Automatic Log Based Anomaly Detection in Cloud Operations using Machine Learning

Jacob Gummesson Atroshi

Christian Le

LUND UNIVERSITY

Department of Automatic Control

MSc Thesis
TFRT-6129
ISSN 0280-5316

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

Abstract

For modern large scale cloud services a fast and reliable anomaly detection is of utmost importance. Traditionally developers perform simple keyword search, for keywords such as "error" or "fail" in the log data, one of the main data sources that depicts the state of the system. In today's large-scale systems however several TB of log messages can be output every day making manual search highly ineffective. To address the problem there have been many anomaly detection methods based on the few publicly available log data sets. In this thesis we present a unique data collection method using a virtualized OpenStack cloud system to collect log data from six simulated anomaly scenarios. Three different detection methods are presented using both the dynamic and static parts of the individual log messages. An investigation of the impact of parameters such as time window size is done by an evaluation of the various anomaly types. Among the four conventional machine learning models based on the static parts gave a good performance of a 50% detection rate with a 0.35% false alarm rate. In addition the results show a better LSTM model performance when using the dynamic rather than the static parts. For the LSTM using dynamic parameters the results depended on the anomaly type, and the parameter, with the best average scores around 55-65% detection rate with a false alarm rate around 0.5-1%.

Acknowledgements

We would like to thank Johan Eker, Torgny Holmberg and Robert Marklund at Ericsson for their wonderful help and taking care of us during this period. Even though we have only met once outside Microsoft Teams, without their time and contributions this thesis would not have been possible. We would also like to thank our examiner Karl-Erik Årzén for all his valuable advice regarding the report.

Contents

1

Introduction

1.1 Background

The concept of *Cloud Computing*, on-demand computing as processing power, storage, and applications, is not something new and has been a long-held dream for computing as a utility. The term *Cloud Computing* includes both the hardware and system software of the cloud provider's data center [41, 3]. *Utility Computing*, the service sold to the public has given many companies and developers new opportunities, both in terms of economics and reduced processing time.

Today there are several large businesses offering cloud services, such as Amazon Web Services, Microsoft Azure and Google Cloud [4]. For these cloud providers to offer a high reliability service, behaviors that are not normal, or anomalies, must be detected. Specifically a fast and accurate anomaly detection system is required. If an anomaly is not found it can impact a sizeable amount of concurrent users. The systems are monitored in various different ways, and constantly output a lot of data, such as numerical metrics of the servers, or textual software logs. Traditionally, developers manually try to find anomalies in logs using for example a simple keyword search for *error* or *fail* in the log data. However, manual detection is both error prone, and time consuming even on small-scale systems [37].

Simply the fact that today's large-scale system services output several GB, or TB of data each day makes manual searching very ineffective [37]. However, modern software design makes the problem even more difficult. Due to modular design many different log files are produced on different physical servers, and anomalies in one component may affect the whole system in an unpredictable fashion. To exemplify this difficulty in large scale distributed systems one may look at the distinction made between faults, errors, and failures. Behrooz Parhami defines these terms as states that the system may be in and transition between [51]. For instance, the state of being faulty occurs when a system has a flaw which has the potential to cause problems, however, the system still functions and problems are not noticed. He gives the example of a tire with a hole; the driver will not immediately notice the fault, but there is a potential to cause problems.

When a fault is active it is called an error, with the example of a tire with a hole, this would be when the tire pressure is low. However, due to modern car design it would still be possible to drive the car. A failure would be when it is no longer possible to drive the car, or in terms of software, the user of a software system gets incorrect results. Parhami defines in his model several more states, including ideal, defective, malfunctioning, and degrading, exhibiting the complex nature of a distributed software system. The classical keyword search will find all errors, however, can not distinguish between an error, degradation, malfunction, or a failure, contributing to a need for more intelligent anomaly detection systems.

1.2 Objective of Project

As stated earlier, there is a need for fast, and effective anomaly detection. At Ericsson Research the experience has been that anomalies in the system have not been detected until customers report issues. This is a large problem with cloud infrastructure promising high reliability and very low downtime. The aim of this thesis is to develop such an anomaly detection system. Furthermore, we wish to generate data in a controlled environment using virtual machines, VMs, on top of virtual machines, in order to understand in what scenarios the anomaly detection system can detect faults. We wish to compare, and improve, different state of the art models in terms of their efficacy, and how long time it takes to detect the anomalies from the generated data.

1.3 Automatic anomaly detection

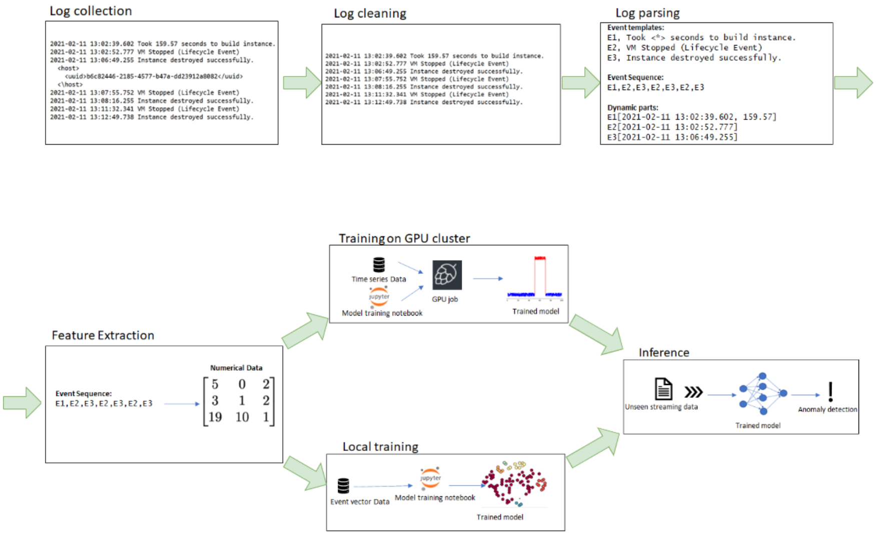
Our approach to create an automatic anomaly detection system can be broken down into several stages: *Log Collection*, *Log Cleaning*, *Log Parsing*, *Feature Extraction*, and *Anomaly Detection*. Each part is summarized below and will be covered thoroughly later in the thesis. The pipeline that will be developed for this thesis is shown in Figure 1.1.

Log Collection
Large cloud systems generate an enormous amount of logs which reflect the state of the system, for example the Microsoft system presented in [37] produced over 1 PB every day. The logs consists of metadata, such as a timestamp and a log message, and have multiple uses, where one is for anomaly detection. In Figure 1.1, seven lines of collected log data is shown. In this thesis logs from OpenStack are used. OpenStack is a distributed cloud platform software where different services handle separate parts of the system [49] and it is thoroughly explained in Section 2.2.

One of the main difficulties is that each of these services produce individual logs for each server running the service resulting in an enormous number of log files updated in parallel.

Log Cleaning

Often collected log data contain log messages which do not conform to the general structure of the data. These must be removed or dealt with in a cleaning process. In Figure 1.1, three log lines are shown which do not have the metadata present in the other log lines. In the cleaning step, these lines may be removed, or in some cases handled in a more intelligent way by appending to previous lines. This is described in detail in Chapter 6.



```
2021-02-11 13:02:39.602 Took 159.57 seconds to build instance.
2021-02-11 13:02:52.777 VM Stopped (Lifecycle Event)
2021-02-11 13:06:49.255 Instance destroyed successfully.
  <host>
    <uuid>b6c82446-2185-4577-b47a-dd23912a8082</uuid>
  <\host>
2021-02-11 13:07:55.752 VM Stopped (Lifecycle Event)
2021-02-11 13:08:16.255 Instance destroyed successfully.
2021-02-11 13:11:32.341 VM Stopped (Lifecycle Event)
2021-02-11 13:12:49.738 Instance destroyed successfully.
```

```
2021-02-11 13:02:39.602 Took 159.57 seconds to build instance.
2021-02-11 13:02:52.777 VM Stopped (Lifecycle Event)
2021-02-11 13:06:49.255 Instance destroyed successfully.
2021-02-11 13:07:55.752 VM Stopped (Lifecycle Event)
2021-02-11 13:08:16.255 Instance destroyed successfully.
2021-02-11 13:11:32.341 VM Stopped (Lifecycle Event)
2021-02-11 13:12:49.738 Instance destroyed successfully.
```

Event templates:
```
E1, Took <*> seconds to build instance.
E2, VM Stopped (Lifecycle Event)
E3, Instance destroyed successfully.
```

Event Sequence:
```
E1,E2,E3,E2,E3,E2,E3
```

Dynamic parts:
```
E1[2021-02-11 13:02:39.602, 159.57]
E2[2021-02-11 13:02:52.777]
E3[2021-02-11 13:06:49.255]
```

Event Sequence:
```
E1,E2,E3,E2,E3,E2,E3
```

$$\begin{bmatrix} 5 & 0 & 2 \\ 3 & 1 & 2 \\ 19 & 10 & 1 \end{bmatrix}$$

Figure 1.1: Pipeline for automatic anomaly detection from log data.

Log Parsing

It is difficult to use the textual data from logs in machine learning models. They are produced as unstructured text made to be read by humans whereas machine learning models require numerical data. Therefore parsing is done to form structured data, from which numerical features can be extracted. A common way to do this is to create so called *Event templates* by splitting a log message into a constant and dynamic part. Constant parts are shared with many logs, whereas the dynamic part can for example be IP addresses or file paths. Each unique type of constant part forms a log template and is assigned an *Event ID*. It is the sequence of *Event IDs* that has been used in previous anomaly detection methods. Three *Event templates* are shown in Figure 1.1 where they are made into events. A more in-depth description is found in Chapter 3.

Feature Extraction
Using the *Event ID* sequence output from the parsing of the logs, a *Feature vector* is extracted which then can be used in the next step, anomaly detection. There are several techniques for this. The two most common are *Sliding window* and *Fixed window*. These algorithms create an event counter during a certain time window where one coordinate in the vector represents the number of occurrences of a specific *Event ID* within the time window. There is also a method which is time independent, known as *Session window*, in which events are grouped together by some method other than time, for example the *Block ID* in certain logs. All three methods will be further explained in the thesis in Chapter 3. Feature extraction can also include extracting the dynamic parts of the log messages to form a time series.

Anomaly Detection
The *Event vectors* are thereafter used as input to train a machine learning model. When receiving an incoming log sequence the aim of the model is to determine whether or not it is an anomaly. In the thesis we also use anomaly detection models which use the sequence of *Event IDs* without the feature extraction. These models are first trained offline, and will then be utilized on streaming logs.

1.4 Outline of thesis

The report is organized as following: Chapter 1 is an introduction to the thesis, and Chapter 2 explains the basics of *Cloud Computing*, including the *OpenStack* technology from which the data in the thesis is used. Chapter 3 gives the background to *log parsing*, and Chapter 4 gives an overview of anomaly detection, defining terms, and introducing general methods. Chapter 5 goes in-depth into the machine learning methods utilized in the thesis. Following this, Chapter 6 describes the method used to collect data and test different anomaly detection methods, and Chapter 7 gives the results for these experiments. Finally the thesis is concluded with Chapter 8.

1.5 Individual Contributions

Both authors contributed equally to building the execution pipeline and investigating the historical data. There was also an equal contribution in the design and execution of experiments on the TripleO infrastructure. The final, and most significant, part of the thesis was divided such that Christian was responsible for developing and analysing models using event vectors, whereas Jacob was responsible for the models using an LSTM and ensemble models.

1.6 Related Work

There have been a significant number of previous works investigating various anomaly detection methods using log data. One instance is log clustering, used by Microsoft, presented in [37]. In the article the authors utilize a parsed labeled log sequence with different weights and thereafter employ clustering algorithms on extracted feature vectors. The resulting clusters are represented by a single value. When used on new unseen data, each new *feature vector* is compared to the representative cluster values, if none of the values are within a set distance, a new cluster is formed with the vector as a representative value. With a new cluster the engineers must manually label it as an anomaly, or a normal value. On the other hand if the value can be successfully placed in a previously found cluster the system can automatically classify the vector as normal, or anomalous. The article was the first to use inverse document frequency (IDF) weighting of the input vectors, a technique we will employ in this thesis, and is explained further in Chapter 3. Moreover the clustering method is highly similar to several methods we will use for anomaly detection, described in Chapters 4, and Chapter 5.

Following the log cluster method, several other methods have been used for anomaly detection applying the same initial pipeline of window feature extracting, and weighting the feature vectors. In [28], S. He et. al. compare several existing methods for anomaly detection using log data. In addition to log clustering the authors tested the unsupervised principal component analysis (PCA) and invariants mining method. Principal component analysis classifies vectors as anomalous if the vector is far from its principal component projection using dimensionality reduction but did not give a good result. On the other hand invariants mining, which attempts to find linear relationships between the number of occurrences of among the *Event IDs*, performed well. Also used were the supervised logistic regression, decision trees, and support vector machines (SVM).

In addition to comparing methods, the effect of varying parameters in the whole pipeline are investigated. Examples of said parameters are the window type: sliding, fixed, or session, and the window size and step size. The authors found that in general, session windows scores exceeded sliding which in turn performed better than fixed. The order of effect however depended on model used. For unsupervised models long windows were in general better with window sizes of 9-12 hours performing optimally, whereas there was not a noticeable difference between the window sizes for the supervised models. The study was done on two data sets, from HDFS (Hadoop Distributed File System) logs, and logs recorded by the BlueGene/L supercomputer. The authors found there was a significant difference in performance between the data sets, and therefore it is necessary to further study the methods for various collected data. In comparison, this thesis aims to do a similar study, but with data generated by OpenStack. We also aim to further compare other unsupervised models, and also to test shorter window sizes.

Newer examples include the cloudseer method which works by workflow monitoring [67]. The authors construct a model not by using event vectors, but by using the sequence of log events themselves. They build an automaton for the workflow of management tasks. For example producing logs by booting up a VM which may interleave in different ways, but still has a pattern to it. When in use, the sequence of log messages is checked against the known automata and if there is a divergence in the workflow, such as an expected log message not arriving or the order of log messages being wrong, an anomaly is classified. This was one of the first examples where the authors tested their work on OpenStack logs which will be done in our thesis. We will on the other hand not use the mentioned method, however it serves as a precursor to the DeepLog method which we will be using.

The state of the art is the DeepLog method, presented by Min Du et. al. [19]. The method is split up into two parts, the first being log key anomaly detection and the second being parameter value anomaly detection. The first part works with the same assumptions as cloudseer. However instead of creating an automaton explicitly, which is difficult due to the many interleaving possibilities and quickly evolving source code for the cloud infrastructure, the authors propose the use of a long-short-term-memory network (LSTM) to learn the workflow, taking a window of previous log messages, and constructing a probability distribution of which log messages may appear next. For the parameter value anomaly detection, the dynamic parameters from the logs are extracted when parsing the event templates, and are used for anomaly detection as a time series. This method will be implemented, and used in this thesis, and will therefore be described in detail in Section 4.1. The article authors additionally compare the method to a PCA method, and invariants mining on a small lab environment. We wish to extend this comparison to a wider set of scenarios, and a slightly more complex lab environment.

Ericsson has previously produced several related theses. First, and most similar to ours, is [20] published in 2018 by Alexander Emmerich. The thesis explores the use of automatic anomaly detection in cloud installation logs using frequency and language models and also DeepLog. The presented result indicated a superior performance using the previously mentioned DeepLog method, however, the cloud installation logs was a substantially different data set compared to the OpenStack logs which will be used in this thesis. Further, Sarah Beschorner in [5] investigates the use of an automated hyperparameter tuner when creating anomaly detection models, and finally Janina Jäger [31] presents methods to artificially create labeled data sets.

1.7 Available Data

This thesis extensively explores the implementation of anomaly detection methods on a data center. Firstly, historical log dumps from Ericsson Research's external cloud Xerces are used to investigate the difficulties of creating models with real data by training models with the data and qualitatively judging the performance of the models. In addition to this investigation, a unique lab environment is created with the TripleO architecture [66], described more in detail in Section 2.3. In this lab environment the models are tested for several different anomaly scenarios in order to compare, and improve the methods. This is the main data source that is used in the thesis.

2

Cloud computing

Cloud Computing refers both to the cloud service provided over the internet and the software and hardware in the data centers and is used mainly for data storage and computing power [4]. The main technology enabling cloud computing is virtualization [4] which lets a single physical computing server to be split into multiple separate virtual machines. For the user this implementation is completely hidden and is identical to using a physical device. When the service is available in a pay-as-you-go format to the general public it is called a public cloud whereas if the deployment model is exclusive internally to a single organization it is called private cloud [42]. The authors in [4] present the new possibilities using cloud computing. First is the pay-as-you-go model where the consumer only pays for what is used. For example one hour of 1000 servers is as costly as 1000 hours of one server. Also the user's possibility of an instant sizable increase of computing power gives the user the illusion of infinite cloud resources which especially benefits short but large batch works. The result is additionally that the need for planning ahead for a computing expansion is no longer necessary. Another aspect new with Cloud computing is the redundancy of large capital hardware investments. In particular it has led to more developers being able to offer applications and other internet services.

2.1 Cloud and Datacenter Elements

A complete cloud infrastructure requires hardware supporting various tasks. The cloud architecture is built upon a control plane, a network, compute servers, and storage [56]. The control plane manages all aspects of the cloud infrastructure, for example authentication and scheduling of resources [15]. The compute servers are the fundamental part of the cloud, it is on these servers that the virtual machines exist, which is the real product of the cloud facility [56]. Storage is necessary for the users of the cloud, to store data for applications in the form of persistent storage, or ephemeral storage which only exists during the lifetime of a virtual machine. Storage is also necessary for the cloud infrastructure, for example to store virtual machine images [59]. Finally a network is necessary. To connect the control plane, compute servers, and storage, a physical infrastructure consisting of switches is implemented. Software defined networks are also used to communicate between servers [44]. Often a layered network is used with for instance TCP, and dedicated network nodes manage the network configuration [56].

2.2 OpenStack

OpenStack is an open source cloud platform started by NASA and RackSpace in 2012 [49]. It aims to provide an IaaS (Infrastructure as a Service), meaning it provides the virtualized hardware, such as servers, routers, or disks, as a service, and the developer creates their own applications, including authentication, storage models, and other run time aspects. OpenStack provides a compute, networking, and storage platform through a set of services, each of which have APIs that may be used to integrate them. The infrastructure owners decide which services to use, and how to use the physical hardware [49].

The main allure of cloud systems was the availability of *compute on demand*, and this is satisfied by compute nodes. The OpenStack service that provides the API to these servers is Nova. It offers access to computing resources in several forms including bare metal access, virtual machines, or containers [46]. Nova will manage the creation and deletion of virtual machines, as well as scheduling of compute resources. In addition to compute, networking needs to be set up. The OpenStack service that handles networking is known as Neutron. Neutron manages everything involving virtual networks and the access layer of physical networks. This allows infrastructure administrators to control IP addresses, firewalls, and any communication between servers [45].

Storage on OpenStack includes the services Swift and Cinder. OpenStack defines Swift as a highly available, distributed, eventually consistent object/blob store. It is used for storing unstructured data in applications that can scale quickly, and be used concurrently [63]. Cinder, on the other hand, is used by the cloud service itself as block storage for the virtual machines, and acts as persistence for virtual machines in the form of a volume. It is built on principles of high availability, recoverability, and other similar cloud concepts [12].

All other services will interface with Keystone, which is the service that manages authentication. This includes authentication of external users, for example when creating virtual machines, and between the services themselves [34]. OpenStack provides many other services, however these will give an overview as to what the different logs studied in this thesis concern. Further, OpenStack is highly modular, and each service will consist of several components that are designed much like a service with an outward facing API, and internal processes.

To give an idea of the flow between services one can look at the creation of a VM. The communication between the different services can be seen in Figure 2.1, and the following steps are a simplification of what happens when a VM is created [67, 35].

1. To begin with, a user initiates the process by using, for example the command "Nova boot", or using the horizon GUI.

2. The users credentials are then authenticated by Keystone, and a request is created and sent to Nova-api. Nova-api must once again use Keystone to authenticate this request, and once done, the request is sent to Nova-scheduler.

3. Nova-scheduler selects an appropriate *compute server*, and sends the request to Nova-compute on that server, which in turn sends a request to Nova-conductor.

4. Nova-conductor extracts the information about the instance, and sends it back to Nova-compute.

5. Nova-compute then uses the Glance service to obtain a VM image, which once again needs to be authenticated by Keystone.

6. Nova-compute then requests an IP-address using neutron. Neutron authenticates using Keystone, and provides the address, and network information.

7. Next Nova-compute requests a volume from Cinder, Cinder authenticates using Keystone, and Nova-compute gets the storage information.

8. Finally, Nova-compute releases the request to the VM hypervisor, which creates the VM.

Through this example it is clear that this is a very complicated system, with log messages for a simple request being present in several files, on different servers, indicating the need for automatic log analysis.

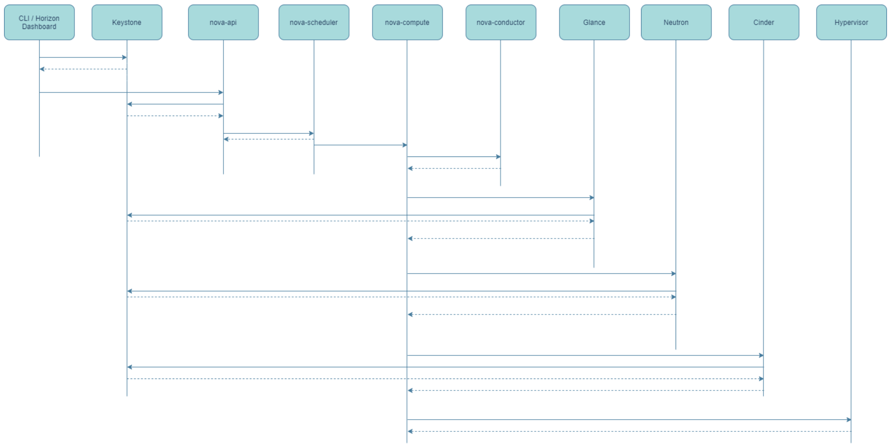


Figure 2.1: The sequence of communications between OpenStack services in order to create a VM. A lower vertical position of the arrow indicates a request is sent at a later time. The dashed arrows represent a response to a previous request.

2.3 TripleO

OpenStack on OpenStack, or TripleO, is a deployment of an OpenStack cloud on top of an existing OpenStack cloud. It is an existing project on OpenStack, and is commonly used as a way to deploy a complex cloud using OpenStack's API, instead of deploying it directly on the bare metal hardware [66]. The concept is based on creating virtual machines within virtual machines, and will make it possible in this thesis to inject anomalies into the system, without harming physical hardware. The lower level of VMs are known as the undercloud, and the upper level of VMs are known as the overcloud. Injecting errors is done by modifying the virtual machines in the undercloud, simulating an anomaly on physical hardware while a cloud is running on top of it. An illustration of the lab environment can be seen in Figure 2.2.

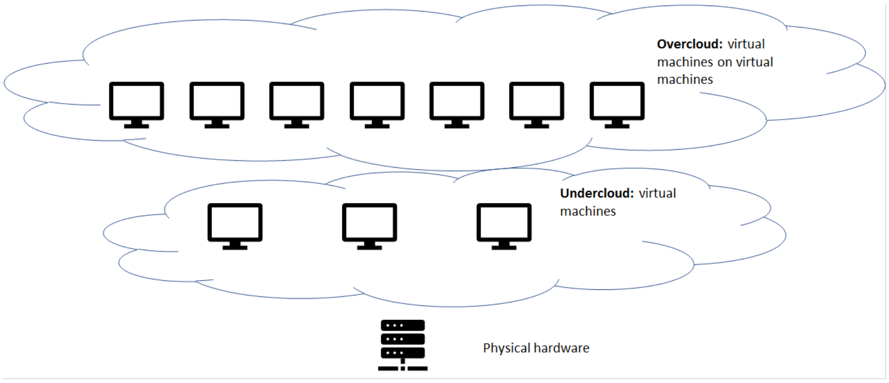

Overcloud: virtual machines on virtual machines
Undercloud: virtual machines
Physical hardware

Figure 2.2: An illustration of the TripleO architecture where a cloud is on top of an existing cloud.

2.4 Docker

Docker is a platform for developing, shipping, and running applications within containers. Containers are essentially a virtualization at the OS level, where a single kernel is split up into separate user spaces for different containers. Due to the fact that low level hardware does not need to be virtualized, containers have better resource usage than virtual machines. Containers running different applications are isolated, although can communicate through specified channels. There are several advantages with using containers. One large advantage is not being dependant on the installed software on the machine the application is running, instead all necessary software, including the OS, is bundled in the container and different containers can use different versions or configurations. Deploying containers is also a very quick process, as it is only an image, a template for a container to be built, that needs to be pushed to remote servers [16, 17].

3 Log parsing

Some data produced from logs are already structured, such as metrics. It is usually formatted as data specifying what metric is measured: the timestamp, the measurement, and various other metadata. In this case one can proceed straight to the feature extraction, or anomaly detection. However, as previously mentioned, a key part of anomaly detection using logs is parsing the log data from unstructured statements produced from logging statements in the source code, into structured log events consisting of a template with constant parts, and parameters with dynamic parts, encoding the current state of the log. Furthermore, the log can be split up into a known format, including for example the time, and level of the log. An example of the parsing of an unstructured OpenStack log, from the Nova component Nova-conductor is seen below:

```
2020-02-03 10:57:25.718 7 INFO oslo_service.service
[req-30c23595-09cd-41ee-9be5-daecd4365f6d - - - - -]
Child 37 exited with status 0
```

This log is parsed by Drain, explained in the next section, into the following format:

```
Date,Time,Pid,Level,Component,RequestId,EventId,
EventTemplate,ParameterList
```

The log event is, thus, structured as:

```
2020-02-03,10:57:25.718,7,INFO,oslo_service.service,
[req-30c23595-09cd-41ee-9be5-daecd4365f6d - - - - -
,A0001,
Child <*> exited with status <*>,"['37', '0']"
```

The event ID, in this case A0001, would be used in feature extraction to encode what log event took place. The structuring of the log data could in theory be done by using regular expressions, however, due to the vast amounts of data, this quickly becomes difficult to manage and is also computationally inefficient. The technique is also hindered by the fast development of the source code. A large open source project such as OpenStack will contain many logging statements, written by hundreds of developers, and these may be added, removed or altered frequently, necessitating a change in the regular expression templates. Modern parsing methods will, however, be robust to small changes in the message, such as spelling changes, and if the parsing is done online, a new log message will be parsed without any problems [27, 68].

Another requirement due to the very large amounts of logs produced by data centers is that a log parsing method is fast. Furthermore, to be used with production data it is important that it can handle streaming data, and parse online. This is the major breakthrough found with the Drain method, which is a performant method that parses logs online. The previous state of the art methods have often seen the problem as a classical clustering problem that train on offline data, and attempts to use streaming data have been slower than the offline versions [27].

Several log parsing methods are compared in [68]. It finds that Drain is both the most accurate in the tested data sets, and contains the least amount of variance. However, there are still some data sets where other parsers have better accuracy. In terms of efficiency, it was found that Drain is the most efficient along with IPLoM (Iterative Partitioning Log Mining), an offline parsing method [68].

3.1 Drain

The state of the art fixed-depth tree-based, online log parsing method Drain, was introduced in [27]. Drain works in five steps: Preprocess by Domain Knowledge, Search by Log Message Length, Search by Preceding Tokens, Search by Token Similarity, and Update the Parse Tree.

The first step "Preprocess by Domain Knowledge" involves manually crafting regular expressions to match in the logs, such as IP-addresses. Usually very few are required, but they may improve accuracy [27]. The next step, "Search by Log Message Length", is the first branching in the tree. Here log messages are partitioned based on their lengths, in terms of the number of tokens, or words. The assumption is that all log events always have the same length, which is most often the case, and in the few cases that it is not true, the authors propose that post-processing can solve the problem.

The following step, "Search by Preceding Tokens", involves separating log events by single tokens, starting from the first word of the message, and going further as the depth of the tree increases. The assumption here is that most log messages have constant parts in the beginning of the message. There are cases when this is not true, therefore, if the token contains digits, Drain does not match on the token, and instead inserts the special token * matching with anything. Similarly, if the token has more than a certain number of children, meaning there are a lot of different log messages with the same preceding tokens, but differ in this specific one, Drain will replace the token by the dynamic wildcard *.

"Search by Token Similarity" takes place when the parsing has reached a leaf node, the depth of which is given as a parameter. A leaf node contains several log groups each defining an event, and this step finds a suitable log group for the message. The log message is compared to the template using the following equation:

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

where seq_1 represents the log message, and seq_2 represents the template for the log group, $seq_1(i)$ is the i'th token in the log message, and n is the length of the log message. The function equ is defined, for two tokens t_1, t_2 as:

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

If the $simSeq$ value is larger than a specified similarity threshold parameter, the compared log group is returned as the one the message belongs to.

In the final step, "Update the Parse Tree", Drain adds the message to a log group if the previous step deems the log message to belong to a specific group. It then updates the template, such that if some token is not shared between the log message and the current template, it is replaced by the * wildcard. If no log group matches, a new one is created.

There are three parameters necessary to execute Drain. The depth of the leaf nodes is used to determine when to end the "Search by Preceding Tokens" step. The max number of children is used in step "Search by Preceding Tokens", in order to determine if the token is constant or dynamic. Finally, the similarity threshold is used to determine if a log message belongs to a specific group.

Drain was first developed into an open source program by the LogPai authors [28], but was improved by IBM engineers to include state persistence, and to better handle streaming data[1]. This is the implementation used in this thesis.

3.2 Feature extraction

Once the logs are parsed, one has a large sequence of log events. The question then becomes how one may use this data in anomaly detection models. This is the

[1] Available at https://github.com/IBM/Drain3

process of choosing what features of the data to use, and how to use the sequence of log events as input to machine learning models.

The simplest methods are the fixed window and the sliding window, whereby the input vectors to the models are a vector where each component of the vector corresponds to the number of occurrences an event had during a window of time. The fixed window has a length, and no overlap, whereas the sliding window has both a length and a step size, leading to an overlap between windows. For the sliding windows, the step is often chosen smaller than the length, which means there is an overlap between the windows. As stated in Section 1.6, the performance of sliding windows has previously been observed to be better than fixed windows, possibly due to the fact that the shorter steps produce more data [28]. The number of windows is determined mostly from the step size, as the start of each window is independent of the window length. Figure 3.1 illustrates what may happen to a sequence of log events as it is divided up into sliding windows, where each window has a specified length, and the next window starts at the first log event which is a step size away from the start of the previous window, and these sliding windows are transformed into event vectors using the number of each event in the window. In the remaining chapters of the thesis, the window length will be known simply as the *window size*.

The session window counts the occurrences of each event with a respective identifier. Previously mentioned in Section 1.6 was that session windows performed better than sliding windows, which could be due to the fact that all of the log messages in the window are related, and can therefore produce a higher correlation [28]. In OpenStack logs, a suitable identifier is the request ID. A problem with this, however, is that the request ID is local to the specific OpenStack service, and therefore may not capture the state of the whole system, and is difficult to use for root cause analysis. Another alternative is the instance ID, however, this identifier is only present in the log messages pertaining to a specific VM, and therefore one is imposing a large restriction on the data that can be used, as most of the produced log messages will not have an instance ID.

Further alterations can be made to the event vectors before they are used in the model. For instance in [37], duplicate events are only counted a single time, and the final event vector is weighted using IDF weighting. IDF weighting stands for inverse document frequency weighting, and is a scheme that applies larger weights on terms that are less frequent. The technique is widely used in natural language processing due to the fact that frequent words, such as "the", do not contain much information and are seen as unimportant. Assume there are N documents, the IDF weight for term i is defined as, $IDF_i = \log_2(N/n_i)$, where n_i is the number of documents that include term i [36]. For log analysis the number of documents can be interpreted as the number of log sequences, or windows, and n_i, can be interpreted as the number of of log sequences which contain event *i*. Weighting, or normalizing, inputs is a common step in anomaly detection, and other techniques for doing this will be described in Chapter 5.

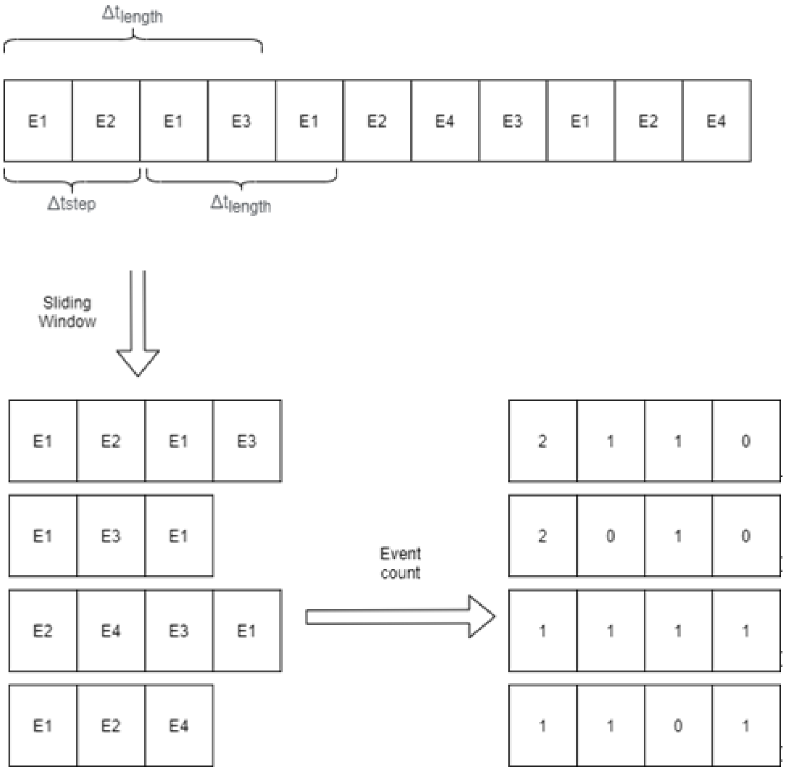


Figure 3.1: An illustration of log key feature extraction with sliding windows.

4

Anomaly Detection

To define the terms outlier or anomaly is difficult, and there is no universal definition [7]. In [11], the authors simply define an anomaly as "patterns in data that do not conform to a well defined notion of normal", however this requires a second definition of "normal". In Figure 4.1, one can see two clusters that are much smaller than two large clusters, intuitively these large clusters would be "normal", and the smaller ones would be outliers, however it is not a precise definition. In [7], a definition of "normal" given by the authors as common is “An observation which deviates so much from other observations as to arouse suspicions that it was generated by a different mechanism". As described earlier, in the context of large scale distributed systems, there may be errors in the systems that do not lead to a failure and may therefore be a part of normal behavior. One example of an anomaly is if the users of the cloud experience a slower than usual response, which is an example of a *performance anomaly*. Other anomalies such as a server being shut down, may manifest as novel log events, or in an unusual sequence of logs.

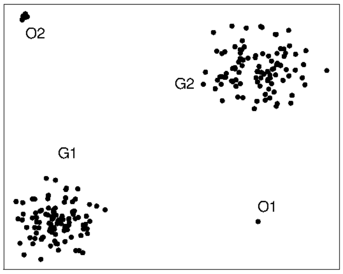


Figure 4.1: Depiction of the relation between the outliers O1 and O2 and the two clusters G1 and G2 [50].

Within these definitions, anomalies can be categorized further [11]:

Point based anomalies: The simplest kind of anomalies are point-based anomalies. This is when a single data point is an outlier with respect to the rest of the data. For example an extreme value in a time series, or a data point that is far from clusters of other data points.

Contextual anomaly: Another subcategory is the contextual anomaly, which is the case when data points are an outlier in a certain context, but not in other contexts, or in a global sense for the whole data. This is common with for example seasonal time series, where an outlier in a certain time span would not be an outlier if it was in a different time span.

Collective anomaly: Finally, collective anomalies, is when a collection of data points are considered an anomaly, however the individual data points may not be outliers. This is relevant in the context of this thesis, as an anomaly may be present for a certain time period, leading to a collective anomaly being present in the data, or that a sequence of log messages is an anomaly, but the log messages themselves are not anomalies.

The term outlier and anomaly are often used interchangeably, although outlier is sometimes considered a wider term encompassing noise, and other expected deviations in the data set. Another concept related to these terms is novelty detection. Novelty detection, as the name suggests, attempts to detect data points that have not previously been seen. Many anomaly detection methods presented in this thesis are also novelty detection methods, as they are trained using data that does not contain any anomalies. This raises an important issue involved in anomaly detection, which is the fact that when the behaviour of the system changes, so does the definition of outliers, and models which were trained to the old behaviour of the system may recognize normal data as anomalies. In the data center, the software is occasionally updated, and care may have to be taken to ensure the efficiency of the models with the updated software.

Anomaly detection is a very old subject, and, hence, there exist countless methods with different use cases. A common technique among many methods is to model the data, and detect anomalies as points that are not well modeled, although exceptions exist, such as the isolation forest method, described in Chapter 5.

If the data is temporal, a distinction is made between estimation methods, and prediction methods, where prediction methods only use previous data points to predict the current data point, and estimation methods use past, current, and subsequent data [7]. As prediction methods can handle streaming data, they are often necessary. A simple method to define an anomaly with these methods is to take the predicted value $\hat{x}_t$, the actual value x_t, and compute the absolute error $|x_t - \hat{x}_t|$, and if this value exceeds some threshold, also defined by the model, it is an anomaly. For time series, to build a model one could use advanced machine learning methods such as LSTM networks, or classical methods such as ARIMA models [7].

Clustering Methods One example of an anomaly detection method for data that is not time dependant are clustering methods. There are a few ways to define what an anomaly is in the context of clusters. The simplest way is to define an anomaly as a point that does not belong to a cluster, this may be sub-optimal, however, as algorithms are usually created to find clusters, and not anomalies. A second method is to define anomalies as the data points which have a distance to their centroid which exceeds some threshold, where a centroid is some definition for the middle of the cluster such as the component wise mean of all points in the cluster. Neither of the previous methods will work, however, if there are enough anomalies in the training data that they form a cluster themselves. In this case anomalous clusters can be defined as small and sparse clusters, or due to the reduction in number of items to manually label after clustering, the clusters can be manually labeled to be anomalous [11].

Density-based methods There are several related methods to clustering, such as k'th nearest neighbor algorithms, or more general density based methods. Using a k-nearest neighbor algorithm, outliers are defined as points such that the distance to its k'th nearest neighbor exceeds some threshold. Similarly, density-based methods estimate the density, which can be done with a k-nearest neighbor algorithm, and define anomalies as points with low relative density. Density-based methods may work poorly for data with naturally varying density. A disadvantage to clustering methods is that there is not a single centroid to compare to; in order to test the models, the density would have to be calculated with all test data, or including the test point in the training data. Furthermore, the computational complexity is often $O(N^2)$ for these algorithms, whereas clustering is typically $O(N)$ [11].

Statistical methods Another method of detecting anomalies is through the use of statistical modelling. One assumes that the data is generated through some distribution, and employs statistical tests to check if it is likely that a single data point was generated by that distribution. These are somewhat simple techniques, that provide a lot of information, as well as being able to be used in an unsupervised settings, however the assumption that the data is generated by a simple distribution is often not the case for real world data sets, and choosing the correct statistical tests is not always simple [11].

Classification based methods Many machine learning methods such as SVM's, or neural networks can function as a classifier, and therefore given labeled data could classify a point as a normal point, or as an outlier [11].

A common technique used in anomaly detection is dimensionality reduction. In multivariate time series, dimensionality reduction is used to decrease the number of variables, in order to utilize univariate methods on uncorrelated variables [7]. In non-temporal data dimensionality reduction may be used in itself as an anomaly detection method, whereby anomalies are points that have a large projection on the dimensions which are not principal components [28].

All of these methods operate as point anomaly detection methods, however they can be adapted to be used for collective anomalies. For time series this consists of either finding full time series that are anomalous, or finding subsequences that are anomalous. The easiest way to transform the latter problem into a point based anomaly detection problem, is by using sliding windows. The length of the window is often unknown, and part of modelling will include searching for window sizes. There are methods that automatically detect subsequence length, however they are not used in this thesis [11, 7].

4.1 DeepLog

As described in Section 1.6, one of the methods used for anomaly detection using logs is DeepLog. It is split into two separate algorithms, a *log key anomaly detection* model, and a *parameter value anomaly detection* model. The point outlier detection method used in the *log key anomaly detection* system is not described by the previous methods in this section, as it attempts to use discrete events instead of a continuous variable in which a value can be compared to a threshold. These models are connected in such a way that the *log key anomaly* detection model will first classify a log message, and if it classifies it as normal, the *parameter value anomaly detection* model is used to determine if an anomaly can be detected from the parameters. Both models use an LSTM network, which is described more in detail in Chapter 5.

4.1.1 Log Key anomaly detection

In order to detect anomalies in the log execution path, the authors model the probability for an event to occur, given the previous log messages. The probability distribution is created by training an LSTM network in a multiclass classification setting, to classify the event of the subsequent log message, given a window of h previous log messages. The threshold used is a parameter g, where the events are ranked by their respective probabilities, and if an event is not in the top g most probable events, it is an outlier [19]. This is illustrated in Figure 4.2 in which an LSTM network outputs a probability distribution, given a sequence of log events.

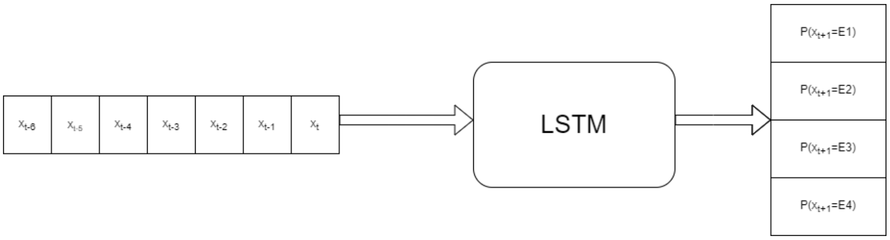


Figure 4.2: An illustration of anomaly detection using the DeepLog architecture. x_t is a sequence of log messages, each one of four events. A window of six is used to predict the probability of the subsequent log message being a certain event.

4.1.2 Parameter value anomaly detection

For the *parameter value anomaly detection* DeepLog creates a separate model for each log event. Each given log key has a set of dynamic parameters, and these parameters form a multivariate time series. Consider the message below :

```
GET status: 200 len: 123 time: 0.0129128
```

Three parameters can be extracted from this message, the status, length, and time, giving a parameter vector of [200, 123, 0.129128], and a fourth including the relative time to the previous log message. As all log messages of this event will have these four parameters, a time series will be formed with the same parameters, and anomaly detection can be done on this time series.

DeepLog simply uses the prediction methods described previously, using an LSTM to predict the next value in this sequence. The authors then model the mean squared error between the predicted value and the observed value as a Gaussian sequence. The log message is classified as an anomaly if the error is not within a high confidence interval of the distribution, 98-99.9% was used in the paper [19].

5

Machine Learning

Today big data is everywhere. The term big data refers to data sets on such massive scale and complexity that traditional analysis tools can not be efficiently applied, such as personalized data collected by current social medias [6]. The ability to process big data is where Machine Learning, ML, comes into the picture and has become increasingly used. It provides a class of methods which can automatically detect patterns in order to predict, or make decisions as classification, even for large sets of data [25]. This comes from one of the main characteristic of ML which is in its ability to learn by experience. By detecting certain correlations in the data set it creates a mathematical model of the data which can continuously improve with more experience, in this case data samples [43].

For problems previously thought to be too complex or time consuming ML can be used, where image and speech recognition and credit card fraud detection are some given examples [25]. On the other hand ML is not some kind of magic which can blindly solve problems. For example, different artifacts in an image data set can give undesirable patterns in training. In [57] a highly accurate ML neural network was created to classify if the image contained wolves or huskies. In reality the created model classified the images by looking if there was snow or not in the background. For the images of the wolves there often existed snow, whereas for the huskies there was no snow. Furthermore, the complexity of ML models makes it difficult for humans to discover these undesirable patterns and caution is needed when using ML. The husky classification is an example of supervised learning which is one of the four main types of ML.

Supervised Learning: The training data set is of the form $\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=0,1...,N}$, where the data samples are characterized by an input feature vector, $\mathbf{x}_i$, and an output, $\mathbf{y}_i$. The learning algorithm task is to learn the mapping function between the feature vectors and its respective target vector to be able to predict the output of previously unseen inputs [43]. Basically, the entire training data has been labeled either as a normal data point or as an anomaly and the model makes a binary classification for previously unseen data points. However, this method is rarely used for outlier detection since manually labeling an entire data set is extremely time consuming [28].

Unsupervised Learning: The input data consists only of feature vectors without any corresponding target classifications or labels. This type of algorithms looks for structures and patterns in the data set. Clustering is such an example where the aim is to discover different groups among the input data according to its attribute similarities. The model is expected to know what data points are normal and what are outliers in the training data set after the learning process [43].

Semi-supervised Learning: A combination of supervised and unsupervised learning where the training data usually consists of a both unlabeled and labeled data. The training algorithm tries to learn the input-output mapping by exploiting both information [2]. In the field of outlier detection, semi-supervised learning refers to novelty recognition [47]. It is when the training data only consist of normal behaviour points and the trained model is then used for new observations to find outliers. If an observations then differs within a given threshold from the normal points it is classified as an outlier. It can be considered semi-supervised as this approach only needs the normal data to be classified.

Reinforced learning: Lastly, reinforced learning uses trial and errors to determine actions to maximize a reward [2]. Since it is not related to this thesis we will not go further into it.

Following, various ML techniques related to the objective of the thesis will be explained. Since the concept of Neural Networks and Deep Learning differs against conventional ML it has its own section. The conventional ML methods which will be described are all related to outlier and anomaly detection. Thereafter, there is a section of the theory about model training for ML algorithms.

5.1 Conventional Machine Learning Algorithms

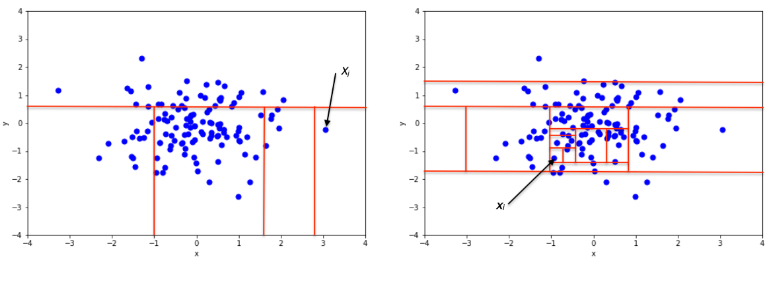
5.1.1 k-Nearest Neighbours

K-Nearest Neighbours, KNN, is mainly used to classify data points in relation to samples in its vicinity. The classification of a new data point is made according to the majority of classes belonging to the k closest neighbors. This method therefore depends on the choice of k and the distance between samples is calculated using the L_2-norm [23].

Different variants of KNN can be made into basic unsupervised proximity-based methods used for outlier detection [32]. Three alternative measurements of the outlier score for each data point is the distance to the k:th nearest neighbor, the average distance to the k nearest neighbors and the median of the distance to the k nearest neighbors.

5.1.2 Isolation Forest

In [38], the authors describe an unsupervised anomaly detection algorithm which differs from other anomaly detection models. Called Isolation Forest it does not profile the normal state. Instead it explicitly isolates outliers by separating data points from the rest. The method is based on the Random Forest algorithm by building up a tree structure, iTree, similar to decision trees. The overlying methodology is taking partitioning actions in the dimensional space of the data until the chosen point is isolated by itself in the final partition. To isolate a chosen data point, the algorithm selects randomly a feature where it splits the space randomly between the maximum and minimum value of the selected feature. This action is performed recursively until the chosen point is isolated. To determine an outlier it takes advantage of the properties that outliers are few and different from the ones belonging to the normal state. Therefore, the number of partitions required to isolate an outlier should be less in comparison to the normal ones. A 2D visualization of the idea is shown in Figure 5.1 where in general the outlier, x_j, requires less partitioning than for the normal state point, x_i.



(a) (b)

Figure 5.1: Illustration of Isolation Forest where the number of random space partitions required to isolate an outlier, (a), is less than that of normal points, (b) [8][9].

In the example the attribute, either x or y axis, is chosen at random and then is split randomly into two portions recursively until only the point remains. Splitting the space recursively can be represented by a tree structure where the depth of the tree is the number of partitions required for the isolation. Generating multiple trees randomly, on average, an outlier will have a lower depth than normal instances [30].

An advantage of the Random Forest algorithm is that it does not utilize any distance or density-based measures which removes costly computations.

5.1.3 One class SVM

A support vector machine (SVM) is a supervised algorithm, used to classify data. The classification is done by choosing a hyperplane such that the distance to the nearest point on each side of it is maximized. Although a hyperplane represents a linear classifier, non-linear classification can be done by mapping the inputs to higher dimensions using a so-called kernel [62].

One class SVM differs from the ordinary SVM in that instead of seperating points with a hyperplane, a hypersphere is used. The most basic formulation of one class classification using an SVM is to choose the hypersphere with the smallest radius that also encompasses all training data. However, this cannot be used if there are outliers in the training data, therefore the optimization problem is altered slightly, adding a parameter that controls the fraction of outliers in the training data. The hypersphere will now contain all data in the training set apart from a certain number of samples controlled by this parameter [48].

5.1.4 LOF

Another algorithm for finding outliers is Local Outlier Factor, LOF, and it is proposed in [10]. Here the data points are not binary classified as an outlier or non outlier. Instead the algorithm computes a score, called outlier factor which indicates its degree of abnormality. The algorithm is density-based, however, the approach is to examine the density ratio between the sample and its neighboring samples. The idea is that samples with considerably lower local density than the samples in its vicinity are more likely to be outliers. More exactly, the outlier factor is equal to the ratio of the average local density for the k user defined nearest neighbors, and its own local density. For outliers it is expected to have a much lower local density than its neighbors while a normal instance should have approximately the same [13]. The main advantage of this approach is that it takes into consideration both local and global properties which makes it perform well for data sets were outliers have different local surroundings [13]. Figure 5.2 depicts a data set where the outlier factor is calculated using LOF.

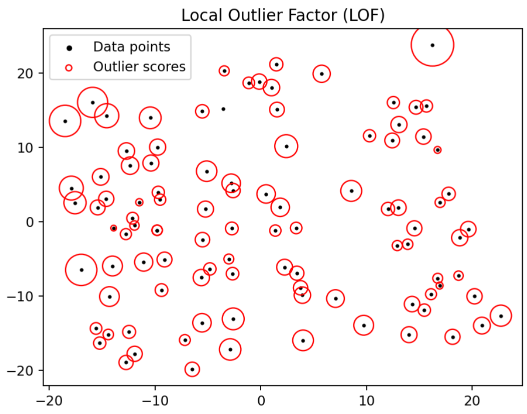


Figure 5.2: Plot showing the LOF algorithm. A wider circle indicates a higher value of the local outlier factor meaning a higher density ratio between the given sample and its neighbouring samples resulting in a more abnormal instance.

5.2 Artificial Neural Networks

Inspired by the human brain, Neural Networks, NN, consist of a collection of nodes called artificial neurons in a graph [1]. Each node can receive and send signals to other connected nodes which models the synapses in our brain. The strength of the synapses are modeled by a weight of the connection between nodes, i.e., the edge weight in the graph [25]. There are three types of artificial neurons [25]. The data is entered to the network through the input neuron. Following the input neurons are the hidden neurons where the computation is performed. The term hidden comes from that the user can not see the output of these nodes. Lastly the result leaves the network through the output neurons. For NNs the neurons in the graph are usually placed in layers of different sizes where the nodes in the first and last layer is the input and output neurons respectively. The layers in between consist of the hidden nodes and when a NN has multiple layers the learning process is often called deep learning [1].

When data is sent through the network, each neuron receives values from its connected neurons in the previous layer. Using the connections weight, the weighted sum of the received values are computed, added with a constant called bias term and then sent through a non-linear function called an activation function [24]. The result of the mathematical operations becomes the given neurons value which are propagated to the next layer. The process is shown in Figure 5.3 and can be expressed as

$$S = b + \sum_{i}^{N} w_i x_i \tag{5.1}$$

$$o = f(S), \tag{5.2}$$

where o is the resulting value of the neuron, f is the activation function, w are the weights and x the input/received value, b is a bias term which is not shown in the figure, but is almost always used. The importance of the non-linearity of the activation functions is to make it possible for the network to approximate non-linear relations [24]. Different activation functions are often used for different kinds of neural network layers, where some commonly used are Sigmoid, RELU and tanh [1].

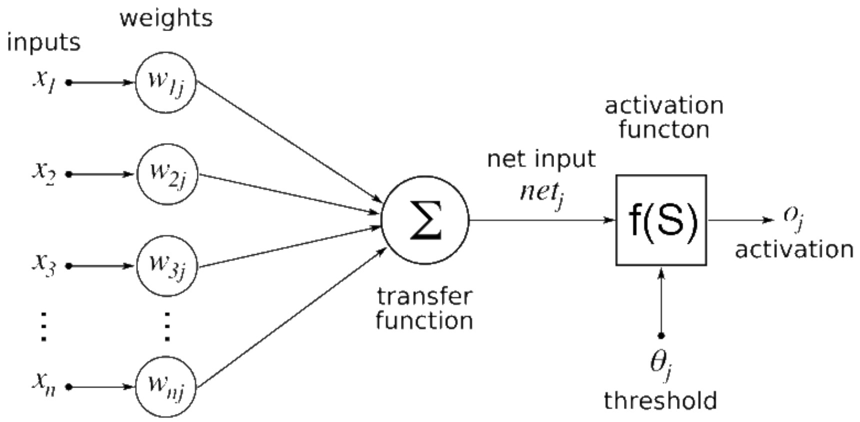


Figure 5.3: The mathematical operation which gives the resulting value of each node in an artificial neural network. The value is the sum of a bias and the weighted inputs through an activation function [58].

The learning of the network occurs when changing the weight values for the connections between neurons. A widely used mathematical method to train a NN consisting of neuron layers is called forward and backward propagation which is explained in [24]. The training data, $\mathbf{x}$, will be passed through the network and result in an output value, $\mathbf{y}$, which is the forward propagation part. Using a loss function, the training data's true label, $\hat{\mathbf{y}}$, will be compared to the NN's classification and result in a loss value $L(\mathbf{y}, \hat{\mathbf{y}})$. In the back propagation the resulting loss value is minimized by a gradient decent variant in terms of the weights of the last layer. This is then re-

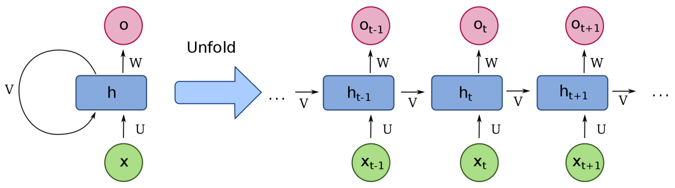


Figure 5.4: An example of a recurrent neural network, which is initially illustrated as a connection from a node to itself, but afterwards can be seen to represent a connection through time [61].

peated backwards. The number of forward- and backward propagation done during training is called epochs and it must be set during the training of NN. However for this training method, forward and backward propagation, reaching a global minima is not guaranteed since the problems usually are non-convex. Lastly, because of the need to compare predicted output and the true output in the training phase these NNs can only be used for supervised learning, which is true for the majority of NNs, [1].

5.2.1 Feedforward NN

The simplest kind of neural networks is the Feedforward Neural Network, FNN, where the data only moves forward in the network without any cycles or loops [24]. Each neuron is connected to all the neurons in the next and previous layer which gives a fully connected network. The hidden and output neurons do the mathematical operation previously mentioned using the values of the previous layer and the connection's weight. The value of the output node is then the final result of the data through the FNN [25]. If the output variables are continuous, the neural network performs regression, whereas if the output is part of a discrete set of values it performs classification [1].

5.2.2 Recurrent Neural Networks

Recurrent neural networks are specially designed neural networks built for handling time series data. They are able to process much longer sequences of data than classical neural networks would be able to [25]. This is achieved by creating connections between the outputs, or hidden units from the previous sample to the current sample. This can be seen in Figure 5.4, in which there is a connection between the outputs of the single hidden layer.

5.2.2.1 LSTM One problem with the basic RNN presented is that gradients have a tendency to vanish when performing back propagation through time on long sequences, and therefore the network can not model dependencies far back in time. The solution to this is using an LSTM, a more advanced type of recurrent network. LSTM, which stands for long short-term memory, consists of a cell, an input gate, an output gate, and a forget gate. The cell remembers information for arbitrary time lengths, information has the ability flow unchanged between cells, which offers the advantage for LSTMs, as gradients can also flow unchanged and therefore do not vanish. In order for the network to learn which time steps have "important" information, and which do not, the three gates can regulate the information that flows between the cells [40].

5.2.3 Autoencoder

The idea of an autoencoder is first to encode the input data and then using a decoder reconstruct the input data as close as possible, hence its name. The encoding often consists of a dimensionality reduction where as much information as possible is preserved. The decoder is optimized without any knowledge of the hidden layers in the encoder, only having information of the input and output data of the encoder. Since some information is lost in the dimension reduction a perfect reconstruction is not possible. This type of NN's loss function uses a sum of squared differences between the input and the output [25].

In outlier detection autoencoder can be useful because it can be used in unsupervised learning [24]. The idea is that the autoencoder models a normal state during the training phase and therefore data points belonging to a normal state will be better reconstructed than an outlier. Using a threshold on the loss function for the reconstruction for a data point the autoencoder can, hence, be used to classify outliers.

5.3 Ensemble models

Ensemble learning is the process of using several models to produce a better result than each of the individual models [21]. One example of using ensembles is the previously mentioned Isolation Forest algorithm, where several trees make up a Forest [30]. There are several ways to make an ensemble model, the simplest using the average value for regression, or a voting system for classification. More complex systems include boosting, where each new model is trained to better classify the instances that were misclassified by the previously trained models. Empirically models that have different behaviour often achieve better results after forming an ensemble, compared to forming ensembles with less diversity [21].

5.4 Model Training

There are multiple essential steps and elements of the training phase required to get a good machine learning model. Firstly, the data set often needs to be processed before being used for training. Thereafter, during the training phase there are several important methods to get a good fit and prevent overfitting. Lastly, an appropriate evaluation method and metric needs to be chosen [24].

5.4.1 Feature scaling

A part of the data preparation is normalization, also called feature scaling. Since the range of values for the different features can differ significantly scaling is needed for most machine learning algorithms [53]. For example, many ML techniques use Euclidean distances and if a feature has a wide range the distance will be unproportionally affected by this feature. One of the normalization methods specifically developed for log parsing analysis is inverse document frequency weighting, IDF, explained in Section 3.2. There are multiple other widely used feature scaling methods used for ML.

Min-max normalization is one of the simplest methods and scales every feature individually such that the values are in the range [0,1]. The general formula is

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$$

where x' is normalized value and x is original value. Using minimum and maximum values in the formula makes the method sensitive to the presence of outliers and can in many cases make outliers not present after the transformation [53].

Z-score normalization, also called Standardization, scales the data so that each feature has zero mean and unit variance.

$$x' = \frac{x - \mu}{\sigma},$$

where μ is the mean of the data, and σ is the standard deviation. Because the range of each feature is of different magnitude the scaled distribution will differ from the original. Moreover since the outliers affect the calculation of the mean and standard deviation the Z-score normalization is sensitive to outliers. However, in many cases the few large marginal outliers will still be present in the transformed data set [53].

Robust scaling bases its centering and scaling around percentiles and median unlike the previous methods. For example using the *IQR*, which is the difference between the value at the 75th percentile, and 25th percentile, one can scale as:

$$x' = \frac{x - m}{IQR},$$

where m is the median of the data. As a result the method is much more robust to a few number of large outliers. The outliers will hence still be present after the transformation and the distribution will be approximately similar [53].

5.5 Overfitting

One of the main characteristics of machine learning is that it is supposed to perform well on new unseen data which is called *generalization*. If the entire data set is used for training then minimizing the training error will simply be an optimization problem. Therefore, in machine learning the goal is generally to minimize the generalization error, also called test error. This value is the expected value of the error for a new data point. To accomplish this the available data set is usually split into a training set and a testing set which is explained in Section 5.5.4. When training, for many ML algorithms there comes a point where further optimization of the training error comes at the cost of the test error. Overfitting then occurs when the difference between the training error and test error is too large which is depicted in Figure 5.5 which means that the model is optimized to give a low loss for the training set, but will give a larger loss for unseen data.

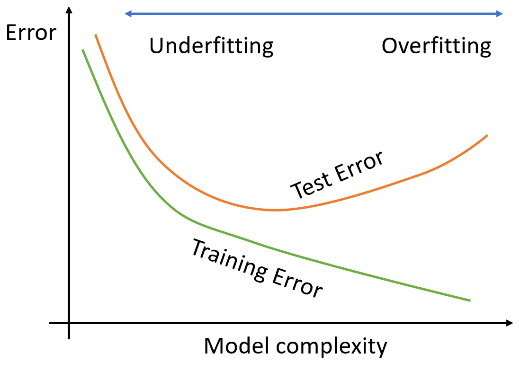


Figure 5.5: Example of how the relation between training and test error may look when training machine learning models, as the complexity of the model increases.

5.5.1 Hyperparameters

In many machine learning algorithms, hyperparameters are parameters that need to be set prior to the training phase. By contrast other model parameters are optimized during the fitting of the training set. One type is the algorithm hyperparameter which controls the learning process, e.g., learning rate and batch size. There are also those who are a part of the model selection. A bad choice of hyperparameters can result in an overfitted model, for example with a large number of hidden units in a neural network, or in polynomial regression choosing a high degree. Poor choice of hyperparameters may also lead to bad model performance, for instance choosing a neural network model which is too small, and does not have the capacity to learn the features of the complicated data set. Hyperparameters for neural networks includes the number of layers and the number of nodes in each layer.

5.5.2 Dropout

A simple and effective way to reduce overfitting in neural networks is to use a technique called dropout. Randomly dropping the weights going into the next layer will reduce the number of parameters of the network. Therefore during the training phase the networks ability to match the training set is decreased which means less probability of overfitting [24].

5.5.3 Early Stopping

Another method to prevent overfitting is early stopping. Many neural networks are based on iterative optimization methods, where training takes place over several epochs. For each epoch the model is improved in terms of the loss on the training set. However, there comes a point where the reduction of the training loss comes at the cost of the test loss, early stopping is to stop training at the point where the loss in a validation set starts to increase [1].

5.5.4 Train, Test and Validation Sets

To get an unbiased evaluation of the final model the data set can be split into three subsets. The training set is used for the model learning, that is to fit the classifiers "internal" parameters. Then the validation set is used for an unbiased tuning of the hyper parameters, for example choosing the number of layers in a neural network. Lastly, because the validation set was used in tuning of the hyperparameters it is not a good measure of the generalization of the model. Therefore the test set is used for measuring the performance of the final model [1]. The pseudo-code below shows an example of the usage of the different data sets.

```
#Splitting data set
training, validation, test = split(data)

#Tuning classifier hyperparameters
for parameter in hyperparameters:
    model = fit(parameter, training)
    performance = evaluate(model, validation)

#Testing the final model
```

```
model = fit(train)
final_performance = evaluate(model, test)
```

5.5.5 Performance Metric

There are several commonly used performance metrics for a binary prediction with classification models [25]. Assuming:

Positive class, p is the set of outliers.
Negative class, n is the set of normal points.

Then four prediction sets can be defined as:

True positives, TP. Number of correctly predicted outliers.
True negatives, TN. Number of correctly predicted normal points.
False positives, FP. Number of false alarms, i.e., normal points predicted as outliers.
False negatives, FN. Number of missed outliers, i.e., outliers predicted as normal points.

which also is illustrated in Figure 5.6.

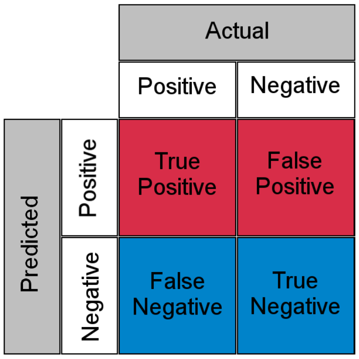


Figure 5.6: Confusion matrix [14]

From the given definitions of positives and negatives, the following benchmark metrics are defined as [25]:

$$\text{Positive predictive value, } \textit{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

$$\text{True positive rate, } \textit{recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

$$\text{False positive rate, } \textit{fall-out} = \frac{\text{FP}}{\text{FP} + \text{TN}}$$

$$\text{F-score} = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$$

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

The true positive rate, *recall*, represents the ratio between predicted and the total number of outliers i.e. the prediction accuracy of the outlier data set. A small value of recall means bad accuracy and there are a large number of outliers missed by the model. The false positive rate, *fall-out*, on the other hand is the ratio of classified outliers to the number of samples in the negative class, i.e. the complement of the accuracy for the negative class. A large value therefore means bad accuracy which leads to many false alarms. Receiver operating characteristic, ROC, analysis studies the trade off between the true and false positive rate and can be visualised by a ROC curve where they are plotted against each other [52]. A visualization of the characteristics of ROC curves for classification models is shown in Figure 5.7. For outlier detection the desired value is a large true positive rate in proportion to false positive since it means a high accuracy for finding outliers and low ratio of false alarms. However, there is always a tradeoff when choosing a model of getting a larger true positive rate, or a lower false positive rate.

Another performance metric is the precision which also measures a false alarm value but it includes prediction rate in both outlier data set and the baseline [52]. Furthermore the F-score, which is the harmonic mean value of recall and precision combines the two metrics into one. It is approximately the mean value when the value of recall and precision is close to each other. Lastly the benchmark value is the accuracy which gives an indication of the correct hit rate of the predictions[25]. However, accuracy can give a misleading value when using unbalanced data sets [52]. For example, having an disproportional number of positives the highest accuracy is achieved by predicting every data point as a positive.

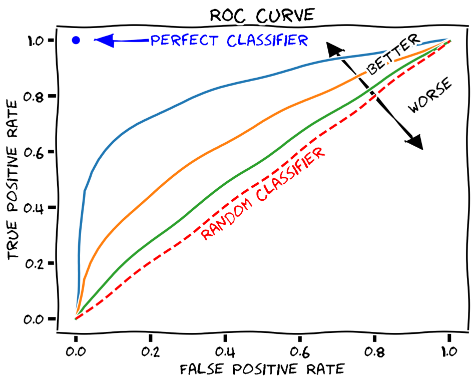


Figure 5.7: A ROC curve which shows the trade off between the detection rate, TP, and the false alarm rate, FP, for better and worse classifiers [55].

6

Method

The methodology chapter will be divided into several parts, detailing both the experiments done on the OpenStack on OpenStack environment, the pipeline used for anomaly detection, and the methodology used to analyze different models and parameters used in anomaly detection. Firstly, in Section 6.1, the lab environment is introduced. Following this in Section 6.2, the specific anomaly scenarios and log collection are described. Subsequently, the first step of the pipeline, the cleaning and parsing of the data is described in Section 6.3. The application of historical data, and on data collected from the experiments is outlined in Sections 6.4, and 6.5 respectively. The last step of the pipeline, inference is outlined in Section 6.9.

6.1 Lab environment setup

To get labeled logs, a lab environment was built using OpenStack on OpenStack, i.e., TripleO, which is described in Section 2.3. It is a cloud environment where the OpenStack services are run on VMs pretending to be physical servers. The lab environment gave us better control over the cloud infrastructure than we could get with the production cloud. It is easier to produce logs representing a normal state with TripleO, since the servers were not physical machines but controllable VMs. Further, as a result of the servers being virtual we could inject errors into the system and produce labeled data sets. Examples of errors that may be injected into the virtual servers are: shutdown of a server, full disks and network delays. By using the lab environment, we also avoided affecting the production cloud and the physical servers themselves when carrying out the experiments.

The TripleO system consisted of 12 VMs, each with 4 cores, 40GB disk memory and 16GB of RAM. Further, the higher level OpenStack services, for instance Nova-api, were deployed as individual containers on each VM using Docker. Some of the specific services on the VMs are shown in Table 6.1, although many more containers were used in the deployment. Finally the VM known as the HA-node runs a proxy service called HAProxy. The server functions as a load balancer, and all the requests between the different VMs are passed through the HA-node.

Table 6.1: Shows the setup of the created TripleO environment and the OpenStack services running on the servers.

	Control	**Compute**	**Storage**	**Network**	**Ha node**
#servers	2	3	2	2	1
Services	Nova -api -conductor -scheduler Keystone Neutron	Nova Compute	Swift	Neutron	HA-proxy

6.2 Log collection

6.2.1 Decide which logs to use

Although the eventual goal with the anomaly detection systems will be to deploy the anomaly detection models for all OpenStack services, some were prioritized in this thesis due to time constraints. Because Nova is the primary service used in managing the life cycle of virtual machines, it was decided that Nova logs should be used. We also decided to build models by merging logs from separate components, and servers, due to this producing a larger training set and is simpler to deal with, rather than having different models for each Nova component. Merging logs from different servers may give worse performance as some of the anomalies will only be present in one server at a time, however, we decided that this was not a major concern, and that it is a realistic scenario for the models to learn the behaviour of several servers. The exact logs used were Nova-scheduler, Nova-api, and Nova-compute. In contrast to the DeepLog experiments which only used INFO level logs [19], all logs were used in our models.

6.2.2 Baseline

In order to detect errors and benchmark the models, a baseline needs to be established. When there are no incoming requests from the user, very few logs are produced unless there is a large problem in the system. Therefore we created a script that would continuously send requests to the lab, leading to the production of enough logs for the models to be reliably trained.

The script consisted of a cycle of creating one virtual machine, waiting for a specified time, thereafter deleting the machine and finally waiting another specified time before once again creating a new virtual machine. There is a certain trade off when choosing the length of these cycles, if the time between starting virtual machines is too large, it will take a very long time to collect enough data, and when using unsupervised methods, there is a risk of the requests from the script such as creating a virtual machine will be classified as an anomaly. On the other hand if the time in between creating virtual machines is too small, errors that would only give minor symptoms with realistic usage may be amplified, for example a small delay causing the delete request for a virtual machine to be sent before the creation process is finished. This may lead to unrealistic scenarios, where the performance of the models in the lab is not reflected in production. In general the usage of the cloud can vary, so creating a realistic scenario can be difficult.

Figure 6.1 shows the frequency of the logs collected from the lab environment for a baseline cycle of creating and deleting a VM instance. What can be seen is that in between creating and deleting, when there are no user commands, there are almost no logs produced, which differs from the historical data. The cause is probably that our lab environment can be described as an ideal system, meaning without any user commands the cloud system is static and gives no logs. Since there are no log messages in between VM creations and deletions the only thing that can be analysed is said processes. Importantly, this results in that the time spans chosen in between creation and deletion, if large enough, will not influence the end result due to the fact that the models only see logs from these processes, and adding waiting time does not influence the input to the models. To have enough time to create a VM before deletion, both for baseline and during anomalies, a three minute wait time in between creating and deleting and one minute before creating a new one was decided upon. Figure 6.1 is from this configuration of cycle times.

The reason for choosing a simple cycle of create and delete commands is that it creates a lot of interaction between the different components of the system, as described previously in the flow of virtual machine creation. Other tasks that are possible include stopping and starting a virtual machine, which would produce fewer logs, and mostly involve Nova. Scenarios using start and stop, or other similar tasks could be useful for testing the models in different environments, however, the general performance of anomaly detection should be captured by the simple loop of creation and deletion.

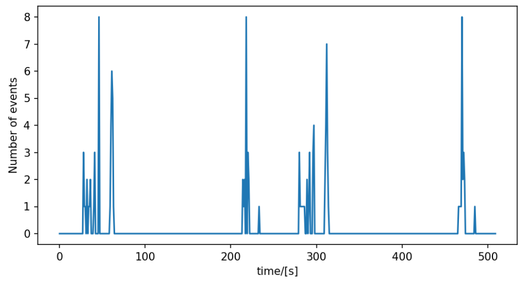


Figure 6.1: Number of events, i.e. the number of log messages, collected during a one second time window for the OpenStack on OpenStack system.

6.2.3 Injecting Errors

Using the OpenStack on OpenStack lab environment, several experiments were carried out. The experiments lasted for about one day each, and all except the *RabbitMQ network delay* were carried out on a single control node.

Control node shutdown A node shutting down is a simple anomaly, and was therefore chosen as an initial experiment. As services were deployed as Docker containers, in order to shutdown a service the `docker stop` command was used [18]. For instance to stop Nova-scheduler, the following command was issued:

```
docker stop nova_scheduler
```

Similarly, to simulate the shutdown of an entire server, all services were stopped.

Full disk A server may fail when the disk is filled, and may also experience performance degradation as it becomes close to being full. To simulate a disk being filled, a very large file was allocated. This was done on both the disk partitioned for the user and additionally for the disk partitioned for the Docker containers where the OpenStack services ran. The experiments will henceforth be known as *Full Docker disk*, and *Full user disk*. The `fallocate` command was used to fill the disks, for instance [22]:

```
fallocate -l 50GB
```

Maximum CPU Load A strained server may have a very high CPU usage, and while it is acceptable for a server to run at a high load for short amounts of time in production, a prolonged period of high load should be considered an anomaly also in the production cloud. Furthermore, while not always a symptom of a failure to the system, it is interesting to investigate whether the models detect the change in the system. To simulate a very high CPU load, the `stress-ng` program was used, using the following command.

```
stress-ng --cpu 4 --timeout 24h
```

This command specifies that the stress-ng program should spawn processes on 4 CPUs, which is the number of CPUs the virtual servers have, and maximize the load for 24 hours [60].

Network delay A throttled network is a clear source of performance degradation in the cloud, and may even be a sign of an external attack [19].

To simulate a network anomaly the Linux `tc-netem` command was used [64]. This command has many options, such as packet loss or data corruption. However due to the TCP layer using retransmission if data is lost or corrupted, most of the alternatives will manifest as a delay, thus we only tested a delay. The configurations attempted were a simple constant 100ms delay due to time constraints. The length was chosen as a realistic delay that would cause problems, and small enough that it should be subtle enough that the models are needed. The exact command used was:

```
sudo tc qdisc add dev eth0 root netem delay 100ms
```

In addition to this, an anomaly which has often been encountered on the production cloud at the Ericsson Research cloud are problems with the messaging system RabbitMQ. Therefore an experiment with a delay on the server running this container was also carried out, to simulate the message queues filling up. The delay experiments carried out were a *Control node network delay*, and *RabbitMQ network delay*.

In total, six kinds of experiments were carried out.

6.3 Cleaning and Parsing Data

The raw logs could not be parsed directly, but rather needed to be cleaned first. In order to parse the logs they need to have a consistent format, which could be used to identify the timestamp, or request ID, needed for feature extraction. Although there were deviations for a few of the components, the OpenStack logs used in this thesis had the following format: `<Date> <Time> <Pid> <Level> <Component> <RequestId>] <Content>`, where `Content` is the actual log message being analyzed, `Pid` is an ID for the process that sent the log, and `component` specifies which component of the OpenStack service that sent the log message.

There were several ways in which a single line of the log file did not conform to this format. For instance, there are log messages that are interrupted, and a new log message is started on the same line. In some cases a line break in a log message is started on a completely new line without the time, or other metadata, and may be cut off on the new line as well. In this thesis lines that do not conform to the general structure of the logs are removed.

Two methods were used to clean lines where a single log had been split up into multiple logs, i.e. a line break in the log message in the case that the time and other metadata was included in the following line. This often happened for example when a traceback had occurred, which is when something went wrong with the python code causing an exception, and a report of which functions were called to get to the point where the exception occurred is logged. The first method was simply to remove lines that did not include a request ID, which is useful for session windows, as logs that did not include a request ID could not be placed in a session window, so removing them already at this stage made the training process faster. The second method developed is to look at the metadata of the log, and if they are all equal to the previous log message, the message content is appended to the previous line. Due to the asynchronous nature of logs this could potentially cause problems, however this is checked by inspecting the parsed templates.

To parse the logs into templates, the drain method is used. For the historical data, the parameters chosen were simply the default: `maxdepth=4`, `similarity threshold=0.4`, `max number of children=100`. The reason they were not tuned was because it is difficult to automatically evaluate the content of several different log files. An option to search for parameters automatically is to include these parameters in the tuning of the full anomaly detection system. However, as there was no quantitative analysis done on the performance of anomaly detection on historical data, this could not be done. For the TripleO environment, a few different values of the similarity threshold were tested.

6.4 Historical Data

To begin with, historical log dumps were used. In order to analyze them, a pipeline was built merging all necessary steps. First the logs were cleaned, and parsed. Following that, feature extraction was implemented, and the output was used to train several models. Finally, the models were used for inference on previously unseen data.

First, the logs from the Nova-conductor component of a control node were clustered in order to study different types of normalization, and the qualitative attributes of the data. The data set is unlabeled and therefore we implemented various unsupervised algorithms, including Isolation Forest and Autoencoder [38, 24].

The following experiments in this qualitative investigation were done on a set of logs containing the services Nova-scheduler, Nova-conductor, and Nova-proxy, as these logs had several months of overlap. Due to the logs only storing a fixed amount of lines before overwriting previous logs, many of the logs from the historical dumps did not overlap. Some of the logs spanning several years, and others spanning only a few days. This made it difficult to model using several of them, and compare results.

As previously stated, the data set does not contain any labeled data and manual labeling is too time consuming. Without a labeled testing data set, a parameter tuning of the unsupervised models is not possible and therefore we cannot create any optimized models. Instead, we test the model on the training data and investigate if the results are reasonable. This is not the final result, but more a preparation for later stages and therefore an optimized model is not necessary at this stage.

The models are tested both for sliding windows, and session windows using request IDs, and the results are compared, both with different window sizes, and the differences between session window and sliding window.

6.5 TripleO

6.5.1 Data set

Our log data set comes from the two OpenStack on OpenStack lab environments. The logs are collected from the OpenStack services Nova-api, Nova-scheduler and Nova-compute from the combined five control and compute nodes, see Table 6.1. In total the data sets have 496,346 log messages where we collected logs from each experiment, see Section 6.2.3, once per lab environment. The data set is completely labeled since we manually inject the errors into the system and thus it can be used for evaluating the trained model. Table 6.2 shows the number of log messages collected for the baseline and outlier set for each lab, since these were collected separately. Furthermore, the number of parsed log keys per set is shown in the table. The log set for lab1 does not include the *Full Docker disk* or *RabbitMQ network delay* experiments. The reason is that the system almost broke down completely when executing these experiments. Moreover, the problem led to that the baseline data set for lab1 is significantly smaller than for the other lab2, see Table 6.2. In addition to that, the lab environments are not perfectly identical, hence it was decided to only use the data sets collected from lab2.

Table 6.2: The data sets collected from the two TripleO systems.

Lab	Log set	#Logs	#Log keys
1	Baseline	237853	79
	Outlier	109200	89
2	Baseline	68796	70
	Outlier	80188	75

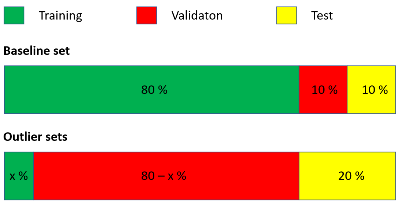


Figure 6.2: The split ratio of the data sets where *x* is the percentage of the outlier set needed for the contamination of the training set.

6.5.2 Data set split

To get an unbiased evaluation of the model, the data sets are split into three parts. We decided upon an 80% training, 10% validation and 10% testing split for the baseline data. How to divide the outlier data when using event vectors is more complicated. For a real cloud system, the logs collected will in most cases mainly consist of normal points with a small amount of outliers. To simulate this for the models using event vectors, we inject some of the outlier data into the training set which consists of the baseline data. This ratio of injected outliers is a parameter used in the grid search later on. The number of injected points needed in the training set will be distributed equally for all the experiment data sets. We hold 20% of each experiment data for testing and the rest 80% for training and validation. The outlier training data varies in the grid search and therefore the training and validation ratio. The two data splits are shown in Figure 6.2. For the models using an LSTM we did not contaminate the data set. This is because it is difficult to contaminate the training set in a realistic way, as this would involve inserting data points into a time series. This is in contrast to the models using event vectors, where one can simply insert an anomalous event vector into the training set.

6.5.3 Performance evaluation

Evaluating a classification model for finding outliers is not trivial. It is a balance between the hit rate of finding anomalies and the frequency of false alarms, i.e. miss classification of normal data. Commonly used measures are a combination of the metrics recall, precision and F-score, see Section 5.5.5. However these performance metrics are not optimally suited for our data. Once again using TP (true positive), and FP (false positive), the precision score is defined as TP/(TP+FP). Since TP is the number of correctly classified points in the outlier set, the TP value is completely independent of the baseline data. Respectively only the baseline data set affects the value of FP. Since our data set is split by baseline and outliers the precision score will be greatly dependant on our choice of validation set ratio of outliers to baseline. For instance by using a baseline data set unproportionally small in regards to the outliers, even if all baseline points are misclassified as outliers, TP+FP is dominated by the TP term, and TP/(TP+FP) will be close to 1, which is the optimal value for precision.

Instead we decided to use a ROC graph where the True positive rate is plotted against the False positive rate, an example previously shown in Figure 5.7 and explained in Section 5.5.5. It illustrates the balance between finding outliers and reducing the number of false alarms. Additionally, the result is independent of the size ratio of the baseline set and the outlier set, since both axes values are only dependant of their respective data set. One drawback is that there is not just a single value to compare the models with. However, one method of determining an optimal model is to set an upper limit of the percentage of false alarms. Thereafter the optimal model is the one which gives the maximum true positives. The upper limit is mostly dependant on the use case of the classification. For example, for detecting anomalies in critical systems determining when a failure occurs is of more importance than the false alarm rate.

6.5.4 Defining anomalies

Deciding how to measure the performance of the methods is not a simple task in the sense that a decision has to be made on how to label the collected data. Even though faults were present in the whole data set, not every log message is necessarily an anomaly, and if using sufficiently short sliding windows, not every window will be an anomaly either. Because the cycle used to send requests in the experiments is three minutes long, it is reasonable that windows with similar sizes or larger can be labeled as anomalous. Since the event vector methods do not use much shorter windows than this, we decided that all windows in the outlier sets are labeled as outliers, and all windows in the baseline data set are labeled as a normal point.

There is a further difficulty for the models that classify each log message as an anomaly or not. In this case one has to decide how to measure the performance of the model. Once again windows must be used due to the previously stated fact that each log message in the outlier sets will not be an anomaly. Further, it is difficult to manually go through the logs and label which are anomalous. In [19], the authors compare models with a resolution of a session defined by an instance ID. They classified a session as anomalous if the model classifies at least one log message within the window as an anomaly.

We decided to use the simpler method of sliding windows, as many of the logs we used were not part of a session defined by an instance ID. The alternative is to use session windows defined by the request ID which is used for the historical data, however, it is not certain that a whole session defined by a request ID is an anomaly if it belongs to an outlier set. We used the same metric of a window being anomalous if there is at least one log message classified as an anomaly within it. Once again, because of the length of the cycles, we decided to use windows with a size of three minutes and a step size of one minute. However, an initial investigation into the effects of window size and window step size using the LSTM parameter anomaly detection model was performed. By making this investigation we wish to ensure that the differences between the measurements with different window parameters were not very large, and conclusions could be made using the performance calculated with these windows.

6.6 Anomaly detection using Event Vectors

After having parsed the log messages into a sequence of log events one feature extraction method was to create event vectors, see Section 3.2. Using the obtained event vectors, a number of different conventional unsupervised machine learning algorithms were trained to find anomalies. To be able to use our models on real cloud systems, only unsupervised models were chosen since most collected log sets from industrial cloud environments are too large to label entirely.

Of the three types of event vectors, explained in Section 3.2, only sliding window was tested. To begin with it was not realistically possible to use fixed window on our collected data because of its collection duration. For instance a fixed window of 30 minutes results in only 48 event vectors per experiment since the data sets are collected over 24 hours. As for session window, which tracks individual requests through the OpenStack work flow, the way OpenStack handles request ID made it problematic. The issue occurs when a request crosses between two OpenStack services. At the receiver service a new request ID is generated, hence tracking a request between services becomes exceedingly difficult [39]. Therefore we decided not to continue with the session window.

6.6.1 Grid search

A grid search was performed for the hyperparameter optimization of the machine learning models. The hyperparameter values are set before the training phase and have a large impact on the model learning and thereby the result. A model has several types of hyperparameters that can be set, for example for KNN the number of neighbors and the distance metric. By testing multiple hyperparameter configurations the aim is to find the optimal model. The result of a specific hyperparameter combination was then plotted in the ROC graph whereafter the resulting ROC graph was analysed. In addition to the machine learning model hyperparameters, parameter values which determined the input event vector were included in the grid search.

6.6.1.1 Input There are several parameter types which impact the input event vector. First it is the ones which effect the feature extraction from the log sequence, e.g., the window size. Secondly it is the types determining the additional operations on the extracted event vectors before being used as input to the normalization method.

Window size and step size
As mentioned before, only sliding windows with *window size* = 0.5, 4, 10, 30, 60 and 120 minutes were tested. Exclusive to *window size* = 0.5min is the problem where all the data sets consists of a considerable number of event vectors which are zero vectors, since between creation and deletion of VM the system mostly does not generate any logs. As a result, the outlier data sets contain a sizable ratio of baseline data which would worsen the performance when validating the model. As such every zero vector in all data sets were removed beforehand. Then the step size was set to one minute for every window except for the 0.5 where a 50% overlap was used instead. For the baseline data set one minute step size resulted in approximately 16 000 event vectors and for each outlier set around 1300. The number is not exact since depending on the size of a window there will be between 10 and 100 additional vectors. The overlap of the vectors using one minute steps is high for most of the window sizes but required because of the small number of event vectors. For instance the result of a step size of two minutes is 750 event vectors and three minutes gives 430. These numbers, we think, are too few to get an unbiased result.

Number of log keys
Parsing the baseline data set gave a total of 79 log keys, i.e. 79 number of log type templates while further parsing of the outlier set gave a total of 89 log keys. This means 10 new types of log messages were found outside the baseline logs. Intuitively we can classify every log message belonging to the new log keys as outliers since they are previously unseen. This in turn keeps the event vector dimension to 79. An alternative that also was tested is to use all the 89 log keys, expanding the event vectors to 89 dimensions and letting the model find these outliers and possibly more. However, this option will not be possible using streamed data since expanding the dimension requires a retrained model. A third method used was to reduce by half, the number of log keys to the 40 most common log types in the baseline set and treat every other log key as anomalies. The false positive rate will as a result increase, but on our data set only approximately 0.1% of the baseline log messages were outside the 40 most common log keys. Additionally, an advantage

is the large dimension reduction, which not only can give better result but will also shorten the time required to train and run the model.

Feature scaling
Four different normalization methods, in addition to not normalizing, were tested in the grid search: IDF, see Section 3.2, Min-max normalization, Z-score normalization and Robust scaling. They are all defined in Section 5.4.1.

Training set contamination
To resemble a real cloud system's log data, which inherently will contain anomalies, outliers were injected into our training set consisting of baseline data. An additional reason is that some algorithms need to have outliers during the learning phase, for example the Isolation Forest algorithm works by isolating outliers in the training set. Interestingly, unique to the project, is the said possibility to vary the ratio of anomalies in the used data set. In a log dump from a real cloud system the ratio is already determined. For example, in the BGL data set used in [28] approximately 7.3% of all log messages were anomalies. Therefore the performance dependency of the training set anomaly rate is of interest. The ratio of injected outliers was used as a parameter in the grid search. 0, 1, 5, 10 and 20% outliers of the training set were the values used.

6.6.1.2 Model hyperparameters

All models were implemented in Python using PyOD [54], an outlier detection toolkit built on top of Scikit-learn. Except for the varied hyperparameters, all values were set to their respective default values as in the PyOD documentation [54]. The chosen machine learning algorithms were Isolation Forest, KNN, LOF and One Class SVM. The four algorithms cover three types of detection algorithms. KNN and LOF are proximity based, Isolation Forest uses outlier ensembles, and, finally, OCSVM is a linear model. The two tested OCSVM kernels were rbf, which is the default, and the linear kernel, since it performed well in [28]. In addition to the euclidean, the Manhattan distance was set as the metric for KNN and LOF. The POP algorithm in [26] used the Manhattan distance when comparing two event vectors to be able to weigh every log key equally. That is, when adding events to one of the event vectors, the distance will change equally independently of which log key the event has. Overall, the model hyperparameters for event vectors were not the main focus in the thesis. The result of the input parameters, such as window size and number of log keys used are more interesting in the context of the automatic outlier detection pipeline. Finally, the total parameter search space in the grid search is presented in Table 6.3.

6.7 Online methods

The online anomaly detection models tested were *log key anomaly detection with an LSTM*, and *Parameter value anomaly detection using an LSTM*. These were tested and analyzed separately, unlike what the authors of DeepLog did. In that work the *Log key anomaly detection* method first classifies a log, and if it classifies it as normal the *Parameter value anomaly detection* method is used [19].

Table 6.3: The parameter space in the grid search when using event vectors.

Window step	1 min
Window size	0.5, 4, 10, 30, 60, 120 min
Norm types	None, IDF, Min-Max, Z-norm, Robust
#Log keys	40, 79, 80
Training set contamination	0, 0.005, 0.01, 0.08, 0.1, 0.12, 0.2
Model hyperparameter contamination	0.005, 0.01, 0.08, 0.1, 0.12, 0.2
Isolation Forest	Max features = 1, 2, 3, #Estimators = 90, 100, 110, 150, 200
OCSVM	Kernal = linear, rbf
KNN	#Neighbors = 3, 5, 7, 20 Distance metric = Euclidean, Manhattan
LOF	#Neighbors = 10, 17, 20, 25, 40 Distance metric = Euclidean, Manhattan

For both methods, a grid search was performed, varying neural network structure, and model parameters. This was done on a GPU cluster of several GeForce GTX 1080 Ti GPUs. The hopsworks program was used to submit training jobs as notebooks [29]. To create the neural network, Keras and Tensorflow were used [33, 65]. The values used in the grid search were chosen close to the parameters used in DeepLog.

6.7.1 Log key anomaly detection using an LSTM

6.7.1.1 Input After cleaning and parsing the logs, they are separated into windows of h logs. This is equivalent to a sliding window, however, instead of time, it is the number of logs that determines the width, and the step size is a single log message.

6.7.1.2 Grid search The default neural network structure was two LSTM layers with 64 memory units each, and finally a dense layer with a softmax activation function outputting the probability distribution. The default window size was $h = 50$ logs. A grid search was carried out varying the structure of the neural network. Both one and two LSTM layers were tested, each with 64, 128, or 264, units. The input window was also tested with values 5, 15, 50, and 150 in the grid search. Finally the g parameter was varied in the grid search between 2 and 15. The number of epochs was not included in the grid search, but was chosen by observing the loss, and categorical accuracy for a few different arbitrary parameters. Training was stopped when the validation loss was decreasing at a low rate, or the validation accuracy stopped increasing.

The LSTM is trained to classify what message follows the input window of h logs. In order to train the model, an output vector is created, which is the time series of log events shifted h steps. Furthermore, in order to use the neural network in a classification setting, each output is represented using one-hot encoding. This is where event k is represented by a vector which has the value 0 in all components, except k, where it has the value 1. To detect anomalies, the deep log strategy was used, defining a message to be anomalous if it is not in the g most probable events.

6.7.1.3 Variations to the method A few variations to the methods used in DeepLog were attempted in the training of the model in order to investigate possible improvements to the technique. First of all, one possible drawback of using an LSTM is overfitting, as described in the machine learning chapter. Therefore dropout was tested, dropping the inputs to the second LSTM layer, at rates 0.25, 0.5, 0.75. Furthermore, the data is very unbalanced, which could again lead to overfitting, where the result of the neural network does not represent the actual probability distribution, but tends to be overconfident in a single event. To attempt to alleviate this, a weighting of the loss was introduced using Keras class weights feature. An IDF style weighting was attempted giving each class a weight $m\log(N/n_i)+1$, for different factors of m, where N is the max number of logs for any class in the training data, and n_i is the number of logs with class i. The reason different factors are tested is that if the weights are too large, the resulting function will no longer be representative of the actual probability distribution. However, if they are too small it will not make a difference compared to the unweighted model.

6.7.2 Parameter value anomaly detection using an LSTM

6.7.2.1 Input After the cleaning and parsing step, we added a post-processing step, which searches for numerical values in the dynamic parts of the logs and extracts them to be used for anomaly detection. Other types of dynamic parts, such as IP addresses could be used for anomaly detection, although due to the difficulty of predicting the values of these parameters we have restricted the use to only numerical values. Each event gets a set of metrics, and each one can be used in univariate time series anomaly detection, or together, as multivariate time series anomaly detection. All log events could be used for this purpose if the time is treated as a dynamic parameter, which would likely have a good performance in scenarios with delays. However, this was not done in this investigation due to the fact that training a single model takes a long time, thus a grid search would not be possible using all of the log keys. The time is used as a dynamic parameter for all of the log keys that were used.

Firstly, the data had to be filtered further. Many of the log events happened very rarely, which made it difficult to train models using the parameters. Therefore, any log key which appeared fewer than 250 times was filtered out, and a model was not built for it. However, as LSTM networks often need a lot of data to be able to predict more complex behavior, the limit may be too small. The next step was to form windows from the parameter time series in the same way as for the *Log key anomaly detection using an LSTM*. However, in this case an additional step of normalizing the data with Z-score normalization was done. The data was normalized as some of the dynamic parameters were of very different sizes, and the training would diverge if the data was not normalized. Z-score normalization was chosen because it was simple to implement, and the normalization had to be implemented from scratch due to the GPU servers not having packages installed. The robust norm as described in Section 5.4.1 could also have been used, but was not attempted in this experiment.

The models used here differed from the ones in DeepLog in the sense that we used separate models for each specific parameter in the log sequence. For example a log message having the parameter vector $[p1, p2]$ has one model which predicts $p1$, and one model that predicts $p2$, using past values of both $p1$ and $p2$ as inputs. Each model has a separate classification threshold for each parameter, whereas DeepLog has the same input, but just a single model that predicts both and detects an anomaly using the mean squared error of the components. This offers more flexibility in tuning the models, and the possibility to decide for which parameters one should ensemble or remove the models. The drawback is that there are more models, taking longer to train, and that the performance may be worse. The performance effect of using these separate models is investigated by training and validating the models on the collected data from OpenStack on OpenStack. In this case a unique LSTM was used for each parameter, however the same classification method can be carried out with a single LSTM, by simply not taking the mean squared error, and instead classifying each parameter separately.

6.7.2.2 Grid search The default neural network structure was two LSTM layers with 128 memory units each, a dense layer with 64 neurons with a relu activation function, and finally an output neuron. The default window was a window of $h = 50$ logs. A grid search was carried out varying the structure of the neural network. Both one and two LSTM layers were tested, each with 128, or 264, units. The input window, h, was also tested with values 5, 15, 50, and 150 in the grid search, and the dense layer was tested with values 4, 20, 64 neurons. The classification threshold was tested with values 2.5, 3.0, 3.5, 3.7, 3.9, 4.3 where this value is a multiple of the calculated standard deviation of the prediction error.

6.8 Ensemble models

In previous literature the aim has been on improving state of the art methods by inventing new algorithms to detect anomalies. One possible improvement is instead to create a system where several different models are used together to determine if an anomaly has occurred. To investigate if this is possible, ensemble models are tested by implementing a voting system where a window is classified as an outlier only if several models classify it as an outlier.

To study if there is a possibility that ensemble models improve performance, several models are chosen from the investigation into event vector models. Two models each from LOF, KNN, OCSVM, and Isolation Forest were used in the investigation. As the number of combinations is 2^n, with n models, only a few can be tested at a time, which is why only 8 models were tested. The two models chosen from each category were the highest performing models, given a maximum false positive rate. The investigation was first done using models with a maximum of 25 % false positive rate, and subsequently using models with a 10 % maximum. If the two top performing models had the exact same true positive and false positive rate, one lesser performing model was chosen, due to the high probability that they have the same rates by classifying in a very similar way, and therefore will not be effective in an ensemble. For the 8 chosen models, every subset of two or more models is tested. For a given ensemble of models, every voting threshold is tested, from a window being classified as an outlier if two models vote that it is an outlier, to all models needing to vote that it is an outlier. The previously detailed investigation was done on both 60 minute windows and 10 minute windows. The outlier set used to evaluate the combinations was the *Maximum CPU load* set, due to its relatively good, but not perfect, performance. The online methods are not used, and only event vector models with the same window size and step are used at the same time, due to the difficulty of designing a system for the models with different windows to vote.

6.9 Inference

When the previously described pipeline is used in practice, the data will be streaming, and not performed in batches on logs that have previously been collected as is done in the training stage. Therefore a separate step was added to the pipeline, in which data is simulated to be streaming and the models may detect anomalies in an online fashion. In this thesis, this step of the pipeline is used to test the previously trained models on unseen data. The simulation of streaming data is done by looping over log files, and in each step of the pipeline using python generators to create streams of log messages and feature vectors.

The first step is to clean incoming lines. Since part of the cleaning process was appending lines to previous lines, it is first when a line appears that should not be appended to a previous message, or removed completely, that the previous message is parsed by Drain. The state of the Drain tree from training has been saved to a file. The parsed log message is then streamed to the feature extraction part of the pipeline. During the clean and Drain process, the log files are also merged by comparing times from the messages in all files, and streaming the latest line to the feature extraction.

For sliding windows, extracting features with streaming data works exactly as it would for offline data. Log messages are collected, and added to the feature vector when a log message is received in which the time stamp has passed the end of the window. The feature vector is then streamed to the models, in order to infer if the window was an anomaly or not.

However, when the data is streaming, one has to decide how to define the session windows. Since it is difficult to know what event corresponds to a request ending, it is difficult to know when to return a feature vector to use for inference as there might be additional messages belonging to the window in the future. Therefore, a variable cutoff time was added, such that when the time since the first message in the window arrived has exceeded the cutoff time, the feature vector is used by the models. In order to determine the cutoff time, the distribution of time spans for session windows was studied, and a suitable cutoff was chosen, such that false positives are not detected due to evaluating a session window too early, and feedback on whether or not a request is anomalous is produced as quickly as possible.

Another issue when testing the anomaly detection models is what to do when an unknown log message appears. If the Drain algorithm creates a new log group for this message, the anomaly detection models can not use it, as they are trained using a fixed number of dimensions. Due to the large amounts of data collected for training, if a new log message is detected it will be marked as an outlier.

Finally, the feature vector is scaled using the previously saved weights, and used as input into the previously trained anomaly detection models.

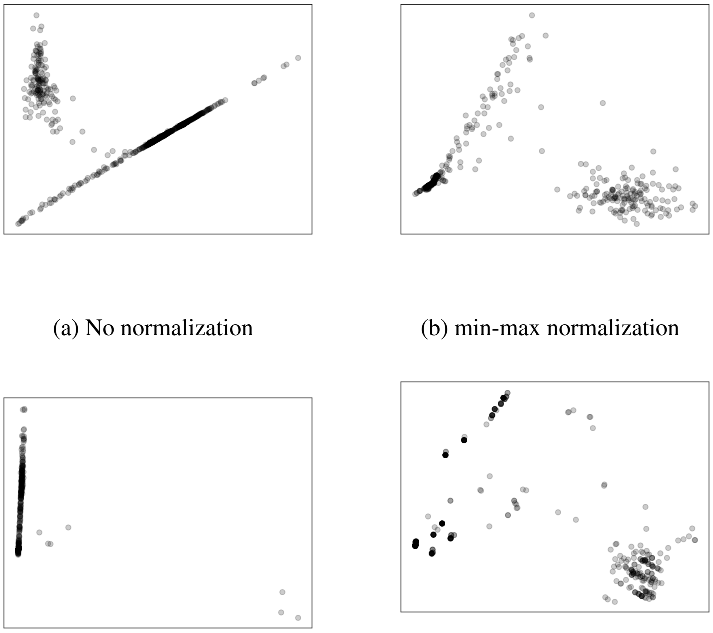
7

Results

In this chapter the results of the thesis will be presented, and discussed. The chapter is divided into several parts detailing the various investigations that were carried out. The results from the historical log dump are presented in Section 7.1. Afterwards all the results will be from the data collected from the TripleO lab environment. In Section 7.2 general results regarding the data from the TripleO experiments are presented, before the results from the different anomaly detection methods are shown. These results are divided into five sections, with the models using event vectors presented first in Section 7.3. The investigation into the performance measurement of online methods using sliding windows is exhibited in Section 7.4, before the two methods from DeepLog are presented in Sections 7.5 and 7.6. Finally, the investigation into ensemble models is presented in Section 7.7.

7.1 Historical data

To illustrate the qualitative aspects of the data, an initial clustering was done. The chosen window was a sliding window with a size of 1 hour and a 20 minute step size. The resulting event count matrix had the dimensions 12568x104 which means 12568 *feature vectors* with 104 unique *event IDs*. To get a rough overview of the feature matrix, we reduced its dimensions to 2 using PCA and then scattered them as illustrated in Figure 7.1. Firstly, there is clearly structure in the data, suggesting that it is well suited for clustering and anomaly detection. Secondly, the figure shows the effect of weighting the input data. The unweighted data shows two main clusters. Using the min-max norm, shown in Figure 7.1b, these clusters remain, but appear to be somewhat less dense, which might not be good for anomaly detection as it is less clear if a point does not belong to a cluster. Z-score normalization is often used because the outliers remain separated, which can be seen in Figure 7.1c, with a few of the points very far from a main cluster. Finally, IDF weighting can be seen to produce more variation in the data, with several clusters, as unimportant features that dominate using other normalization methods are removed. These ob-



(a) No normalization

(b) min-max normalization

(c) Z-normalization

(d) IDF weighting

Figure 7.1: Log event vectors from a sliding window after PCA for different normalization methods.

servations indicate that Z-norm, and IDF weighting likely are advantageous to use when detecting anomalies.

7.1.1 Session Window

Before evaluation the session windows, a suitable cutoff time, described in Section 6.9, has to be determined. In Figure 7.2, a histogram is shown of the time from the first to the last log message with a given request ID for the log data set used for training in this experiment. It can clearly be seen that most of the requests are less than one second in length. 90% of requests are executed within one minute, and 98% within one hour. Therefore, for the following investigations, a cutoff time of one hour was used. In production this would likely be lower, as it is likely one would want the results faster.

Several of the requests that took a significant amount of time were also the ones identified by the models as being anomalous. A very simple anomaly detection model could therefore be simply to label a request as anomalous if it takes longer than a specified time.

The results on the test data using Isolation Forest are seen in Figure 7.4. This is almost identical to the result for the training data, namely that it finds one anomaly when a messaging database server is unavailable. The requests that are labeled as outliers remain in the system for a very long time, and contain several log statements similar to the following:

```
AMQP server on 10.129.8.18:5672 is unreachable: timed out.
Trying again in 1 seconds.: timeout: timed out
```

Figure 7.3 shows the results with the same test data using the autoencoder, with ReLu activation, and hidden layers with the following architecture: 500-300-2-300-500. The figure shows that the reconstructed points are very close to the original points, in other words it can encode the data well. The two requests with highest reconstruction error correspond to an anomaly where there were unknown hosts in the system. Messages such as the following can be seen in these requests:

```
Received an update from an unknown host
'eselda11u01s12.xerces.lan'. Re-created its InstanceList.
```

A request with the unresponsive AMQP server, found by the Isolation Forest model, is also seen as the third largest reconstruction error. The finding that Isolation Forest and the autoencoder both detect anomalies, but produce slightly different results indicates that using the models in an ensemble could lead to improved performance.

These anomalies could also be found simply by looking at the number of log messages over time in the training data. Figure 7.5 shows the number of events during a one hour period. There are clearly three large spikes in the rate of log messages being produced. The final spike contains the anomaly with the unreachable AMQP server, and the other two contain the unknown host anomaly. The fact that the anomalies are characterized by such a large increase in request time and number of log messages suggests that the models may be better trained using data without these anomalies. This could result in better detection of subtler anomalies, such as a performance degradation, and indicates that the TripleO data will be needed to compare the different models.

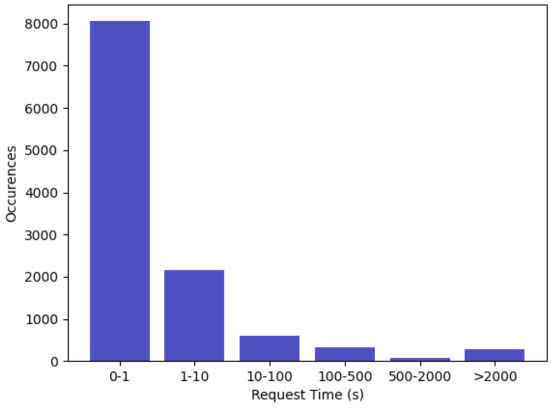


Figure 7.2: A histogram of the time a single request ID was active within a subset of OpenStack Nova logs.

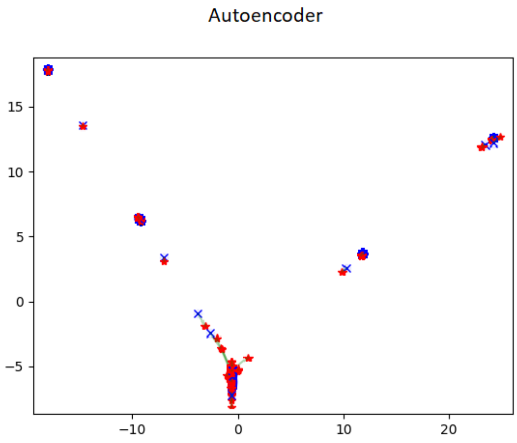


Figure 7.3: Event vectors from test log data after reducing to two dimensions with PCA, and weighting with IDF. Red points are reconstructions, and blue points are original data.

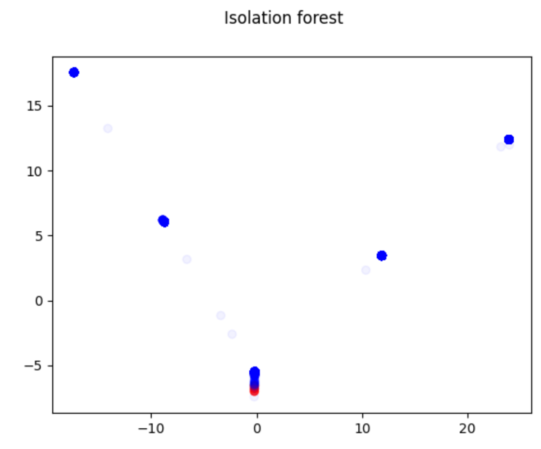


Figure 7.4: Isolation Forest labels after classifying test data. Data has been reduced to two dimensions with PCA, and weighted with IDF. Red points are outliers, and blue points are inliers.

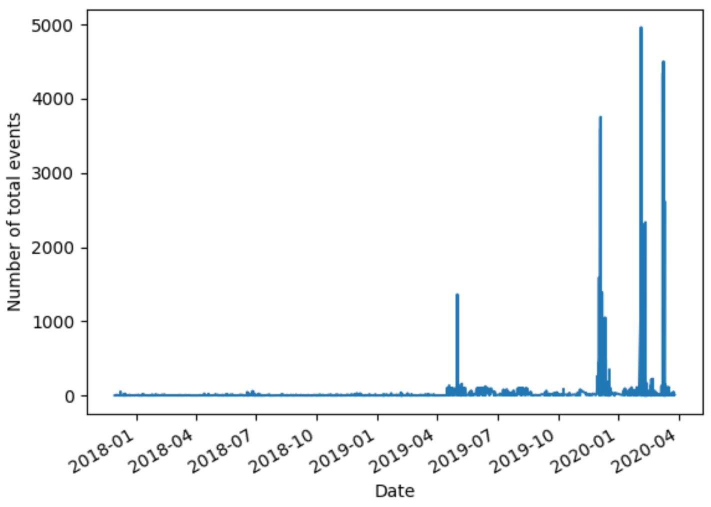


Figure 7.5: Total number of OpenStack log events during a window of one hour.

7.1.2 Sliding Window

A sliding window was also tested on the same data. Using a window of 6 hours with a step of 1 hour the first difference to the session window is that there are a lot more logs used by the models. This is because of the fact that the session window does not use any logs without a request ID. A further consequence of this fact is that all log events found in the test data using session windows were also present in the training data. Using a sliding window, however, there were 3 messages that were not present in the training data, and directly classified as outliers. These three messages consisted of two different Python Tracebacks, and it is unknown if they caused actual failures in the system, however it is possible that they are false positives, and that the exceptions were handled properly. It may also be an error in cleaning the data that caused the messages to be different than in the training data, with the assumption of the metadata from a log message staying constant after a line break failing. In addition to the unseen messages, the Isolation Forest model finds anomalies on the final peak in Figure 7.5. With a 1 hour long window, with a 12 minute step, anomalies from all three peaks are found by the Isolation Forest model, meaning that it is the best result compared to the longer sliding window, and session window. This indicates that the results found in [28], such as the longer windows giving better results, may not be applicable to this data set.

An observation that can be made from Figure 7.4, is that the outliers are not easily identified from the two-dimensional plot, which is consistent with results from previous work stating that PCA did not work very well for finding anomalies.

A difficulty with using the historical data is the fact that the logs do not completely overlap in time. The log files have a maximum number of messages that can be stored, after which old log messages are deleted and replaced by new messages. This means that some logs span over the course of several years, whereas some only span across several days. To get optimal models, more time may have to be spent in order to train on data that spans a longer period of time for all logs.

7.2 TripleO data

7.2.1 Clean and Drain

After the first cleaning and parsing of the data collected from the OpenStack on OpenStack the first observation was that the parsing was not perfect, producing several different templates from what should have been the same event, for example several messages where the instance ID was not detected as a variable part:

```
[instance: a71004ce-425b-46af-b36e-b6957b7c6bea]
```

To remedy this, preprocessing was added to mask IDs, hashes, and IP addresses, with the regular expression patterns shown in Table 7.1. The next observation made

Table 7.1: The regular expression patterns used to preprocess OpenStack logs before parsing using the Drain algorithm.

Mask	Regular expression
Id	[\\da-f]{8}\\-[\\da-f]{4}\\-[\\da-f]{4}\\-[\\da-f]{4}\\-[\\da-f]{12}
Hash	\\/[\\da-f]{40}
IP address	\\d{1,3}\\.\\d{1,3}\\.\\d{1,3}\\.\\d{1,3}

was that changing the parameters for the Drain algorithm drastically changed the templates mined. For instance, with the `similarity threshold=0.4`, the top message was:

```
"<*>,<*> <*> <*> HTTP/1.1"" status: <*> len: <*> time: <*>"
```

whereas for 0.6, the message was split into two,

```
"<*>,<*> ""GET <*> HTTP/1.1"" status: <*> len: <*> time: <*>"
```

and

```
"<*>,<*> ""POST <*> HTTP/1.1"" status: <*> len: <*> time: <*>"
```

However, when testing a simple anomaly detection using the *Log key anomaly detection* from the DeepLog architecture, the difference, illustrated by the F-score in Table 7.2 was not large. The parameters for the model were not optimized here, however it is unlikely the effect will change for different parameters, therefore the `similarity threshold` likely does not have a large impact on the final anomaly detection. On the other hand when using the templates for parameter anomaly detection it is better to split up the messages. This is exemplified by the case above as the dynamic parameters `len`, and `time`, represent different time series when produced from a `POST` request and a `GET` request, and one therefore might get more accurate results by splitting them up. Thus, we chose to use 0.6 in all following results.

Table 7.2: The F-score for *Log key anomaly detection using an LSTM* when parsing with drain using different values of the parameter `similarity threshold`.

`similarity threshold`	F-score
0.4	0.880
0.5	0.881
0.6	0.881
0.7	0.877

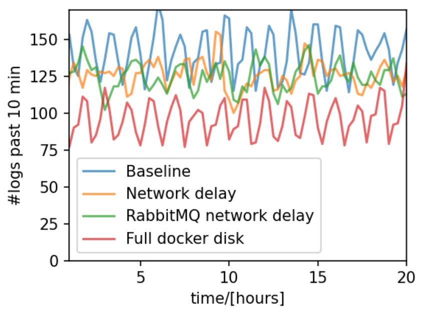
7.2.2 Log frequency

Figure 7.6 shows the total number of log messages generated in the OpenStack on OpenStack environment during 10 and 30 minute windows for the baseline and the different outlier data sets. On average the number of logs produced each minute are between 10 and 15, which can be considered a low rate considering that the parsing gave 79 event IDs. This may imply that a significantly longer window than one minute is required to collect enough information to distinguish outliers from normal points. Three data sets stand out in Figure 7.6b. With a collection duration of 30 minutes the experiments consisting of a *Control node network delay*, a *Full Docker disk*, and a *RabbitMQ network delay* stand out by having a much lower number of total logs generated than the rest. The difference is enough to almost perfectly separate the above mentioned experiments with the baseline by setting a given boundary around 400 logs. This in turn suggests more complex models are not required to find the three anomalies when using windows larger than 30 minutes, but can be a good performance reference. The lower frequency is most likely the request queue build-up when having a network delay which slows down the entire workflow and gives fewer logs. Furthermore, when filling the Docker disk, the OpenStack services running in the given container do not have much space to write and save log messages and therefore the logging frequency will be reduced. Decreasing the length of the window to 10 minutes, in Figure 7.6a, the same method as before can still be used for detecting the experiment with a *Full Docker disk*, however the two network delay anomalies can no longer be separated using an upper limit, which is the same as for the rest of the anomalies for tested window sizes.

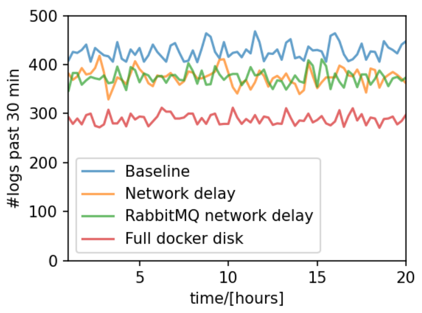
Finding 1. A fixed boundary of the total number of logs generated during 30 minutes or more is enough to detect the two different network delays and the *Full Docker disk* anomaly.

7.3 Event vectors

Looking further than the log frequency, the grid search using event vectors resulted in roughly 150 000 runs per anomaly type, where each iteration had a specific parameter configuration, for example a *window size* = 30 minutes using KNN. Isolation Forest was much faster than the other algorithms, around four times faster than KNN and LOF and as much as 30 times faster than OCSVM on the local computer. The result of each run is given as a true and false positive score, i.e., the accuracy in the outlier set and the inaccuracy in the baseline set respectively. These were plotted in the ROC graph as points for each outlier validation set, seen in Figure 7.7a for the *Maximum CPU load* experiment. To find correlations the points were colored according to desired parameters, for instance, in Figure 7.7a the points are differentiated by their window size.



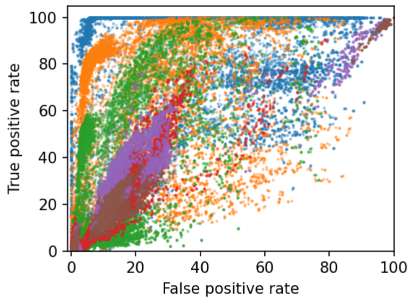
(a) Window size = 10 minutes



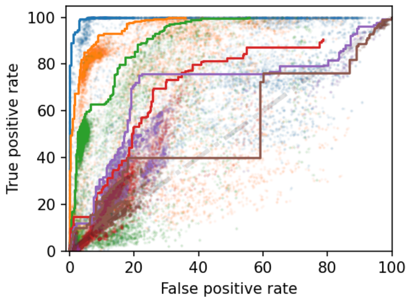
(b) Window size = 30 minutes

Figure 7.6: Number of log messages generated in the OpenStack on OpenStack system during 10 and 30 minute windows over 20 hours.

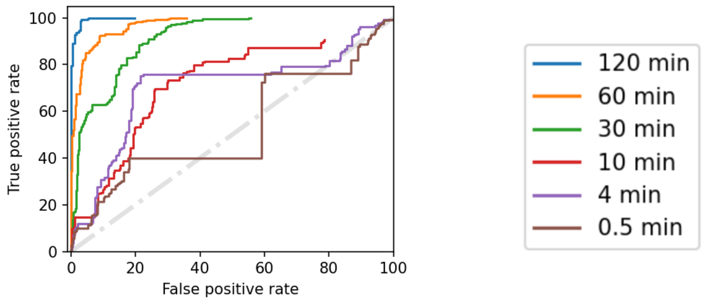
The ROC graph was also represented by a ROC curve, Figure 7.7c, for easier viewing by only plotting increasing true positive score points for each window size, i.e., only the best runs are shown for different false positive rates. In addition to the curve, a grey line was added where the true and false positive rates are equal, which shows the result of a randomized classifier. In several figures including Figure 7.8 the figures only include values for small false positive rates, in this case from 0 to 10, as for a real system one is not interested in models that have a very large false positive rate.



(a) ROC graph



(b) ROC curve in ROC graph



(c) ROC curve

Figure 7.7: The transformation of the ROC graph results to a ROC curve with the different window sizes colored. The figures shows the true and false positive rate scored from the grid search tested on the *Maximum CPU load* anomaly data set. The plots are color mapped according to the window size used to create the event vectors and shows an increase in performance with larger windows. The gray line represents the score of a 50/50 randomized classifier.

7.3.1 Window size

The performance of the event vector models is highly dependant on the window size, which is in accordance to the previous finding with log frequency. The dependency is shown in the respective ROC curves for every anomaly plotted with the window size as the color parameter in Figure 7.8. The overall observation is that longer windows give better results. The only exception were the anomaly sets where no model performed well, but in the average over all test sets, shown in Figure 7.8g the performance correlates with the window size. The better model performance could be a result of the fact that more information is collected when using longer time window, for instance with a *window size* = 0.5min, each event vector will not be able to cover both a VM creation and deletion when using our script.

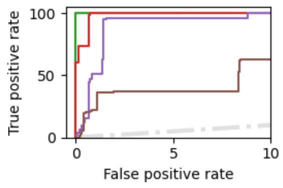
Moreover, also seen in Figure 7.8 is the large variation in performance depending on the anomaly type.

Finding 2. Larger window size achieves better results when using event vectors.

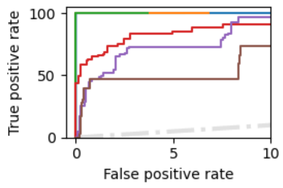
The *Full Docker disk*, *Control node network delay* and *RabbitMQ network delay* experiments, shown in figures 7.8a, 7.8c, 7.8b, were the anomalies the models performed best on. As the event vectors include information about the frequency of log events, this is expected from Finding 1. For the *Full Docker disk* and *RabbitMQ network delay*, a *window size* = 120 min, 60 min and 30 min gave almost a perfect score, which is on par with the Finding 1 result, but additionally performed well even with a *window size* = 10 min. An example of one of the best model scores on the *RabbitMQ network delay* outlier set is a model using a *window size* = 30 min, scoring a true positive rate = 99.9% and false positive rate = 0.05%.

The results for the *Maximum CPU load* experiment, shown in Figure 7.8d, gave the greatest difference in performance with regards to the window size. In accordance to the previous finding, the performance increased with a longer time window and as an example, two well performing models using *window size* = 120 min and 60 min gave true positive/false positive rates at approximately 80%/0.1% and 84%/4%, respectively. The score not being perfect and its large dependence on window size makes this anomaly one of the more interesting anomaly types to analyse.

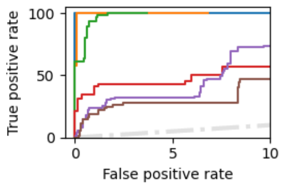
The *Control node shutdown* and *Full user disk* anomalies shown in Figures 7.8f, 7.8e gave noticeably worse results. Two scores using a *window size* = 60 min and 30 min are a true positive/false positive rate at around 30%/5% and 10%/0.02% respectively, which is considerably higher than a random classifier which guesses 50/50 if a point is an anomaly or not. However, the smaller window sizes mostly followed the random classifier line where the true positive is equal to the false. Remarkably, unique for these anomalies, the *window size* = 120 min gave a worse result than when 60 min and 30 min. There are some possible reasons why the models gave lower performance in comparison to the other anomalies. Firstly, the OpenStack services were running in a Docker container which had a separate memory partition. Filling the user disk could thus have a low impact of the workflow of the VM creation and deletion. Secondly, the worse scores may be due to the fact that there are two control nodes, and the TripleO system has a redundancy implemented such that the redundant node takes over.



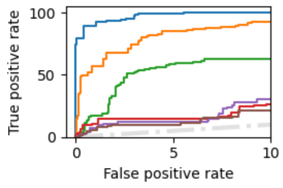
(a) Full Docker disk



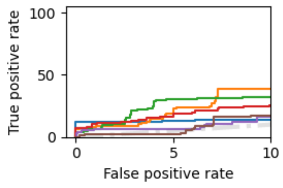
(b) RabbitMQ network delay



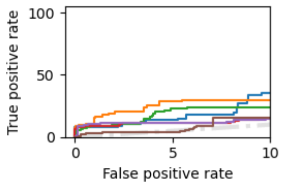
(c) Control node network delay



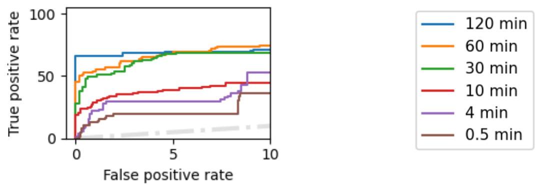
(d) Maximum CPU load



(e) Full user disk



(f) Control node shutdown



(g) Average

Figure 7.8: The true and false positive scores for each anomaly log set where the color is mapped according to the window size used. In figures (a), (b) and (c) the *window size* = 120 min and 60 min overlap and are not visible along the axis border. A difference in performance can be seen between the anomaly types where (a), (b) and (c) performed substantially better than (e) and (f) which for the low window size almost follows the random classifier's diagonal grey line.

Finding 3. The detection score for a *Full user disk* and the *Control node shutdown* were substantially lower than the rest, independent of window size.

7.3.2 Machine learning models

The average score when using the same model on all the outlier sets is plotted in Figure 7.8g. However, by using different models depending on the anomaly type, a higher average performance can be achieved. Additional benefits is the flexibility of deciding exactly the upper limit of the false positive rate for every different type of anomaly. A scenario might be if a large CPU load is more crucial to detect than a network delay, meaning a greater average false positive rate needs to be accepted. In this case, as seen in Figure 7.9, an Isolation Forest model might be used, as the *Maximum CPU load* anomaly using a *window size* = 30 min has a true/false positive rate of 50%/5%. On the other hand to reduce false alarms a KNN model could be used with a performance of 90%/0.6% on the *Control node network delay* even though approximately 100%/1.7% is possible.

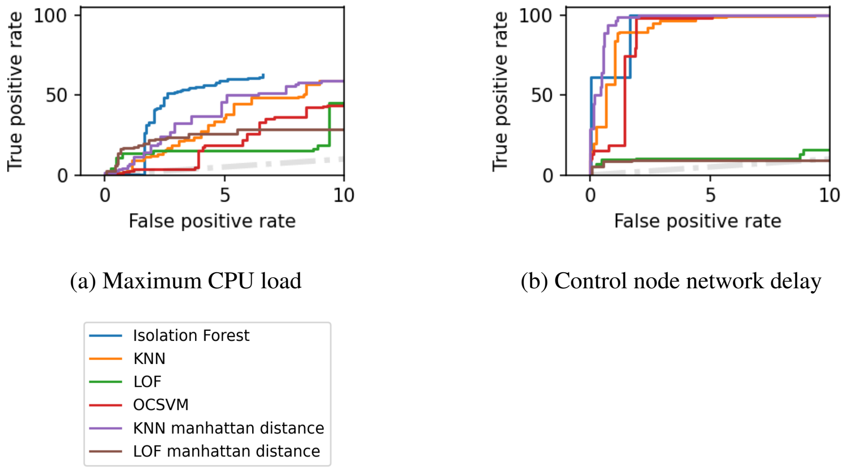


Figure 7.9: The relative performance among the tested machine learning algorithms for the *Maximum CPU load* experiment, (a) and a *Control node network delay*, (b) using a *window size* = 30 min. There was no algorithm which consistently performed best for all the anomaly types. The highest score for most false positive rates were Isolation Forest in (a) and KNN using the Manhattan distance in (b).

Even though the relative performance of the different machine learning methods depends on the anomaly type, there was an overall correlation. The models which performed well on one of the outlier data sets in most cases also scored high on the rest of them. Therefore the average performance using the same model on all types of anomalies can somewhat be representative of using different models depending on anomaly type. On the other hand the window size largely affected the relative performance of the machine learning models when compared to each other. The machine learning models performance on the *Maximum CPU load* experiment using *window size* = 120 min, 60 min and 30 min are plotted in Figures 7.10a, 7.10b, 7.9a respectively. For a *window size* = 60 min, LOF and LOF with Manhattan metric distance could be considered the best, especially with false positive rates between 0% and 3%. However, increasing to *window size* = 120 minutes, then KNN and Isolation Forest perform considerably better than the rest. Consequently, for best result the final model will depend on the window size chosen.

Finding 4. The relative performance between the machine learning algorithms varied across the different window sizes but did not to a large extent depend on the anomaly types.

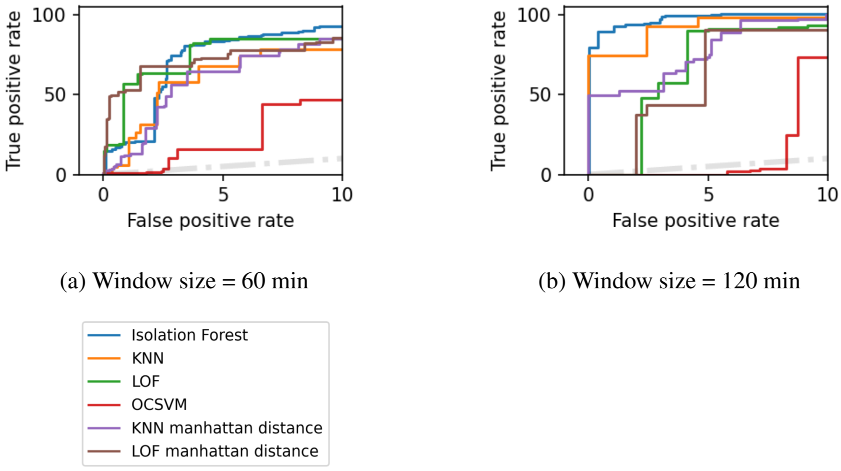
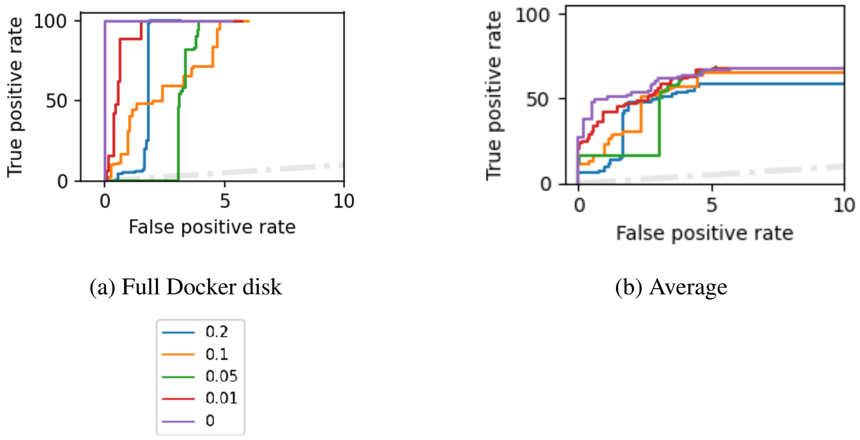


Figure 7.10: The performance for each machine learning algorithm on the *Maximum CPU load* anomaly for *window size* = 120 min and 60 min. A difference in relative scores can be seen between the window sizes. With *window size* = 60min the two LOF performed well whereas Isolation forest was best when increasing the window to 120min.

7.3.3 Training set contamination ratio

One of the more interesting parameters in the grid search is the ratio of outliers injected into the training set consisting of baseline data. Unique to our Thesis is the possibility to vary the contamination ratio since we use the OpenStack on OpenStack system. On the other hand, in available log data sets from real systems the number of anomalies is already set. For instance in [28] they used two data sets: BGL and HDFS with a 6.5% and 0.15% contamination ratio respectively and 2.2% combined.

The outlier ratio used was varied between 20% and 0% and made a significant difference in the performance of the models. For instance in Figure 7.11a the lowest false positive rate achieved with a true positive score of 100% ranged between 0% to 4.8% depending on the contamination ratio trained on. The average over the contamination ratio with a *window size* = 30 min is shown in Figure 7.11b and represents overall the scores for the different widths and anomaly sets. The consistently best score was achieved when the ratio was 0, meaning no outliers in the training set.



(a) Full Docker disk

(b) Average

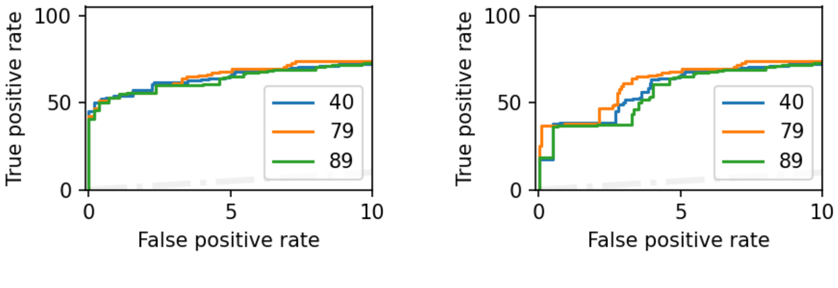
Figure 7.11: The model performance when varying the training set contamination which is colored in the plot, with *window size* = 30 min. Visible is a great change in model score whereas a lower contamination increases the performance, especially for the *Full Docker disk* anomaly in (a).

Furthermore, excluding the 0.2 ratio, the score and ratio had an inverse relationship as the performance increased when the ratio decreased. The score of the considerably high ratio of 0.2 varied greatly. For a few cases it performed as well as the 0, 0.01 pair but also occasionally was among the worst. However, for the majority of anomalies and widths, 0.2 gave a result in between the pairs of 0, 0.01 and 0.05, 0.1 ratio. As mentioned before, the zero ratio gave the best results but it is not very realistic to not have any anomalies in a data dump from a real system. Likewise in comparison to the BGL and HDFS data sets, the training set with a 0.2 ratio was much more contaminated. The disproportional high number of outliers in the training set could explain the variance of the the performance using the unsupervised learning algorithms with a contamination ratio of 0.2.

Finding 5. The model performance increased when the outlier ratio in the training set decreased as the optimal performance was reached with no contamination.

7.3.4 Log keys

Changing the number of log keys, i.e., the number of parsed templates did not consistently impact the score as much as the previous parameters. Using the 79 log keys parsed from the baseline set resulted mostly in the highest score by a small margin where with 89 keys, the combined templates parsed from both the outlier and baseline sets, performed the worst. In between came the score with 40 keys, half of the baseline templates. The significantly greatest difference was when using Isolation Forest, the average with *window size* = 60 min is in Figure 7.12b, whereas a false positive rate = 3% gave a true positive = 65%/55%/40% for the log keys 79/40/89 respectively. However, overall the average using all the models, plotted in Figure 7.12a for *window size* = 60 min only gave a maximum difference of approximately 2%-points.



(a) Total average (b) Average using Isolation Forest

Figure 7.12: The relative model scores when using 40, 79 and 89 log keys when *window size* = 60 min. It is observed that the number of log keys has a lower impact in comparison to window size and contamination ratio. 79 templates over performed the rest for the majority of cases. The larges variance were when using Isolation Forest in (b).

The low impact is probably because of the small number of log messages that belong to the templates outside the 40 most frequent templates. These infrequent log templates correspond approximately to only 0.1% and 1% of the log messages for the baseline and outlier sets, respectively. Remarkably, only one log message is outside the 40 for the *Control node shutdown* experiment, indicating all of the abnormal templates are generated from the control node that was shut down. Finally, an interesting result which was not measured is the model run time of the training and testing phase, since the number of log keys corresponds to the number of dimensions of the event vectors.

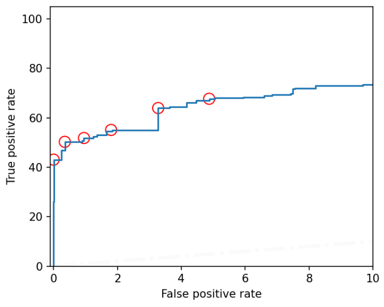


Figure 7.13: The average over all anomalies when *window size* = 60 min and a contamination ratio of 0.5%. The chosen presented scores in Table 7.3 are circled and represents the highest scores with various set upper boundaries.

7.3.5 Choosing model

As shown, the optimal detection model varies highly depending on both the window size and the contamination ratio. The window size represents the longest time duration until an anomaly is detected and can therefore not be too long. An additional trade off to the true positive rate is the false positive rate since it indicates the rate of false alarms, which cannot be too high. To get a final performance measure of a model we decided upon a *window size* = 60 min and contamination ratio of 0.5%. The test and validation score of the total average using the same model on all the anomaly types is shown in Table 7.3 and also circled in Figure 7.13.
We present multiple possible performance scores by different upper boundaries to the false positive rate, in this case 0.1, 0.5, 1, 2 and 5%. The false positive score on the baseline test set are for most of them zero, which is unusually low. In addition, the false positive rate is smaller than on the validation set, which is uncommon since the models are chosen according to the validation score. The cause of the relation might be how the data is split for training, validation and testing. The split is made in chronological order of log collection dates where the test set is the last 10%. This represents the last day of logging and the validation set is the second last day. The date order could therefore cause a high variance between the sets which is undesirable and could have been prevented by a randomised data split. Furthermore, the low number of event vectors in the test and validation set could give a biased score. With approximately 1600 event vectors in the validation set, a false positive rate = 0.35% is equal to only around 6 outliers.

Otherwise, the table shows a true positive validation score around the 50% mark. To choose a specific model to use, we think the model performing a false positive rate = 0.35% and true positive rate = 50.27% is the optimal. The chosen model is a KNN with 20 neighbors using the robust norm and 40 log keys. As previously set the *window size* = 60 min and a contamination ratio of 0.05%. A false positive rate = 0.35% gives one false alarm per 285 vectors and with a 33% window overlap each day will create 36 event vectors, hence one false alarm per approximately 8 days. However, since the VM creation and deletion frequency is manually set by us the number of days is not a good representation of the false positive rate.

Table 7.3: The highest validation scores when setting different upper boundaries on the false positive rate and their respective test scores.

	Validation score %		**Test score %**	
Max FP %	FP	TP	FP	TP
0.1	0	43.06	0	36.44
0.5	0.35	50.27	0	47.63
1	0.89	51.81	0	50.07
2	1.8	55.07	0	56.46
4	3.27	63.92	4.87	69.47
5	4.87	67.67	1.65	72.64

7.4 Performance measurement for online methods

Figure 7.14 shows the performance of *Parameter value anomaly detection using an LSTM* model. The different points correspond to different classification thresholds, and different window parameters using a single log event as an example, although other log events show similar results. It is clear from the image that a different length will affect the true positive rate and false positive rate. As expected, larger windows will have a larger chance of having a log message classified as an anomaly present in them, and therefore both the true positive rate, and the false positive rate increase. On the other hand, changing the window step size did not produce a different performance measurement.

However, one must also consider that longer window step sizes imply that there are fewer windows in total, and therefore the absolute number of false positives will be lower. In reality this is the metric that matters as it is the time spent on investigating false positives one wishes to minimize. As we wish to have fast feedback when the models are in use, and easily label the data, henceforth a *window size* = 3 min, and a step size of one minute will be used for the performance measurements. However, as these results show, this is not a perfect performance measure and care should be taken when comparing positive rates with models with different window parameters. To optimally evaluate the performance of the online methods each individual log needs to be labelled.

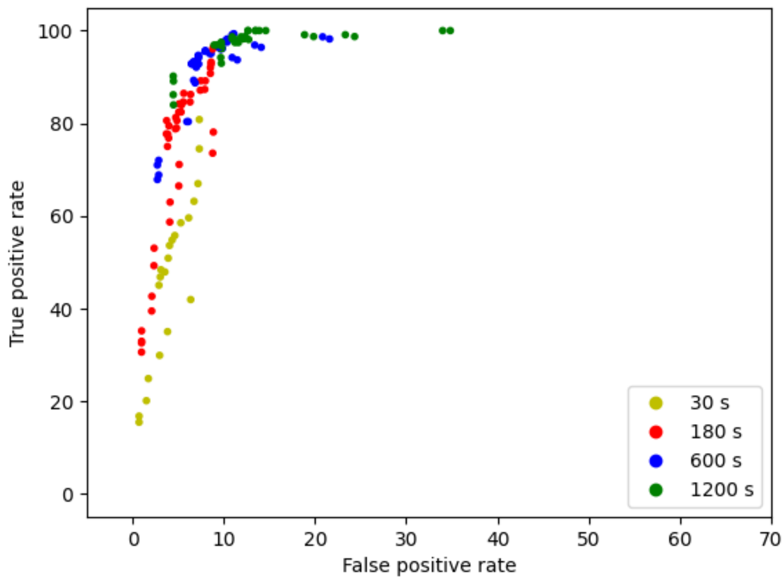


Figure 7.14: Performance of *Parameter value anomaly detection using an LSTM* for a single log event and the *Maximum CPU load* outlier set, with varying window sizes. The figure shows that increasing the window size increases the measured value of both the true positive and false positive rates.

7.5 Log Key anomaly detection using an LSTM

Figure 7.15 shows the results for the *Log key anomaly detection using an LSTM*, with different number of memory units in the first LSTM layer. For many of the data sets, the model does not significantly exceed the performance of a random classifier. For three of the data sets the model gets decent performance. In Figure 7.15e the results for the *Full Docker disk* are shown, illustrating a near perfect classifier with a true positive rate near 100%, and a false positive rate near 0%.

Similarly for the delay data sets in Figures 7.15a, 7.15b the model manages to classify a large portion of the windows in the anomaly sets as anomalies for a relatively smaller false positive rate. However, considering the results in the previous sections, showing far better performance, overall this anomaly detection system is not very useful for these data sets. One reason it does not perform well could be that the experiments in question manifest mostly as delays in the system, whereas the model builds on the assumption that it is the order of logs, or which log messages that appear, that will change in the system. One possible improvement in the future could be to add a feature to the model including the relative times of the log messages. However, more research may be needed to investigate in what scenarios the DeepLog method of modelling a probability distribution for the next log message works well.

Finding 6. The *Log key anomaly detection using an LSTM* performs worse on our data sets than the event vector models.

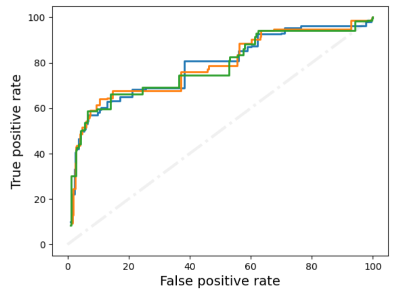
Figure 7.15 also shows the impact of a different number of memory units in the first LSTM layer, suggesting that increasing the capacity of the neural network does not have a significant effect on anomaly detection. Similarly changing the number of LSTM units in the second layer, or removing the second layer completely, does not have an effect on the performance. These results are very similar to that of the original DeepLog paper, in which the authors also find that changing the structure of the neural network does not highly effect the classification performance [19]. This is a positive result, as it means the model can likely be implemented for different systems without a lot of parameter tuning.

In Figure 7.17 one can see the effect of changing the number of logs in the input window for the model. Although there is not a large difference between the different window sizes in general, in Figure 7.17e it appears the model is not able to capture the full behavior of the system using only the last 5 logs, and hence the classification performance is much worse for the *Full Docker disk* experiment. Although there is some improvement in using a window of 50 instead of 15, it is not possible to make the conclusion that it is better in general. Finally, it can be noted that the g parameter tunes the rate of outliers, increasing g leads to points closer to the lower left corner in the ROC curves, whereas decreasing g leads to points closer to the top right corner.

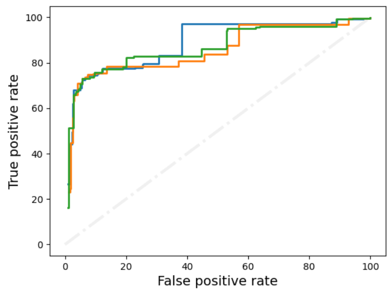
Finding 7. *Log key anomaly detection using an LSTM* is not sensitive to hyperparameter changes.

7.5.1 Variations to the DeepLog method

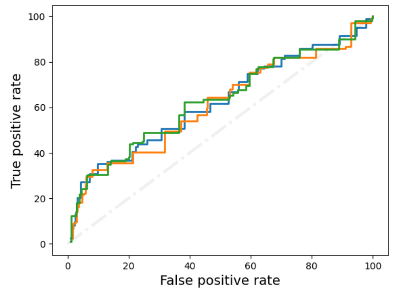
Two variations to the method were attempted in order to improve the result. The results shown below are for the *Maximum CPU load* data set, but the results were similarly negative for the rest of the data sets.
In Figure 7.16, the results for the dropout rate for the input to the second layer. There is no significant difference between using dropout, and not using dropout, the results for high dropout are slightly worse. It is still possible that the model is overfitting, as it achieves a very high classification accuracy, in other words the log message with the highest calculated probability by the model is the one that appears, however due to the concurrency of the system, it is likely that this should



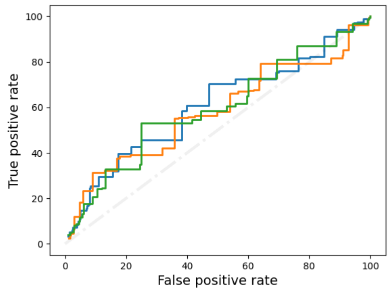
(a) Control node network delay



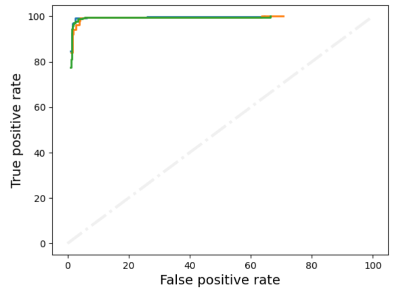
(b) RabbitMQ network delay



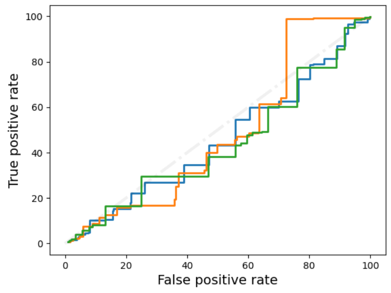
(c) Maximum CPU load



(d) Full user disk



(e) Full Docker disk



(f) Control node shutdown

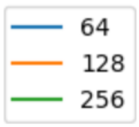


Figure 7.15: The performance of *Log key anomaly detection using an LSTM*, for different number of memory units in the first LSTM layer. It illustrates the small difference when changing hyperparameters, and the poor classification performance for most test sets, as the performance is close to the random classifier illustrated by the gray line.

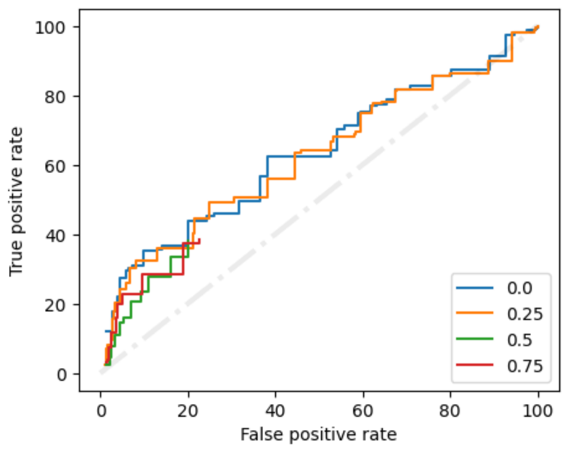
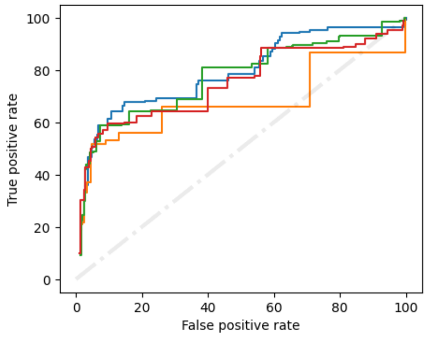


Figure 7.16: The performance of *Log key anomaly detection using an LSTM* on a *Maximum CPU load* data set, for varying dropout rates. The figure shows that there is little effect in adding dropout.

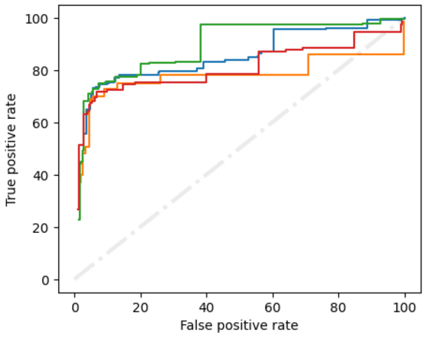
not be possible. One possible future attempt could be to add a recurrent dropout, which drops the recurrent connections, however, this was not possible due to the GPU implementation of the LSTM layer, so it was not tested.

The use of weighting the output was tested, and this can be seen in Figure 7.18. Once again, this attempt at improving the results did not show any significant difference, and using weights with a larger factor m did not improve performance to that of the original network, which can be seen in the figure as $m = 0$. Here, the performance data is collected only from the default neural network, and with fewer g values, which is why there are fewer data points than in the previous figures.

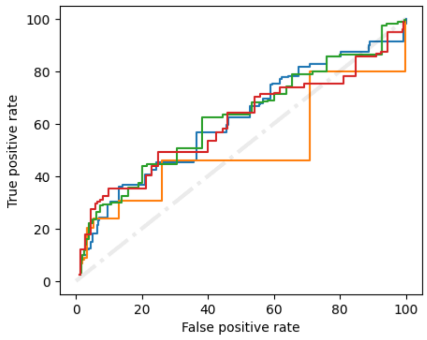
One final investigation was taken into the probability values for a given log event. Figure 7.19 shows a moving average for 500 log messages of the probability in the *Maximum CPU load* experiment, and the baseline validation set. The figure shows that the probabilities for log messages in the outlier set is on average slightly lower than that for log messages in the set of normal points. Therefore a possible improvement to the DeepLog architecture would be to create a model using the probability values output by the LSTM as inputs, instead of simply looking at the most probable events. However, more research would be needed to determine if this will improve performance.



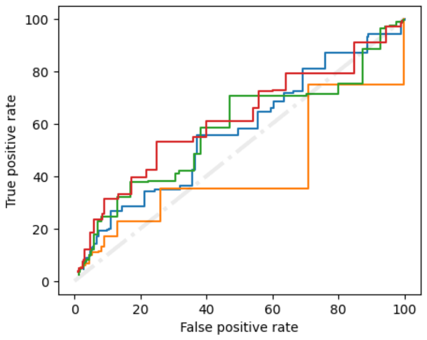
(a) Control node network delay



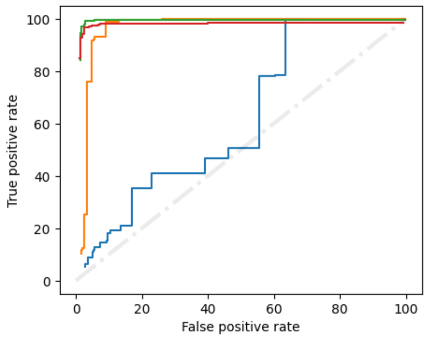
(b) RabbitMQ network delay



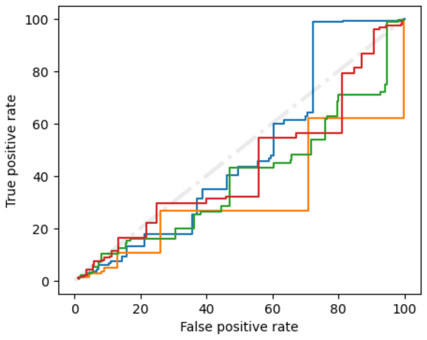
(c) Maximum CPU load



(d) Full user disk



(e) Full Docker disk



(f) Control node shutdown

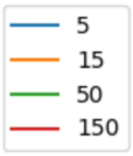


Figure 7.17: The performance of Log key anomaly detection using an LSTM, for different number of logs in the input window. Illustrated is the fact that a window of 5 logs performed significantly worse in the *Full Docker disk* experiment (e).

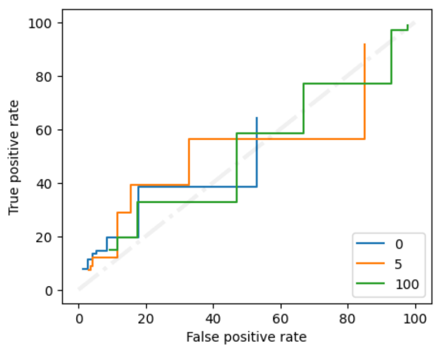


Figure 7.18: The performance of *Log key anomaly detection using an LSTM* on a *Maximum CPU load* data set, where less frequent log events are weighted higher, for varying weight factors. The figure shows that there is little effect in weighting less frequent log events.

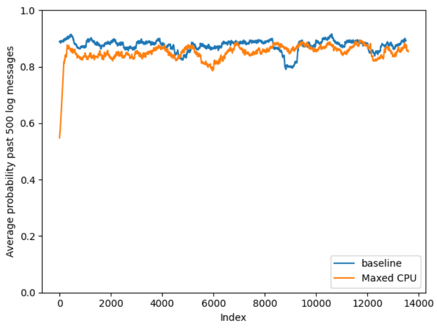


Figure 7.19: A 500 log message moving average of the probability that the received log message is the next log message given the previous log messages as calculated by an LSTM. The figure shows that on average the log events from the *Maximum CPU load* experiment were calculated to happen at a lower probability.

7.6 Parameter value anomaly detection using an LSTM

Of the 79 log templates in the training set, only 8 met the conditions to be used for *Parameter value anomaly detection using an LSTM*. The conditions, as described in Chapter 6, include that there were more than 250 occurrences of the log event in the training set, and that the dynamic parts of the log template consisted of a numerical value. These log events can be seen in Table 7.4, which shows that most of the log message parameters include some kind of time parameter, such as the time a request took, or how long operations on an instance took. This is likely a good thing, as performance degradation is often experienced as a slower system, and the fact that OpenStack has several messages logging the time, implies that it should be possible to detect slowdowns in the system.

Figure 7.20 shows the results for each test set, comparing the best result from each log event. In general Figure 7.20 shows that many of the log events can be used to accurately detect anomalies within three minutes for the delay experiments, the *Full Docker disk*, and the *Maximum CPU load* experiment. However, none of the models are able to get a better than random performance in the *Full user disk* and *Control node shutdown*, as opposed to the event vector models, which managed to get significantly better performance even for windows as short as 10 minutes.

Finding 8. *Parameter value anomaly detection using an LSTM* has nearly optimal performance for the *Control node network delay*, *RabbitMQ network delay*, *Full Docker disk*, and *Maximum CPU load* experiments, but does not outperform a random classifier in the *Control node shutdown*, and *Full user disk* experiments.

Figure 7.20a shows the different log events with the *Control node network delay* test set. This test set was one of the easiest to find for most of the models, yet the performance here is very poor for two of the log events having event IDs A0011 and A0012. For these two events it was the case that the loss for the LSTM did not significantly decrease which gave poor performance in all of the experiment sets. This could be for two reasons. Firstly, it is possible that the threshold of 250 events was too few for the LSTM to learn the behaviour of the parameters.

A second possibility is that the parameters are more stochastic in nature, and therefore will never be able to be predicted well. In this case these models should be removed from the anomaly detection system as they do not offer anything. This will also improve the training time of the models, as fewer models have to be trained and tuned. The figure further shows that some events are more sensitive to the classification threshold than others. For example in Figure 7.20c, the log event A0006 spans a larger portion of the graph, whereas A0018 only has a small curve, as all the classification thresholds tested ended up within that area. This shows that it is likely different events will have to use different classification thresholds. As some of the events have a quite large false positive rate as their lowest value in this search, in the future one should experiment with even larger thresholds.

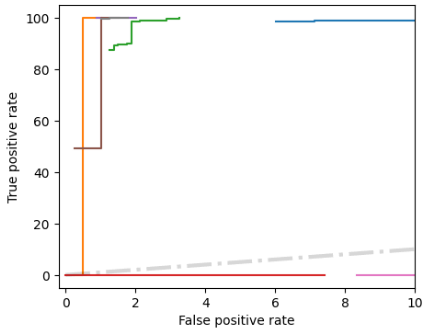
Finding 9. Not all log events are suited for *Parameter value anomaly detection using an LSTM*.

Table 7.4: The log events used in *Parameter value anomaly detection using an LSTM*.

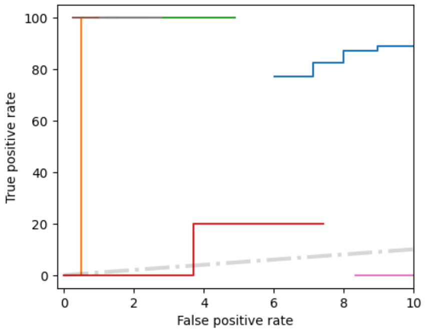
Event ID	Log template
A0001	<*>,<*>""GET <*>HTTP/1.1"" status: <*>len: <*>time: <*>
A0003	"DELETE /v2.1/servers/<*>HTTP/1.1" status: 204 len: <*>time: <*>
A0006	<*>,<*>""POST <*>HTTP/1.1"" status: <*>len: <*>time: <*>
A0011	Running instance usage audit for host <*>from <*><*>to <*><*><*>instances.
A0012	Function 'nova.servicegroup.drivers.db.DbDriver._report_state' run outlasted interval by <*>sec
A0018	[instance: <*>] Took <*>seconds to <*>the instance on the hypervisor.
A0019	[instance: <*>] Took <*>seconds to deallocate network for instance.
A0025	[instance: <*>] Took <*>seconds to build instance.

Another interesting observation that can be made when comparing the different experiments in Figure 7.20 is that a parameter has different performance for the different test sets. For instance the log key A0025 which has a dynamic parameter that performs well on the *RabbitMQ network delay* experiment, but does not perform well on the *Maximum CPU load* experiment. This is reasonable as A0025 is a log message specifying the time it took to build an instance from Nova-compute. The *Maximum CPU load* anomaly on a control node might not affect the performance on the compute node, whereas the *RabbitMQ network delay* will affect all of the servers, and specifically the compute nodes that run Nova-compute. This illustrates an advantage with the *Parameter value anomaly detection using an LSTM*, namely that one can get information on what is wrong, rather than a whole window being anomalous.

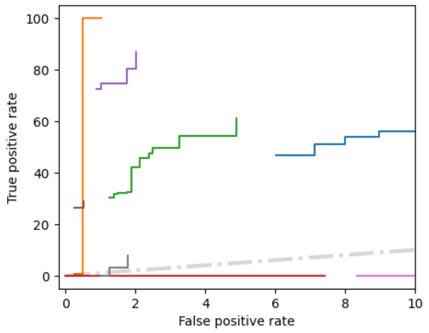
Figure 7.21 shows the performance of our LSTM with a unique models for each dynamic parameter, and the performance of the LSTM proposed by the DeepLog article taking the mean squared error from a single LSTM for each log event. The ROC curve is created by taking the best model for each false positive, including taking the log event with the best performance. It is clear that the performance is very similar, in fact for all experiments except the *Full user disk*, and the *Maximum CPU load*, the curves overlap almost entirely. Looking at the *Maximum CPU load* specifically, however, in Figure 7.21c one can see that there is a large improvement by using separate models. Not only does the single LSTM fail to reach the near perfect true positive rate, but it only reaches its maximum value at around 3% false positive rate, as opposed to the single LSTM reaching the maximum value at around a 1% false positive rate.



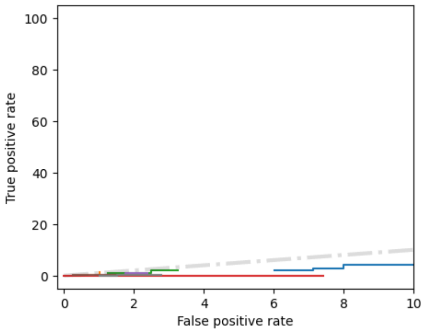
(a) Control node network delay



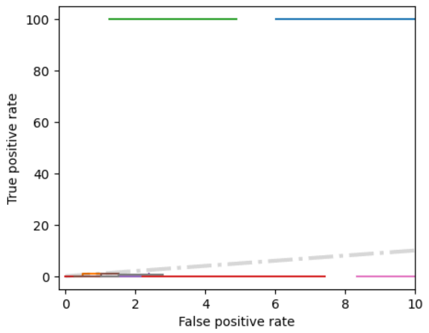
(b) RabbitMQ network delay



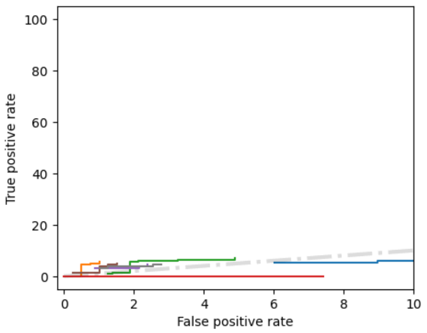
(c) Maximum CPU load



(d) Full user disk



(e) Full Docker disk



(f) Control node shutdown

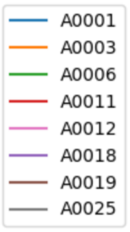
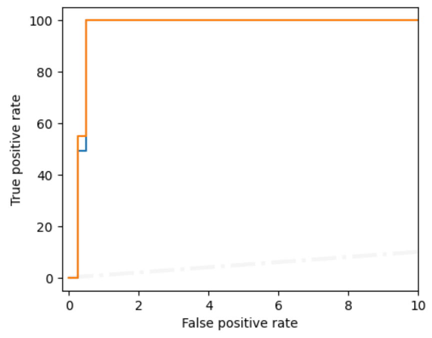
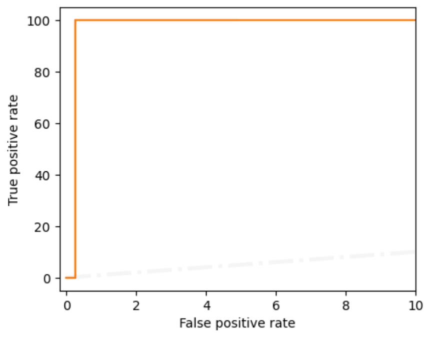


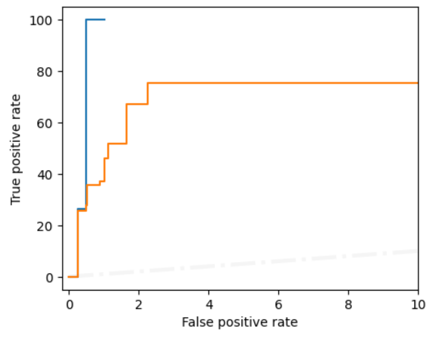
Figure 7.20: The performance of *Parameter value anomaly detection using an LSTM* for different log events. The best performing dynamic parameter is plotted for each event ID used. The figure shows that many different dynamic parameters can be used for anomaly detection using an LSTM.



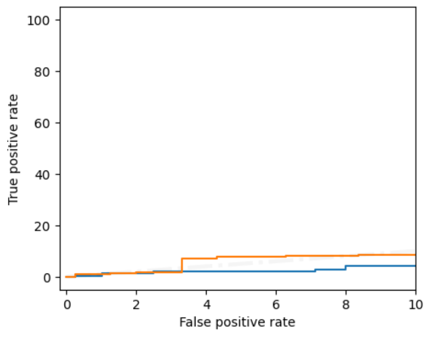
(a) Control node network delay



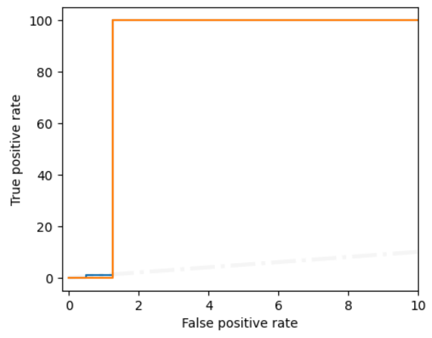
(b) RabbitMQ network delay



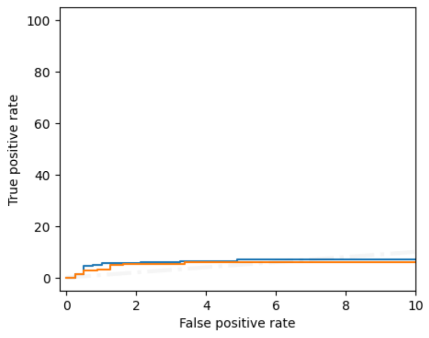
(c) Maximum CPU load



(d) Full user disk



(e) Full Docker disk



(f) Control node shutdown

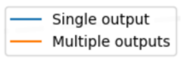


Figure 7.21: The performance of *Log key anomaly detection using an LSTM*, with one unique LSTM per dynamic parameter (single output), compared with one LSTM per log event taking the mean squared error of the result (multiple outputs). The figure shows that the performance is mostly the same, however for the *Maximum CPU load* experiment (c) there was a significant difference.

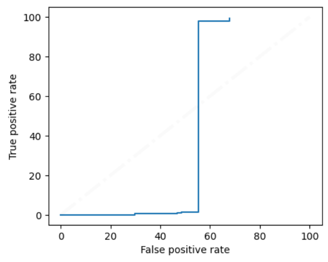


Figure 7.22: The performance for a single dynamic parameter of a log event on the *Control node shutdown* data set. The figure illustrates the strange behavior for this parameter, with a jump from almost 0 % to 100 % in positive accuracy.

The reason for the better performance could be that some of the features are impossible for the LSTM to learn, and therefore simply add noise to the output, reducing the precision of the prediction error, and thus reducing the accuracy of the classification. There were, however, individual log events where the DeepLog approach taking the mean squared error of the prediction error was significantly better. This is likely due to the ensemble effect, where all parameters were possible to predict, and therefore taking an average of the prediction error over all of them gave better performance. In order to get optimal performance, it is probably necessary to test different configurations, of which parameters one should take average, or have separate. It would also be possible to have a voting system between the different parameters, as another method to ensemble the models.

Finding 10. One can achieve a performance increase over DeepLog by having a unique LSTM for each dynamic parameter and a unique threshold for the prediction error, rather than taking a mean squared error over several dynamic parameters.

An interesting case is given by the *Control node shutdown* experiment using the log key A0001 and the second dynamic parameter, shown in Figure 7.22. It shows a classifier that achieves a high false positive rate, with a very low true positive rate, exemplifying a very poor classifier. However, the values imply that one can construct a good classifier by doing the opposite of what this classifier does. The classifier then jumps to a very high true positive rate. The reason for these strange results is in part due to the nature of the parameter, as looking at Table 7.4 it is seen that this particular parameter is the status code of a request. This parameter is therefore not a continuous variable, but has a few fixed values it can take, causing jumps in the classification when the threshold reaches these values. Furthermore, all of the data sets contain error codes, however the reason for the high false positive rate when comparing to the low true positive rate could be that there is actually an anomaly in the validation set for the baseline data. A possibility is that it contains fewer success codes or behaves differently than the training set, and that the problem, therefore, is in the labelling of the data.

This example shows that perhaps the models should be treated differently for non continuous variables, such as using classification instead of regression. It also exemplifies a need for each parameter to be treated separately. Firstly, due to the fact that one parameter may be more difficult for the model to predict, such as the status of a request, therefore adding noise when taking the mean squared error and reducing the performance. Secondly, due to the fact that one parameter may be anomalous, such as the status of a request, while another, such as the request time, may be normal. It should be noted in this second case, however, that if all parameters are used as inputs to the models other parameters that are not anomalous may still be classified as anomalies due to the fact that the model was trained for normal behaviour of all the input variables, rather than them being anomalies. Finally, it should be noted that the results from the grid search were the same as for the *Log key anomaly detection using an LSTM*, that is to say the model hyperparameters had very little effect on the performance of the models.

Finding 11. *Parameter value anomaly detection using an LSTM* is not sensitive to hyperparameter changes.

7.6.1 Evaluation

A final evaluation was done on the test set as the threshold parameter is chosen by using the standard deviation of the prediction error in the validation set. The conclusion is that the results hold, as the performance was very similar on the test set. Specifically one of the parameters for the event with event ID A0006 can be seen in Table 7.5. This was chosen as it performed fairly well on the *Control node network delay* set, although not optimally. What can be seen is that the results are very similar, with slightly worse performance for the *Maximum CPU load* experiment and a much better performance for the *Control node shutdown* experiment. Another parameter from log key A0003 can be seen in Table 7.6, in which the mean true positive rate is seen for different classification thresholds. Once again the test set performance is slightly better than the validation set, however, overall the result is good.

7.7 Ensemble models

Figure 7.23 shows the performance of various ensemble models with a *window size* = 60 min. In the case when using models with a larger maximum false positive rate, shown in Figure 7.23a, there is a large performance increase by using ensemble

Table 7.5: Comparison of the performance in the test set and validation set for *Parameter value anomaly detection using an LSTM*, for a specific log event and dynamic parameter in the event.

Test set	Validation FP %	Validation TP %	Test FP %	Test TP %
Control node network delay	3.0	95.9	1.0	95.7
RabbitMQ network delay	3.0	97.1	1.0	98.0
Maximum CPU load	3.0	42.8	1.0	30.2
Full user disk	3.0	1.9	1.0	0.0
Full Docker disk	3.0	100.0	1.0	100.0
Control node shutdown	3.0	2.7	1.0	21.2
Mean	3.0	56.3	1.0	57.5

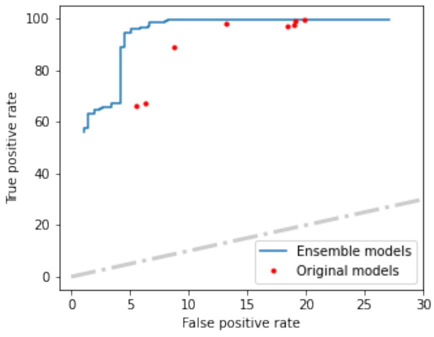
Table 7.6: The mean performance over different anomaly types for *Parameter value anomaly detection using an LSTM*. The table shows the best performance for a specific log event and dynamic parameter in the event for different false positive rates.

Validation FP %	Validation TP %	Test FP %	Test TP %
1.0	57.5	0.76	63.5
0.76	40.7	0.51	60.0
0.51	37.9	0.51	53.5

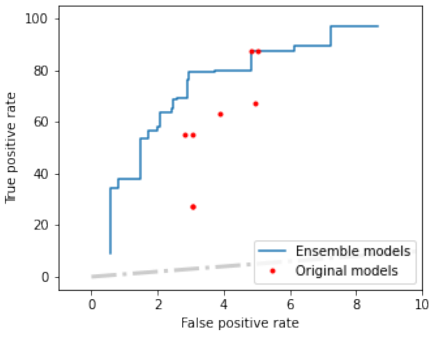
models, resulting in more than 10% decrease in false positive rate, with the same true positive rate as the best single model. When using models with a lower false positive rate, the improvement is not as large, however, there is still a distinct improvement when comparing to the results seen in Figure 7.8. Here one can trade-off a higher false positive rate for a higher true positive rate, or a lower true positive rate for a lower false positive rate, when compared to the best two individual models. The effect is similar in Figure 7.24 which shows the result for ten minute windows.

In Figure 7.24a, no ensemble model is able to get the same true positive rate at the same false positive rate as the best model. However, once again it is possible to achieve better performance at different false positive levels. The ensemble models achieve a true positive rate of approximately 47%, at 18% false positive rate, whereas the individual models from the previous grid search only achieve approximately 40% for the same false positive rate. With small differences, however, it is possible that the results do not generalize, and it may be difficult to draw conclusions. For the final case with low false positive rate and a 10 minute window, the ensemble models are in practice worse than the individual models, as seen in Figure 7.24b.

The trend clearly shows that using ensemble models can improve the performance,

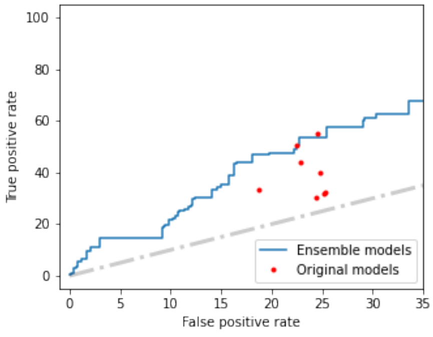


(a) Models with a maximum false positive rate of 25.

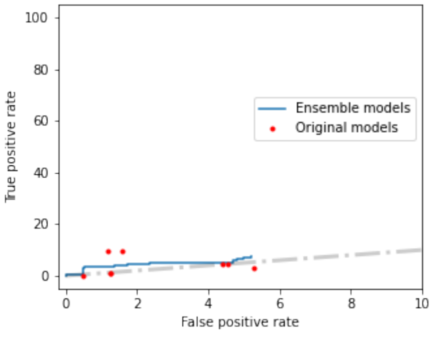


(b) Models with a maximum false positive rate of 5.

Figure 7.23: The performance of 8 individual models and the performance of different ensembles of the models on an OpenStack on OpenStack environment with *Maximum CPU load*. A 60 minute window was used.



(a) Models with a maximum false positive rate of 25%.



(b) Models with a maximum false positive rate of 5%.

Figure 7.24: The performance of 8 individual models and the performance of different ensembles of the models on an OpenStack on OpenStack environment with *Maximum CPU load*. A 10 minute window was used.

with the effect being larger when the true positive rates of the individual models are larger. This is likely because as the true positive rate is large, there is a large overlap between the models classification in the outlier set, and therefore having a high voting threshold will not change the number of true positives drastically. On the other hand, the results show that even though the models are misclassifying many vectors in the baseline set, it is not the same vectors that are being misclassified by the different models, and therefore by using a voting system, a lot of the misclassifications can be removed. In contrast, when the true positive rate is lower, it appears that the vectors classified as true positives also differ between the models, and therefore some of the vectors will no longer be classified as true positives using ensemble models. Finally, using ensemble models can also give more flexibility in tuning the false positive rate and true positive rate as more models with similar performance are produced.

Finding 12. Using an ensemble of models can improve performance, and the effect is larger when the models have a higher true positive rate.

8 Conclusion

8.1 Summary

We started off by using the historical data set given by the Ericsson's data center to build the *anomaly detection pipeline*. The created pipeline consisted of the stages: *Log Collection*, *Log Cleaning*, *Log Parsing*, *Feature Extraction*, and finally *Anomaly Detection*. Since the log data was not labeled we did not create any optimized *anomaly detection models* from said data. Instead to produce labeled data a controlled cloud environment, TripleO, was utilized on top of an existing cloud managed by the OpenStack service. Six types of anomalies were in turn injected into the system whereas the logs produced was collected. The six were: Full user disk, RabbitMQ network delay, Maximum CPU load, Control node shutdown, Full Docker disk and Control node network delay. The unstructured log messages were afterwards parsed with the help of *Drain* resulting in an *Event ID* sequence. Thereafter multiple kinds of *feature extraction* and *anomaly detection models* were applied on the parsed data. Among the majority of the optimized model created was an almost perfect detection rate for the the two network delay and filling the *Docker disk* anomalies. On the other hand the *Control node network delay* and *Full user disk* anomalies were noticeably harder to detect.

First, four different *conventional machine learning* algorithms were used in the *anomaly detection* stage. The input data consisted of *Event vectors* extracted using *sliding windows* from the parsed *Event ID* sequence. To find optimal models a grid search was performed using parameters in the feature extraction. The results showed the *window size* and *the ratio of outliers in the training set* had the greatest impact on the performance and the resulting average detection rate over all experiments was in the 50-60% range.

The LSTM neural network was also used. The feature extraction was done both on the dynamic and the constant part of the log message. The extraction resulted in time series vector of *Event IDs*, used for *Log key anomaly detection using an LSTM*. The result was that it worked very well only for the *Full Docker Disk* experiment.

For the *Parameter value anomaly detection using an LSTM* the extraction resulted in several time series of dynamic parameters. In general an average detection rate over all experiments of 50-60% was found. It was also found that there may be improvements by discarding certain parameters, and by classifying dynamic parameters separately, rather than taking a mean squared error of the prediction error for all dynamic parameters in an event ID.

8.2 Findings

All of the findings are listed below.

Finding 1. A fixed boundary of the total number of logs generated during 30 minutes or more is enough to detect the two different network delays and the Full Docker disk anomaly.

Finding 2. Larger window size achieves better results when using event vectors.

Finding 3. The detection score for a Full user disk and the Control node shutdown were substantially lower than the rest, independent of window size

Finding 4. The relative performance between the machine learning algorithms varied across the different window sizes but did not to a large extent depend on the anomaly types.

Finding 5. The model performance increased when the outlier ratio in the training set decreased as the optimal performance was reached with no contamination.

Finding 6. The Log key anomaly detection using an LSTM performs worse on our data sets than the event vector models.

Finding 7. Log key anomaly detection using an LSTM is not sensitive to hyperparameter changes.

Finding 8. Parameter value anomaly detection using an LSTM has nearly optimal performance for the Control node network delay, RabbitMQ network delay, Full Docker disk, and Maximum CPU load experiments, but does not outperform a random classifier in the Control node shutdown, and Full user disk experiments.

Finding 9. Not all log events are suited for Parameter value anomaly detection using an LSTM.

Finding 10. One can achieve a performance increase over DeepLog by having a unique LSTM for each dynamic parameter and a unique threshold for the prediction error, rather than taking a mean squared error over several dynamic parameters.

Finding 11. Parameter value anomaly detection using an LSTM is not sensitive to hyperparameter changes.

Finding 12. Using an ensemble of models can improve performance, and the effect is larger when the models have a higher true positive rate.

8.3 Discussion

For the two delay scenarios and the Full Docker disk scenario our pipeline manages to get an almost perfect classification of anomalies, and in fact only using the frequency of log messages is enough to do so. For the scenario with a *Maximum CPU load*, the frequency of log events is not enough to detect the anomaly. However, many of the models we tested were able to achieve a very high true positive rate while having a low false positive rate. For the *Full user disk*, and *Control node shutdown* experiment our models have fairly poor performance.

We investigated several models in the thesis which can be divided into the categories of event vector models, *Log key anomaly detection using an LSTM*, and *Parameter value anomaly detection using an LSTM*. The *Log key anomaly detection using an LSTM* did not work very well on the data we generated, and will need more investigation in the future as to why that is. The other two methods both worked well, and have advantages and disadvantages to using them. The *Parameter value anomaly detection using an LSTM* model has the benefit that it is an online method, and therefore can detect anomalies as soon as they happen. It is possible that in practice the event vector models with large windows are also as effective, if they are able to detect anomalies before they reach the stage where there is a fault, or failure, in the system.

Another advantage of the *Parameter value anomaly detection using an LSTM* is that it was not very sensitive to parameters, and it is likely that one can use the parameters proposed in this work on many other cloud infrastructures. On the other hand, the event vectors had several parameters that were dependant on the test set, for instance the window size needs to be tuned to the specific data, especially considering the result in the Logpai paper where longer windows were advantageous [28]. An advantage to the event vector models is that they are very fast to train, and to utilize on streaming data, as opposed to deep neural networks which take a significantly longer time, even when using GPUs. They also had slightly better performance for some of the experiments, for instance the *Full user disk* and *Control node shutdown* which were difficult to detect. Finally, as these different models look at different aspects of the data, the *Parameter value anomaly detection using an LSTM* only taking into account the dynamic parameters, and the event vector models only taking into account the relation between the number of log events in a window, and as they have different advantages and drawbacks, for a full anomaly detection both of the model types should be used.

8.4 Contributions

We present and review multiple complete *automatic anomaly defection systems*. With a controlled cloud environment we are able to generate log data with specific anomalies. Using said data the thesis evaluates the performance of multiple machine learning models for specific anomalies. This includes both conventional machine learning algorithms and neural networks. Additionally the paper shows the comparison of the use of the dynamic and the constant part of the log messages. Our evaluation further includes the impact in performance for numerous parameters all through the entire pipeline.

8.5 Future work

The results from this thesis are derived from a very controlled environment. The next step in the evaluation of the performance of the anomaly detection models would be to add a more complex baseline to the system; instead of using the simple starting and stopping of a VM, a higher and less regular load should be placed on the system. As mentioned earlier, a maximum CPU load is acceptable on servers for shorter periods of time, however the models should be able to distinguish from normal usage of the servers, and a problematic usage. Similarly, the same experiments could be done, but using different configurations, for example to investigate how the models perform when the delay is smaller, or not a constant delay. Furthermore, the pipeline needs to be tested on a production environment, where the implementation aspects such as the speed of forming windows, or using the models for inference may need to be considered further. Another aspect to be considered in future work would be to use other data sources from the data center in conjunction with the log models in order to improve performance, and investigate how the models may aid in root cause analysis, and solving the actual causes of the anomalies.

Bibliography

[1] C. C. Aggarwal. *Neural Networks and Deep Learning A Textbook*. Springer International Publishing, 2018. Chap. 2.

[2] S. Alla and S. K. Adari. *Beginning Anomaly Detection Using Python-Based Deep Learning With Keras and PyTorch*. Apress, 2019, pp. 19–20.

[3] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. "A view of cloud computing". *Commun. ACM* **53**:4 (2010), pp. 50–58. ISSN: 0001-0782. DOI: 10.1145/1721654.1721672. URL: https://doi.org/10.1145/1721654.1721672.

[4] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. *Above the Clouds: A Berkeley View of Cloud Computing*. Tech. rep. UCB/EECS-2009-28. EECS Department, University of California, Berkeley, 2009. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html.

[5] S. Beschorner. *Automated Hyperparameter Tuning of Language Based Anomaly Detection for Log Files*. University of Applied Sciences Aachen Campus Jülich, 2020.

[6] *Big data*. URL: https://en.wikipedia.org/wiki/Big_data (visited on 2021-02-08).

[7] A. Blázquez-García, A. Conde, U. Mori, and J. A. Lozano. *A review on outlier/anomaly detection in time series data*. 2020. eprint: arXiv:2002.04236.

[8] S. Borrelli. *Example of isolating a non-anomalous point in a gaussian 2d distribution*. URL: https://upload.wikimedia.org/wikipedia/commons/c/ce/Isolating_a_Non-Anomalous_Point.png (visited on 2021-04-12).

[9] S. Borrelli. *Example of isolating an anomalous point in a gaussian 2d distribution*. URL: `https://upload.wikimedia.org/wikipedia/commons/f/ff/Isolating_an_Anomalous_Point.png` (visited on 2021-04-12).

[10] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander. "Lof: identifying density-based local outliers". *ACM SIGMOD Record* **29**:2 (2000), pp. 93–104. DOI: `10.1145/335191.335388`. URL: `https://doi.org/10.1145/335191.335388`.

[11] V. Chandola, A. Banerjee, and V. Kumar. "Anomaly detection: a survey". *ACM Comput. Surv.* **41**:3 (2009). ISSN: 0360-0300. DOI: `10.1145/1541880.1541882`. URL: `https://doi.org/10.1145/1541880.1541882`.

[12] *Cinder*. URL: `https://docs.openstack.org/swift/latest/?_ga=2.60306157.1952184417.1609507891-1861056004.1606907653` (visited on 2021-01-01).

[13] *Clustering*. URL: `https://scikit-learn.org/stable/modules/clustering.html` (visited on 2021-02-08).

[14] *Confusion matrix*. URL: `https://commons.wikimedia.org/wiki/File:ConfusionMatrixRedBlue.png` (visited on 2021-03-10).

[15] *Control plane architecture*. URL: `https://docs.openstack.org/arch-design/design-control-plane.html` (visited on 2021-02-17).

[16] *Docker (software)*. URL: `https://en.wikipedia.org/wiki/Docker_(software)` (visited on 2021-03-26).

[17] *Docker overview*. URL: `https://docs.docker.com/get-started/overview/` (visited on 2021-03-26).

[18] *Docker stop*. URL: `https://docs.docker.com/engine/reference/commandline/stop/` (visited on 2021-04-07).

[19] M. Du, F. Li, G. Zheng, and V. Srikumar. "Deeplog: anomaly detection and diagnosis from system logs through deep learning". In: CCS '17. Association for Computing Machinery, Dallas, Texas, USA, 2017, pp. 1285–1298. ISBN: 9781450349468. DOI: `10.1145/3133956.3134015`. URL: `https://doi.org/10.1145/3133956.3134015`.

[20] A. Emmerich. *Automated Anomaly Detection in Software-Defined Telco Cloud Platforms*. University of Applied Sciences Aachen Campus Jülich, 2018.

[21] *Ensemble learning*. URL: `https://en.wikipedia.org/wiki/Ensemble_learning` (visited on 2021-03-28).

[22] *Fallocate(1) — linux manual page*. URL: `https://man7.org/linux/man-pages/man1/fallocate.1.html` (visited on 2021-04-07).

[23] A. C. Faul. *A concise introduction to machine learning*. CRC Press, Taylor Francis Group, 2020, p. 123.

[24] A. Gibson and J. Patterson. *Deep Learning: a practitioners approach.* OReilly, 2017, pp. 33–35, 67–69, 237.

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

[26] P. He, J. Zhu, S. He, J. Li, and M. R. Lyu. "Towards automated log parsing for large-scale log data analysis". *IEEE Transactions on Dependable and Secure Computing* **15**:6 (2018), pp. 931–944. DOI: `10.1109/TDSC.2017.2762673`.

[27] 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`.

[28] S. He, J. Zhu, P. He, and M. R. Lyu. "Experience report: system log analysis for anomaly detection". In: *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE).* 2016, pp. 207–218. DOI: `10.1109/ISSRE.2016.21`.

[29] *Hops-util-py.* URL: `https://hops-py.logicalclocks.com/` (visited on 2021-04-12).

[30] *Isolation forest.* URL: `https://en.wikipedia.org/wiki/Isolation_forest` (visited on 2021-02-09).

[31] J. Jäger. *Methods and Techniques to Artificially Create Labeled Data Sets to Evaluate Language-Based Anomaly Detection Systems.* University of Applied Sciences Aachen Campus Jülich, 2020.

[32] *K-nearest neighbors algorithm.* URL: `https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm` (visited on 2021-02-08).

[33] *Keras.* URL: `https://keras.io/` (visited on 2021-04-11).

[34] *Keystone.* URL: `https://docs.openstack.org/keystone/latest/` (visited on 2021-01-01).

[35] P. Kumar. *Step by step instance creation flow in openstack.* 22, 2018. URL: `https://www.linuxtechi.com/step-by-step-instance-creation-flow-in-openstack/` (visited on 2021-01-18).

[36] J. Leskovec, A. Rajaraman, and J. D. Ullman. In: *Mining of Massive Datasets.* 2014, pp. 8–9.

[37] Q. Lin, H. Zhang, J. 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.

[38] F. T. Liu, K. M. Ting, and Z.-H. Zhou. "Isolation forest". In: *2008 Eighth IEEE International Conference on Data Mining.* IEEE, 2008. DOI: `10.1109/icdm.2008.17`. URL: `https://doi.org/10.1109/icdm.2008.17`.

[39] *Log request id mappings*. URL: https://specs.openstack.org/openstack/nova-specs/specs/juno/approved/log-request-id-mappings.html (visited on 2021-03-25).

[40] *Long short-term memory*. URL: https://en.wikipedia.org/wiki/Long_short-term_memory (visited on 2021-02-23).

[41] P. M. Mell and T. Grance. *The NIST definition of cloud computing*. Tech. rep. 2011. DOI: 10.6028/nist.sp.800-145. URL: https://doi.org/10.6028/nist.sp.800-145.

[42] P. M. Mell and T. Grance. *SP 800-145. The NIST Definition of Cloud Computing*. Tech. rep. Gaithersburg, MD, USA, 2011.

[43] K. P. MURPHY. *MACHINE LEARNING: a probabilistic perspective*. MIT Press, 2020, pp. 2–3.

[44] *Networking concepts*. URL: https://docs.openstack.org/arch-design/design-networking/design-networking-concepts.html (visited on 2021-02-17).

[45] *Neutron*. URL: https://docs.openstack.org/neutron/latest/install/concepts.html (visited on 2021-01-01).

[46] *Nova*. URL: https://www.openstack.org/software/releases/victoria/components/nova (visited on 2021-01-01).

[47] *Novelty and outlier detection*. URL: https://scikit-learn.org/stable/modules/outlier_detection (visited on 2021-02-09).

[48] *One-class classification*. URL: https://en.wikipedia.org/wiki/One-class_classification (visited on 2021-04-12).

[49] OpenStack contributors. *Install guide*. 2020. (Visited on 2021-01-01).

[50] osrecki. *Two-dimensional outliers*. URL: https://commons.wikimedia.org/wiki/File:Two-dimensional_Outliers.png (visited on 2021-04-05).

[51] B. Parhami. “Defect, fault, error,..., or failure?” *IEEE Transactions on Reliability* **46**:4 (1997), pp. 450–451. DOI: 10.1109/TR.1997.693776.

[52] *Precision and recall*. URL: https://en.wikipedia.org/wiki/Precision_and_recall (visited on 2021-02-11).

[53] *Preprocessing data*. URL: https://scikit-learn.org/stable/modules/preprocessing.html (visited on 2021-02-08).

[54] *Pyod documentation*. URL: https://pyod.readthedocs.io/ (visited on 2021-03-26).

[55] *Receiver operating characteristic*. URL: https://en.wikipedia.org/wiki/File:Roc-draft-xkcd-style.svg (visited on 2021-03-11).

[56] *Reference architecture*. URL: https://docs.openstack.org/networking-odl/latest/admin/reference_architecture.html (visited on 2021-02-17).

[57] M. Ribeiro, S. Singh, and C. Guestrin. "
why should i trust you?": explaining the predictions of any classifier". In: *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Demonstrations*. Association for Computational Linguistics, 2016. DOI: `10.18653/v1/n16-3020`. URL: `https://doi.org/10.18653/v1/n16-3020`.

[58] G. saini. *Artificial neuron consisting of dendrites,axon and threshold function*. URL: `https://commons.wikimedia.org/wiki/File:Artificial_neural_network.png` (visited on 2021-04-12).

[59] *Storage concepts*. URL: `https://docs.openstack.org/arch-design/design-storage/design-storage-concepts.html` (visited on 2021-02-17).

[60] *Stress-ng*. URL: `https://wiki.ubuntu.com/Kernel/Reference/stress-ng` (visited on 2021-04-07).

[61] *Structure of a simple recurrent neural network*. URL: `https://commons.wikimedia.org/wiki/File:Recurrent_neural_network_unfold.svg`.

[62] *Support-vector machine*. URL: `https://en.wikipedia.org/wiki/Support-vector_machine` (visited on 2021-04-12).

[63] *Swift*. URL: `https://docs.openstack.org/swift/latest/?_ga=2.60306157.1952184417.1609507891-1861056004.1606907653` (visited on 2021-01-01).

[64] *Tc-netem (8) - linux man pages*. URL: `https://www.systutorials.com/docs/linux/man/8-tc-netem/` (visited on 2021-04-07).

[65] *Tensorflow*. URL: `https://www.tensorflow.org/` (visited on 2021-04-11).

[66] *Tripleo*. URL: `https://docs.openstack.org/tripleo-docs/latest/install/introduction/architecture.html` (visited on 2021-01-02).

[67] X. Yu, P. Joshi, J. Xu, G. Jin, H. Zhang, and G. Jiang. "Cloudseer: workflow monitoring of cloud infrastructures via interleaved logs". *SIGPLAN Not.* **51**:4 (2016), pp. 489–502. ISSN: 0362-1340. DOI: `10.1145/2954679.2872407`. URL: `https://doi.org/10.1145/2954679.2872407`.

[68] J. Zhu, S. He, J. Liu, P. He, Q. Xie, Z. Zheng, and M. R. Lyu. "Tools and benchmarks for automated log parsing". *CoRR* **abs/1811.03509** (2018). arXiv: `1811.03509`. URL: `http://arxiv.org/abs/1811.03509`.

Lund University **Department of Automatic Control** **Box 118** **SE-221 00 Lund Sweden**	*Document name* MASTER'S THESIS *Date of issue* May 2021 *Document Number* TFRT-6129
Author(s) Jacob Gummesson Atroshi Christian Le	*Supervisor* Torgny Holmberg, Ericsson, Sweden Robert Marklund, Ericsson, Sweden Johan Eker, Dept. of Automatic Control, Lund University, Sweden Karl-Erik Årzén, Dept. of Automatic Control, Lund University, Sweden (examiner)

Title and subtitle

Automatic Log Based Anomaly Detection in Cloud Operations using Machine Learning

Abstract

For modern large scale cloud services a fast and reliable anomaly detection is of utmost importance. Traditionally developers perform simple keyword search, for keywords such as "error" or "fail" in the log data, one of the main data sources that depicts the state of the system. In today's large-scale systems however several TB of log messages can be output every day making manual search highly ineffective. To address the problem there have been many anomaly detection methods based on the few publicly available log data sets. In this thesis we present a unique data collection method using a virtualized OpenStack cloud system to collect log data from six simulated anomaly scenarios. Three different detection methods are presented using both the dynamic and static parts of the individual log messages. An investigation of the impact of parameters such as time window size is done by an evaluation of the various anomaly types. Among the four conventional machine learning models based on the static parts gave a good performance of a 50% detection rate with a 0.35% false alarm rate. In addition the results show a better LSTM model performance when using the dynamic rather than the static parts. For the LSTM using dynamic parameters the results depended on the anomaly type, and the parameter, with the best average scores around 55-65% detection rate with a false alarm rate around 0.5-1%.

Keywords

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title 0280-5316		*ISBN*
Language English	*Number of pages* 1-106	*Recipient's notes*
Security classification		

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