing these tables it can be seen that a step size of 120 yields a lower false positive rate than for the corresponding window size with step size 30.

**Autoencoder with step size 30**

| | **120** | **600** | **1800** | **3600** |
|---|---|---|---|---|
| api-fail | 1.0 | 1.0 | 1.0 | 1.0 |
| decryption-failed-v1 | 1.0 | 1.0 | 1.0 | 1.0 |
| decryption-failed-v2 | 1.0 | 1.0 | 1.0 | 1.0 |
| email-not-found | 1.0 | 1.0 | 1.0 | 1.0 |
| secret-error | 1.0 | 1.0 | 1.0 | 1.0 |
| request-body-validation-failed | 1.0 | 1.0 | 1.0 | 1.0 |
| Average False Positives | 0.040 | 0.039 | 0.044 | 0.033 |

**Table 8.7** Table of the fractions of different anomaly types found using the Autoencoder model with a step size of 30 seconds and varying window sizes. The average false positive rate is also presented in the bottom row

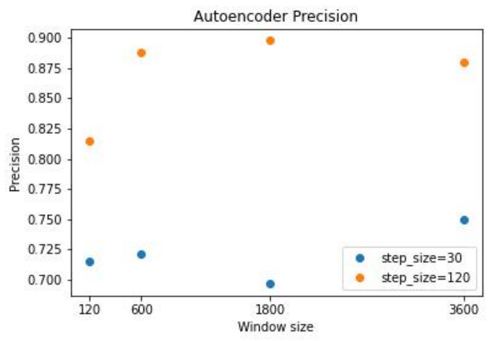
**Autoencoder with step size 120**

| | **120** | **600** | **1800** | **3600** |
|---|---|---|---|---|
| api-fail | 1.0 | 1.0 | 1.0 | 1.0 |
| decryption-failed-v1 | 1.0 | 1.0 | 1.0 | 1.0 |
| decryption-failed-v2 | 1.0 | 1.0 | 1.0 | 1.0 |
| email-not-found | 1.0 | 1.0 | 1.0 | 1.0 |
| secret-error | 1.0 | 1.0 | 1.0 | 1.0 |
| request-body-validation-failed | 1.0 | 1.0 | 1.0 | 1.0 |
| Average False Positives | 0.022 | 0.013 | 0.011 | 0.014 |

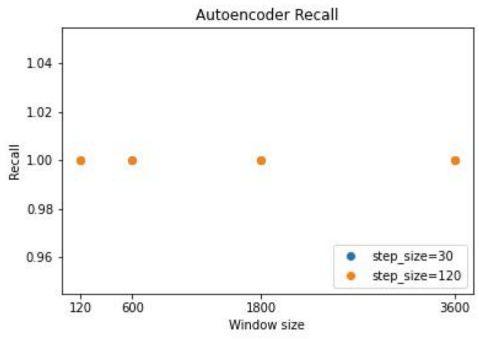
**Table 8.8** Table of the fractions of different anomaly types found using the Autoencoder model with a step size of 120 seconds and varying window sizes. The average false positive rate is also presented in the bottom row

## 8.3 Hyperparameter search

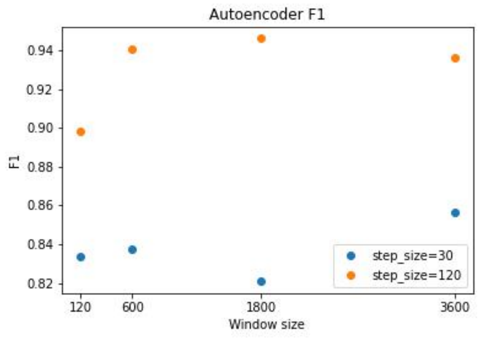
To continue the experiments for each model, the top performing window setup from Section 8.2 was chosen. For the clustering algorithm, the best average performance was found for a window size of 120 seconds, and a step size of 120 seconds. For PCA, these values were a window size of 3600 seconds, and 120 seconds window step. Finally, the autoencoder yielded the best result using a window size of 1800 seconds and a step size of 120 seconds.



**(a)** Autoencoder Precision



**(b)** Autoencoder Recall



**(c)** Autoencoder F1-Score

**Figure 8.4** The evaluation metrics precision, recall and F1-score for the Autoencoder method for the window settings parameter-search

## Clustering

Figure 8.5 shows the precision and recall values for the various hyperparameter setups that were tested for the clustering algorithm. Each dot represents average

| F1-Score | anomaly_threshold | max_dist |
|---|---|---|
| 0.972 | 0.1 | 0.3 |
| 0.967 | 0.1 | 0.4 |
| 0.939 | 0.1 | 0.5 |

**Table 8.9** The three best performing Clustering model parameter setups

precision-recall values, and it effectively illustrates how these metrics correlate against each other for different setups. It is those values that are commonly combined into an F1-score to determine the overall performance of a model. Table 8.9 shows the top three performing models, in terms of F1-score, along with the corresponding hyperparameters. An *anomaly_threshold* value of 0.1 and a *max_dist* value of 0.3 yielded the best overall performance of the model with an F1-score of 0.972. Comparing Figure 8.2c and Table 8.9 shows that the model was able to improve its performance when tuning the hyperparameters, instead of using the default ones.

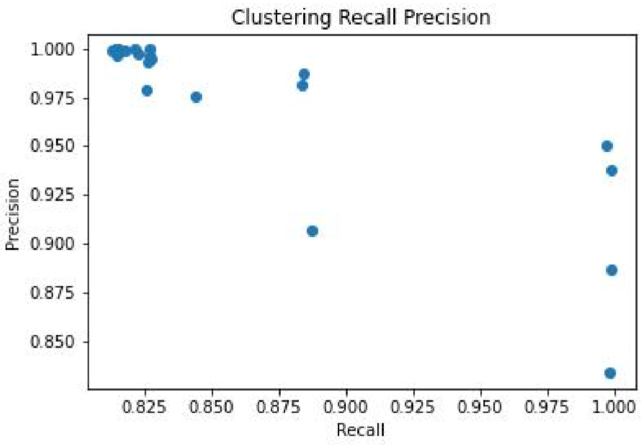


**Figure 8.5** Plot of precision and recall of different hyperparameter-setups for the clustering method

## PCA

Figure 8.6 shows the precision and recall values for the various hyperparameter setups that were tested for the PCA algorithm. There is a much more linear relation between the recall and precision for this model, where higher recall values also yield high precision values. In Table 8.10 it can be seen that high *n_components* values yielded the best F1-scores. An n_components value of 0.97 yielded the best F1-score, which was 0.990. The performance of the PCA algorithm was already high while using the default hyperparameters. However, comparing Figure 8.3c with

| F1-Score | n_components |
|---|---|
| 0.990 | 0.97 |
| 0.988 | 0.95 |
| 0.987 | 0.99 |

**Table 8.10** The three best performing PCA model parameter setups

Table 8.10 reveals that the algorithm was improved ever so slightly by tuning the n_components parameter.

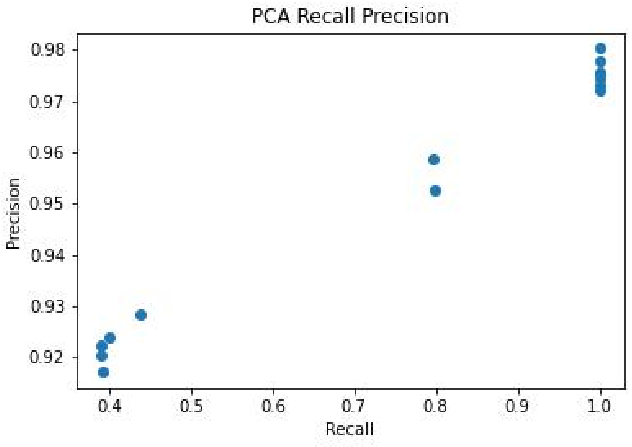


**Figure 8.6** Plot of precision and recall of different hyperparameter-setups for the PCA method

## Autoencoder

Figure 8.7 shows the precision and recall for the various hyperparameter setups having been tested for the autoencoder. These measurements are much more spread out than those for clustering, and especially those for PCA. It seems that this method is more sensitive to changes in its hyperparameter values. However, there are several setups yielding very good results for recall, and a few of those show high precision values as well. The best performing setups can be seen in Table 8.11. The very best model found used a *hidden_nodes* value of 6 and a *dropout* value of 0.1. The F1-score for this model was 0.953. Finally, comparing Figure 8.4c with Table 8.11 show that the model was improved by tuning the parameters compared to using the default parameters.

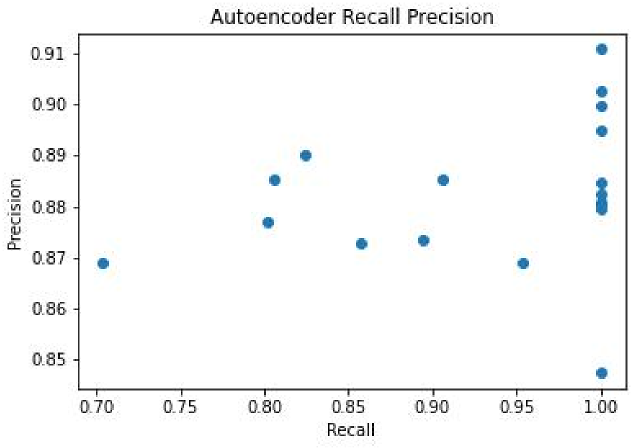


**Figure 8.7** Plot of precision and recall of different hyperparameter-setups for the Autoencoder method

| F1-Score | hidden_nodes | dropout |
|---|---|---|
| 0.953 | 6 | 0.1 |
| 0.948 | 4 | 0.1 |
| 0.947 | 8 | 0.1 |

**Table 8.11** The three best performing Autoencoder model parameter setups

## 8.4 Validating the models with other datasets

### Cluster

Training the Clustering model on the data from March and testing it with the data from April gets the results presented in Table 8.12. The model still performs pretty well by finding all of the injected anomalies. When comparing the values in Table 8.4 and Table 8.9 it is notable that the False positive rate is significantly higher and the precision is lower than what was previously achieved. Both training and testing the model on the data from April yields the results presented in Table 8.13. The recall is substantially lower than what was previously achieved which is a result of the model having a hard time to find the *request-body-validation-failed* and *decryption-failed-v2* errors. There is however an increase in precision when training on the data from April.

| Precision | Recall | F1 | False positive rate |
|---|---|---|---|
| 0.784 | 1.000 | 0.879 | 0.027 |

**Table 8.12** The results of a March-trained clustering model tested with logs from April

| Precision | Recall | F1 | False positive rate |
|---|---|---|---|
| 0.965 | 0.685 | 0.801 | 0.002 |

**Table 8.13** The results of a clustering model trained and tested with logs from April

## PCA

The results from training the best performing PCA model in the previous steps on the logs from March, but testing on the entire set of logs from April yielded the results in Table 8.14. Comparing these values with the values in Table 8.5, 8.6, and 8.10, it is clear that the average false positive rate is significantly higher, resulting in a lower precision score. It also tells how the model suddenly can not find all types of anomalies being injected, resulting in lower recall, and, inherently, a lower F1-score. Specifically, the model is having more problems finding the anomaly types *request-body-validation-failed*, *decryption-failed-v1*, and *decryption-failed-v2*. However, the model still finds the majority of these types of anomalies.

| Precision | Recall | F1 | False positive rate |
|---|---|---|---|
| 0.327 | 0.897 | 0.479 | 0.184 |

**Table 8.14** The results of a March-trained PCA model tested with the logs from April

Table 8.15 shows the results from the PCA model being trained on a subset of the logs from April, and also being tested with the remaining logs from the same month. It is clear that this solves most of the problems encountered in Table 8.14. The model manages to find all the anomalies being injected, and has a relatively low false positive rate. This scenario is comparable to the results presented in Table 8.5 and 8.15, where the model was trained and tested using the logs from March.

| Precision | Recall | F1 | False positive rate |
|---|---|---|---|
| 0.736 | 1.000 | 0.848 | 0.036 |

**Table 8.15** The results of a PCA model trained and tested with logs from April

## Autoencoder

Table 8.16 shows the result from training the best performing autoencoder from the previous experiments on the logs from March, and testing on the logs from April. Comparing this result with the values from Table 8.7, 8.8, and 8.11, reveals that the average false positive rate is much higher. This results in a lower precision value, and, inherently, a lower F1-score. The model still manages to find all the anomalies being injected to the test data.

Table 8.17 shows the results from the same autoencoder being trained on a subset of the logs from April, and being tested using the remaining logs from the same month. It still finds all of the injected anomalies, and the average false positive rate

| Precision | Recall | F1 | False positive rate |
|---|---|---|---|
| 0.293 | 1.000 | 0.453 | 0.241 |

**Table 8.16** The results of a March-trained autoencoder tested with the logs from April

is much lower than what is seen in Table 8.16, and also comparable to what is seen in Table 8.7 and 8.8. The F1-score is still lower than what is achieved when the model is trained and tested using only the logs from March.

| Precision | Recall | F1 | False positive rate |
|---|---|---|---|
| 0.625 | 1.000 | 0.769 | 0.060 |

**Table 8.17** The results of an autoencoder trained and tested with logs from April

# 9 Discussion

## 9.1 Baseline

We found that the overall performance, in terms of the F1-score, was best for a window size of 1800 seconds and a window step length of 30. The precision has a negative trend as for both window step lengths tested as the window size increases. In contrast, the recall metric has a positive trend as the window size increases. This means that as the window size increases, the precision, or the fraction of correct guesses, goes down. The recall, or the fraction of actual anomalies that were found, goes up.

It is important to note that the baseline system is hard coded, meaning it looks for predefined fields within the logs. For this reason, it is logical that the precision will decrease with larger window sizes, since there is likely more log messages matching the query in a larger window. Although there might be more messages matching the query, there is no guarantee that those messages are marked as an anomaly. As we have mentioned earlier, not all errors are anomalies. For the same reason, it is logical that the recall increases with larger window sizes, as the system is more likely to find more messages matching the query that also happens to be marked as an anomaly.

## 9.2 Clustering

During the window parameter search the clustering algorithm had a fluctuating trend for the precision metric, where the best value was found for a window size of 120 seconds and a window step length of 30. The recall metric had a negative trend as the window size increased, and performed the best for a window size of 120 and a step size length of 120. The F1-score had roughly the same performance behavior as the recall metric. For this window setting, the best parameter combinations for the clustering algorithm was *anomaly_threshold* = 0.1 and *max_dist* = 0.3.

The fact that the clustering algorithm performs better for smaller windows with no overlap is an interesting finding. Generally, machine learning models increase their performance with more training examples to learn from. Clearly, this is not the case for this clustering algorithm. From these results we can clearly see a decrease in performance with more duplicated data. The best windows settings create no duplicated data at all. That could be due to the nature of the agglomerative hierarchical clustering algorithm. Duplicating data creates intermediate data points between two adjacent, non-overlapping, data points. These intermediate data points will differ less from each other than non-overlapping windows would. This could potentially lead to clusters that originally should not be joined together, being joined together due to these new, in-between, data points.

What is also interesting is that the clustering model is unable to detect the *api-fail* error. This could potentially be due to the error existing somewhere in the training data, resulting in a cluster created for this error. However, the model was able to detect this error during the hyperparameter search. When having a lower both *anomaly_threshold* and *max_dist* value, the model could all of a sudden find these anomalies. The model being able to find this anomaly was due to the lowering of the anomaly_threshold parameter. The error probably generated an anomaly score, but not one big enough to be considered an anomaly due to the threshold being too high.

When training the model on the data from March and testing it on the data from April, the clustering algorithm performed pretty well compared to the other algorithms. It does not, however, perform well when training and testing on the dataset from April. What is interesting about these results is that there could be a correlation between the lower precision of the first test, and the low recall of the second. The lower precision of the first test could be due to anomalies existing in the dataset, which are not injected by us and therefore labeled as normal data. If this is the case, that could lead to the model creating clusters for these errors when training on the dataset from April. This could be the reason for the low recall in the second test.

## 9.3 PCA

The window parameter search showed that The PCA algorithm has a positive trend for precision as the window size increased, and the recall was 1.0 for all setups except for the window size of 120 and window step length 30. The overall performance was found to be best for a window size of 3600 seconds and roughly the same performance regardless of window step length. A window step length of 120 seconds was chosen to decrease execution time. This window setting resulted in the PCA algorithm being found to perform best using *n_components* = 0.97.

As we have previously seen, the PCA algorithm is great at finding the injected anomaly types in our experiments. This can mainly be seen in the recall metrics throughout the PCA results. An interesting finding is that the precision metric improves the bigger the window size is, which also yields a lot of duplicated data. This is the exact opposite behavior to what we observed for the clustering algorithm. A reason for the precision increasing with larger window sizes could be the fact that this yields higher average event counts for each window in the frequency analysis. Minor changes in these numbers will then have a lesser impact on the relative difference between different windows. It could also be the case that the PCA algorithm performs better with a more general snapshot of the system behavior, which the larger windows provide. We also found that our default value for the parameter *n_components* (0.95) was one of the best performing values, only slightly surpassed by *n_components* = 0.97.

The performance dropped significantly when training the model on the logs from March, and testing on the logs from April. In the first test, the precision dropped to just $\sim 0.33$ with an average false positive rate of 18.4%. For the entire month of April, this means thousands of false positives. Furthermore, this is the first test with these parameters where PCA is unable to find all of the injected anomalies. When training on the data from April, however, the model is once again able to find all of the anomalies being injected, albeit with a lower precision than previously seen. This could have some correlation with what was discussed in Section 9.2. If there are anomalies present in the data from April, and the PCA model is able to find them, that could be the reason for the lower precision, as these anomalies are not labeled. The results from the first test in the validation, where the model was trained on the data from march, are hard to explain. It could potentially be due to the data having vastly different characteristics in the two months. However, that does not explain why the clustering performs so much better in that test.

## 9.4 Autoencoder

During the window parameter search we found the best performing window setups for the default hyperparameters for the three models being tested. The autoencoder managed to achieve 1.0 recall for all window set ups being tested in this phase. The overall best performance was found for a window size of 1800 seconds and a step size of 120 seconds. Using this window setting, the autoencoder performed the best using *hidden_nodes* = 6 and *dropout* = 0.1.

From the window parameter search it is clear that the autoencoder has higher precision for larger windows. Generally, more data yields better results for deep learning models. However, this is not entirely true in our case. Step size 30 yields more overlap between windows and, inherently, generates more data points to analyze. We see higher precision for larger step sizes. This could be due to the overlapping data creating a much more specific notion of normal, since there will be less difference for each window encountered. If the notion of normal is made more specific, the model could potentially lower its threshold for what is considered an anomaly, which would result in a lower precision.

Similarly to the PCA algorithm, the autoencoder's performance during validation was much worse when using the data from April as a test set. The false positive rate was significantly increased, leading to lowered precision. However, in contrast to PCA, it still manages to find all injected anomalies, resulting in 100% recall. This is similar behavior to what was seen from the PCA algorithm, which could be because they both work on the premise of dimensional reduction. For the second test, the precision is still much lower than what was previously achieved, albeit significantly improved compared to the first test. The decrease in performance could be due to the parameters acquired in the previous experiments no longer being applicable since the data has changed. As discussed in previous sections, it could also be because there simply are more anomalies present that should be flagged as anomalies, but are not labeled as ones.

## 9.5 Performance and metrics

When analyzing the performance of the different models we are using recall, precision, F1-score and false positive rate. The recall accurately describes a model's ability to find the injected anomalies. However, the precision metric does not accurately describe the true precision of the model. One of the main challenges in anomaly detection is to create a dataset where anomalies are labeled correctly. We simply do not know if there are unlabeled anomalies in the data. A model finding these anomalies would therefore result in a lower precision metric when in reality it should find these instances. Furthermore, we mark entire windows as anomalous if they contain smaller sequences of messages we consider to be anomalies. In the test phase of our experiments, there is therefore no guarantee that what the model marks as anomalous is precisely the messages we have injected. This could be one of the reasons for the baseline system performing better with larger window sizes, since there is a higher probability of finding something else in the same window.

During our experiments we were surprised by the high recall performance of our models. This is likely due to the fact that the anomalies we inject are specific and contain messages that are rare or never occur in the training data. This makes it easier for the models to detect than more subtle anomalies would be. Also, when the messages are this rare, the *tf-idf* step in the preprocessing makes these occurrences stand out even more.

## 9.6 Anomaly detection in production

The experiments in this project have been conducted in an environment that does not mimic a production environment very well. The experiments utilized a dataset presented as JSON objects in large text files. In production, however, the logs would be streamed to the anomaly detection application, where they would be parsed and preprocessed individually, before being stored for later reference. Instead of the partitioning methods used in the experiments, a single window would be constructed periodically from the log history. This would then be passed as input to the application, which would determine whether or not it is anomalous. The period between window samples would have to be explicitly and appropriately defined. It is important to consider that longer periods between samples means there could be a delay from an anomaly actually occurring to it being discovered.

From the results of our experiments we can also say that the models would have to be continuously retrained to remain viable. We have seen indications that the performance may differ greatly between months, and it may therefore be appropriate to retrain the models more frequently. Since the models, especially the clustering methods, seem to be sensitive to anomalies in the training data, the logs being used as training data for the next model would have to be analyzed before usage. For instance, this could be done through manual feedback on the anomalies having been found to determine if they are true or false positives. If they are actual anomalies, they should be removed from the next training set and potentially added to a set of known anomalies that can be utilized in future testing.

There is also a problem regarding completely new types of log messages. These will create a new log key, resulting in longer event count vectors than what the models are trained with. This would crash the program if not handled properly. This would require a new training of the model, with the new log key taken into consideration. This event should be immediately flagged as an anomaly since the message has never been seen before. This also means that if the system changes its general behavior, e.g., from system updates where new types are created, the models would need to be retrained. There would still be problems regarding new behaviors until enough such events have occurred for the models to recognize them as normal.

# 10

# Conclusion

Looking back at the objectives of this thesis in Section 1.3, we can safely say that we have managed to improve upon the existing error detection system. Both in the aspect of detecting actual faults and anomalies, as well as lowering the number of false positives, which has previously been an issue for this system. We compared a clustering algorithm, PCA, and an autoencoder by looking at precision, recall, F1, and false positive rate, with various training and testing configurations. From these experiments we have seen that the models perform differently based on the training and testing data being used. PCA performs the best when being trained and tested on data from the same period, which indicates that this model would have to be frequently retrained when deployed in production. The clustering algorithm seems to be more sensitive to anomalies being present in the training data, and should be trained with this in mind. The clustering algorithm performs better than the two other models over time, when using training and testing data sets from different periods. Finally, the autoencoder performs similarly to PCA but with a slightly higher false positive rate, and therefore lower precision.

Although we did not find the autoencoder to perform the best, there are a plethora of unexplored architectures and configurations to continue experimenting with. For this thesis we used a simple autoencoder with a single hidden layer.

From the results acquired in this thesis, we can conclude that using an anomaly detection system would be an improvement to the rule based error detection system currently in place. Although, there are more challenges to overcome before using these methods in production. At this stage, we do not find the autoencoder to perform at the same level as the other two methods. Although, it could possibly be improved with other configurations. If frequent training is an alternative, the PCA algorithm would be a good option. Finally, if you can maintain a clean training data set, the clustering method would be a valid alternative where you would not need as frequent training as for PCA.

## 10.1 Future work

The ultimate goal, building on the results from this thesis, would be to deploy the anomaly detection to a production environment. Before doing this, however, it would be interesting to research other hyperparameter and window size combinations, as well as experimenting further with data from different time periods. It would also be interesting to study the performance of other autoencoder architectures in this setting. Another aspect that would be interesting to look into before deploying the models in production is the response time of the models, i.e., how long it takes for an anomaly to be detected.

To get these anomaly detection models ready for production, there is still some way to go. You have to look into how the logs could be streamed to the system, and how the stream would affect the current anomaly detection pipeline. One would have to determine how the logs would be partitioned, how long an acceptable delay for detecting the anomalies would be, and how a clean training dataset would be maintained.

# Bibliography

[1] V. Chandola, A. Banerjee, and V. Kumar. "Anomaly detection: a survey". *ACM Comput. Surv.* **41** (2009). DOI: 10.1145/1541880.1541882.

[2] Z. Chen, J. Liu, W. Gu, Y. Su, and M. R. Lyu. "Experience report: deep learning-based system log analysis for anomaly detection". *CoRR* **abs/2107.05908** (2021). arXiv: 2107.05908. URL: https://arxiv.org/abs/2107.05908.

[3] T. Dillon, C. Wu, and E. Chang. "Cloud computing: issues and challenges". In: *2010 24th IEEE International Conference on Advanced Information Networking and Applications*. 2010, pp. 27–33. DOI: 10.1109/AINA.2010.187.

[4] Docker Inc. *Docker overview*. [Online; accessed 20-April-2022]. URL: https://docs.docker.com/get-started/overview/.

[5] M. Du, F. Li, G. Zheng, and V. Srikumar. "Deeplog: anomaly detection and diagnosis from system logs through deep learning". In: *CCS '17: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017, pp. 1285–1298. DOI: https://doi.org/10.1145/3133956.3134015.

[6] I. C. Education. *Deep learning*. [Online; accessed 9-May-2022]. URL: https://www.ibm.com/cloud/learn/deep-learning.

[7] R. A. Ghalehtaki, A. Ebrahimzadeh, F. Wuhib, and R. H. Glitho. "An unsupervised machine learning-based method for detection and explanation of anomalies in cloud environments." *2022 25th Conference on Innovation in Clouds, Internet and Networks (ICIN), Innovation in Clouds, Internet and Networks (ICIN), 2022 25th Conference on* (2022), pp. 24–31. ISSN: 978-1-7281-8688-7. URL: http://ludwig.lub.lu.se/login?url=https://search.ebscohost.com/login.aspx?direct=true&AuthType=ip,uid&db=edseee&AN=edseee.9758126&site=eds-live&scope=site.

[8] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[9] Google. *Cloud run*. [Online; accessed 20-April-2022]. URL: `https://cloud.google.com/run`.

[10] Google. *Compute engine*. [Online; accessed 20-April-2022]. URL: `https://cloud.google.com/compute`.

[11] Google. *Google cloud overview*. [Online; accessed 20-April-2022]. URL: `https://cloud.google.com/docs/overview`.

[12] HashiCorp. *Terraform documentation*. [Online; accessed 9-May-2022]. URL: `https://www.terraform.io/docs`.

[13] 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)*. 2017, pp. 33–40. DOI: `10.1109/ICWS.2017.13`.

[14] S. He, J. Zhu, P. He, and M. R. Lyu. "Experience report: system log analysis for anomaly detection". In: *27th IEEE International Symposium on Software Reliability Engineering, ISSRE 2016, Ottawa, ON, Canada, October 23-27, 2016*. IEEE Computer Society, 2016, pp. 207–218. DOI: `10.1109/ISSRE.2016.21`. URL: `https://doi.org/10.1109/ISSRE.2016.21`.

[15] A. Hrusto. *Towards optimization of anomaly detection using autonomous monitors in devops*. 2022. URL: `http://ludwig.lub.lu.se/login?url=https://search.ebscohost.com/login.aspx?direct=true&AuthType=ip,uid&db=edsswe&AN=edsswe.oai.lup.lub.lu.se.e02869a2.16f5.429c.bc2c.bcd279ecab26&site=eds-live&scope=site`.

[16] IBM. *Cloud computing*. [Online; accessed 20-April-2022]. URL: `https://www.ibm.com/cloud/learn/cloud-computing`.

[17] IBM Cloud Education. *Neural networks*. [Online; accessed 9-May-2022]. URL: `https://www.ibm.com/cloud/learn/neural-networks`.

[18] Q. Lin, H. Zhang, J.-G. Lou, Y. Zhang, and X. Chen. "Log clustering based problem identification for online service systems". In: *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. 2016, pp. 102–111.

[19] O. Maimon and L. Rokach. *Data mining and knowledge discovery handbook*. New York : Springer, c2010, 2010.

[20] A. A. Makanju, A. N. Zincir-Heywood, and E. E. Milios. "Clustering event logs using iterative partitioning". In: *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '09. Association for Computing Machinery, Paris, France, 2009, pp. 1255–1264. ISBN: 9781605584959. DOI: `10.1145/1557019.1557154`. URL: `https://doi.org/10.1145/1557019.1557154`.

[21] P. Mell and T. Grance. *The NIST Definition of Cloud Computing*. Tech. rep. 800-145. National Institute of Standards and Technology (NIST), Gaithersburg, MD, 2011. URL: `http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf`.

[22] W. Meng, Y. Liu, Y. Zhu, S. Zhang, D. Pei, Y. Liu, Y. Chen, R. Zhang, S. Tao, P. Sun, and R. Zhou. “Loganomaly: unsupervised detection of sequential and quantitative anomalies in unstructured logs”. In: *IJCAI*. 2019.

[23] Microsoft. *What is cloud computing?* [Online; accessed 20-April-2022]. URL: `https://azure.microsoft.com/en-us/overview/what-is-cloud-computing/#uses`.

[24] S. Nedelkoski, J. Bogatinovski, A. Acker, J. Cardoso, and O. Kao. “Self-attentive classification-based anomaly detection in unstructured logs”. In: *2020 IEEE International Conference on Data Mining (ICDM)*. 2020, pp. 1196–1201. DOI: `10.1109/ICDM50108.2020.00148`.

[25] Pragati Baheti. *The essential guide to neural network architectures*. [Online; accessed 10-May-2022]. URL: `https://www.v7labs.com/blog/neural-network-architectures-guide`.

[26] Ravindra Parmar. *Common loss functions in machine learning*. [Online; accessed 11-May-2022]. 2018. URL: `https://towardsdatascience.com/common-loss-functions-in-machine-learning-46af0ffc4d23`.

[27] M. Sakurada and T. Yairi. “Anomaly detection using autoencoders with nonlinear dimensionality reduction”. In: *Proceedings of the MLSDA 2014 2nd Workshop on Machine Learning for Sensory Data Analysis*. MLSDA’14. Association for Computing Machinery, Gold Coast, Australia QLD, Australia, 2014, pp. 4–11. ISBN: 9781450331593. DOI: `10.1145/2689746.2689747`. URL: `https://doi.org/10.1145/2689746.2689747`.

[28] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. “Attention is all you need”. In: I. Guyon et al. (Eds.). *Advances in Neural Information Processing Systems*. Vol. 30. Curran Associates, Inc., 2017. URL: `https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf`.

[29] Wikipedia contributors. *Machine learning — Wikipedia, the free encyclopedia*. [Online; accessed 19-April-2022]. 2022. URL: `https://en.wikipedia.org/w/index.php?title=Machine_learning&oldid=1081891971`.

[30] Wikipedia contributors. *Tf–idf — Wikipedia, the free encyclopedia*. [Online; accessed 26-May-2022]. 2022. URL: `https://en.wikipedia.org/w/index.php?title=Tf%5C%E2%5C%80%5C%93idf&oldid=1071253989`.

[31] C. Zhang, X. Wang, H. Zhang, H. Zhang, and P. Han. “Log sequence anomaly detection based on local information extraction and globally sparse transformer model”. *IEEE Transactions on Network and Service Management* **18**:4 (2021), pp. 4119–4133. DOI: `10.1109/TNSM.2021.3125967`.

[32] S. Zhang, W. Meng, J. Bu, S. Yang, Y. Liu, D. Pei, J. Xu, Y. Chen, H. Dong, X. Qu, and L. Song. “Syslog processing for switch failure diagnosis and prediction in datacenter networks”. In: *2017 IEEE/ACM 25th International Symposium on Quality of Service (IWQoS)*. 2017, pp. 1–10. DOI: `10.1109/IWQoS.2017.7969130`.

[33] Z.-H. Zhou. *Machine Learning*. Springer Nature eBook, 2021.

[34] J. Zhu, S. He, J. Liu, P. He, Q. Xie, Z. Zheng, and M. R. Lyu. “Tools and benchmarks for automated log parsing”. In: *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 2019, pp. 121–130. DOI: `10.1109/ICSE-SEIP.2019.00021`.

[35] Zichen Wang. *Pca and svd explained with numpy*. [Online; accessed 8-June-2022]. 2019. URL: `https://towardsdatascience.com/pca-and-svd-explained-with-numpy-5d13b0d2a4d8`.

| | |
|---|---|
| **Lund University**<br>**Department of Automatic Control**<br>**Box 118**<br>**SE-221 00 Lund Sweden** | *Document name*<br>MASTER'S THESIS<br>*Date of issue*<br>June 2022<br>*Document Number*<br>TFRT-6176 |
| *Author(s)*<br>David Nilsson<br>Albin Olsson | *Supervisor*<br>Gustav Hochbergs, Ikea IT AB<br>Johan Eker, Dept. of Automatic Control, Lund University, Sweden<br>Karl-Erik Årzén, Dept. of Automatic Control, Lund University, Sweden (examiner) |

*Title and subtitle*

Log Anomaly Detection of Structured Logs in a Distributed Cloud System

*Abstract*

As computer systems grow larger and more complex, the task of maintaining the system and finding potential security threats or other malfunctions become increasingly hard. Traditionally, this has had to be done by manually examining the logs. In modern systems, this can become infeasible due to either the large amount of logs or the complexity of the system. By using machine learning based anomaly detection to analyze system logs, this can be done automatically.

In this thesis the authors have researched the area of anomaly detection, and implemented an anomaly detection pipeline for a specific system. Three different machine learning based anomaly detection models were implemented, namely a clustering algorithm, PCA, and a neural network in the form of an autoencoder. These models were compared and evaluated with regards to a baseline error detection system, which was already in place for the target system. They were also compared against each other to find which models performed best, and in which circumstances. To compare the models, six different types of known anomalies were injected into the data.

When comparing the performances of the different methods, all of them were found to outperform the baseline system. In the first experiment, where the models were trained and tested using data from the same time period, PCA achieved the highest F1-score of 0.990. In the second experiment the models were trained and tested using data from separate time periods. In this scenario, the clustering algorithm outperformed the others, with an F1-score of 0.879. Both PCA and the autoencoder found many false positives, reducing their precision and thereby their F1-score.

*Keywords*

*Classification system and/or index terms (if any)*

*Supplementary bibliographical information*

| | | |
|---|---|---|
| *ISSN and key title*<br>0280-5316 | | *ISBN* |
| *Language*<br>English | *Number of pages*<br>1-74 | *Recipient's notes* |
| *Security classification* | | |

http://www.control.lth.se/publications/