CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Anomaly Detection in Syslog-Style Logs

Unsupervised Anomaly Detection in Highly Unstructured System Logs

Master's thesis in Computer Science and Engineering

MARCUS PETTERSSON JOHNSSON
ERIK SÄNNE

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2020

Master's thesis 2020

Anomaly Detection in Syslog-Style Logs

Unsupervised Anomaly Detection in Highly Unstructured System Logs

MARCUS PETTERSSON JOHNSSON
ERIK SÄNNE

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2020

Anomaly Detection in Syslog-Style Logs
Unsupervised Anomaly Detection in Highly Unstructured System Logs
MARCUS PETTERSSON JOHNSSON
ERIK SÄNNE

Supervisor:	Christopher Zach, Department of Electrical Engineering
Advisors:	Jesper Derehag, Ericsson
	Åke Johansson, Ericsson
Examiner:	Christopher Zach, Department of Electrical Engineering

Master's Thesis 2020
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2020

Anomaly Detection in Syslog-Style Logs
Unsupervised Anomaly Detection in Highly Unstructured System Logs
MARCUS PETTERSSON JOHNSSON
ERIK SÄNNE
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

System logs are useful resources when troubleshooting errors and faults in systems, however they become increasingly difficult to analyse as the complexity grows. In recent years, a lot of research has been conducted on creating automatic tools for anomaly detection in system logs, however these either focus only on anomalies on a message level, rely on specific properties about the dataset or require a lot of knowledge from domain experts. In this thesis, we explore the possibility of anomaly detection on a new dataset, that does not conform to the properties of current datasets available to the public, and without the need for manual intervention. We do so by clustering messages into structured events where we employ aggressive pruning rules to reduce the complexity, and then evaluate two very different models proposed in previous research. One based on Principal Component Analysis, which tries to capture quantitative relations between different events in the data, and one based on a Recurrent Neural Network to model sequential relationships. The models are evaluated on both publicly available datasets as well as the new dataset collected for this thesis. We conclude that automatic log analysis is a very challenging task that will require further attention from the research community if good results are to be achieved.

Keywords: anomaly detection, system logs, machine learning, computer science, thesis.

Acknowledgements

We would like to thank Ericsson for giving us the opportunity of conducting our thesis work with them, in an interesting area of research. Especially, we would like to express our gratitude towards our advisors Jesper Derehag and Åke Johansson, who has provided much needed guidance and support throughout the entire project.

We would also like to thank our supervisor Christopher Zach at Chalmers for useful insights regarding the disposition of our thesis, and for helping us with all formalities required.

Marcus Pettersson Johnson & Erik Sänne, Gothenburg, June 2020

Contents

List of Figures

List of Tables

1 Introduction

Logging is an important aspect in both development and maintenance of systems. It provides an insight into the current state of a running application and the produced logs can be used both for live monitoring and as a post-mortem analysis for many different use cases [1]. Perhaps most notably, system logging plays an integral part in troubleshooting unexpected events such as bugs and other faults, and logs are one of the primary sources of information used by developers and operators in this case. As a system grows in complexity, so too does the logging, and it becomes increasingly difficult to inspect the logs by simply looking at them. More often than not, one instead resorts to automated tools that can aid in the process. These often rely on rules such as searching for specific key words, counting occurrences of different events or checking values against a predetermined threshold. Though sometimes sufficient, these methods rely on the knowledge of domain experts [2], and also require time spent on maintenance whenever changes to the system occur [3]. As such, there is naturally an interest in fully automatic methods that can be deployed with less effort. In recent time, we've seen an increasing interest in machine learning across many domains. This also applies to the realm of log analysis and *log anomaly detection* in particular.

The premise here is that errors and faults are rarely occurring events and should thus constitute anomalies in a log file. As system logs in particular are sequences of partially organized text data, it is not trivial to know what an anomaly is. Anomalies can be portrayed in many different ways depending on what type of discrepancy it depicts, and what is interesting to detect varies of course from case to case.
As log messages are produced by print statements located somewhere within the source code of the system, each message is tied to that specific point of execution [1]. As such, the rationale is that a sequence of log messages constitutes a record of the systems execution. If an abnormal sequence of logs can be distinguished from a normal sequence, we call this an anomaly in the *execution path* of a system, and these are the anomalies that this thesis explores. While many problems can be exposed through individual lines in the log, for example rare messages tend to indicate untypical states for the execution, this is not always sufficient for pinpointing actual causes of faults. Also absence of messages, and the relationships between different types of message events are important to consider [4].

Though anomaly detection in logs has been an ongoing research topic for some time, there has historically been an discrepancy in the results and especially in generalization across different datasets. This is mainly due to a lack of availability of different types of log datasets. In recent times however, an effort has been made to collect datasets used in different research [5], which many studies in recent time have relied upon. Most of these datasets are not annotated however, leaving only a few that can be used for proper benchmarking. Though they vary a lot, they are all fairly well structured, something that is not necessarily the case for many other system logs in the industry.

1.1 Problem definition

This study aims to investigate the possibilities of anomaly detection in highly unstructured distributed system logs.

This will be experimented upon with a new dataset collected at Ericsson, where the ultimate goal is to create a tool that can provide hints in fault analysis, with as little manual intervention as possible.

1.2 Limitations

This thesis will only compare two different anomaly detection algorithms due to time restrictions. Additionally, it will only consider execution path anomalies, while some related work has explored different types of anomalies as well.

Only a subset of the entire dataset collected will be trained and evaluated upon, also due to limitations in time and resources. Another limitation on the topic of data is that the Ericsson data is not annotated prior to our work. A small sample was sent to domain experts for annotation, and this data was used for validation only. To aid the annotation process, the data was sent for annotation after results has been gathered, meaning that our models provided hints in the annotation process. Though we aim to provide measures for both precision and recall, we reserve ourselves for some uncertainty in the results that the reader should be aware of. This is discussed further in section 5.1.4.

Due to the differences of our data compared to other datasets, this thesis puts a lot of emphasis on preprocessing such as log parsing, explained in section 2.1. While all implementations brought up in this thesis has been benchmarked, our log parsing has not been compared against "true" templates. For the Ericsson dataset, no such annotation is or will be available, and for open datasets there is a discrepancy in what is included into the parsing process, and the results deviate thereafter. Instead, we consider the parsing evaluated by the results of our anomaly detection models.

1.3 Data

Before delving into the details of the dataset composed for this thesis, it is worth to briefly discuss the data currently available to the public. At this time, there are several system log datasets gathered by LogPAI[1], a few of which comes from prior studies on the topic of anomaly detection. Most of them are collected from real industrial systems, and are thus very different from one another. However, they are all quite well formed. First of all, there is a limited amount of unique log events that can occur in the logs. This is very beneficial for both anomaly detection models, as the dimensionality is kept low, and for initial preprocessing steps aimed to translate log messages into events, see section 2.1. Furthermore, all these systems are very unified in their logging, meaning that all log messages in a log tend to follow the same structure, which has also proven to facilitate preprocessing [6]. Below, we go into detail about the different datasets used for this thesis.

1.3.1 HDFS

The HDFS logs come from the Hadoop Distributed File System (HDFS), which is a distributed solution for storage and big data processing running MapReduce[7] jobs. The data was captured while running HDFS on over 200 Amazon EC2 nodes[4].

```
081110 011237 13 INFO dfs.DataBlockScanner:  Verification succeeded for
    blk_6996194389878584395

081110 014023 5733 INFO dfs.DataNode$PacketResponder:  PacketResponder 0
    for block blk_4741107979793372752 terminating

081110 103333 9016 INFO dfs.DataNode$PacketResponder:  Received block
    blk_4263664068073345850 of size 67108864 from /10.251.214.67

081109 214009 2594 INFO dfs.DataNode$DataXceiver:  10.250.5.237:50010
    Served block blk_3166960787499091856 to /10.251.43.147
```

Figure 1.1: Examples of log messages from the HDFS dataset. The messages are arbitrarily selected and the order does not match the source log.

Several logs, such as HDFS, provides a *session identifier* for each log message, which can be used to group log messages that belong together. For HDFS in particular, these are marked `blk_xxx` (see Figure 1.1), but they are also present as for example virtual machine instances in OpenStack. A lot of previous work has focused on finding anomalous sessions in such logs [4, 8], and this setup should be beneficial from an anomaly detection stand point as a lot of noise disappears when sessions can be isolated from start to end. This also alleviates issues with interleaved log messages, due to parallelism.

The HDFS dataset available consists of a total of 11,175,629 lines across 575,062 sessions. Anomalies are labeled on a per session level, meaning that if any session contains an anomaly, the whole session will be labeled as anomalous.

[1]https://github.com/logpai/loghub

1.3.2 BGL

The BGL dataset comes from the BlueGene/L supercomputer which in June 2006 was number 1 on the top500 ranking list over supercomputers and the only computer to score over 100 TFlops/s on the Linpack benchmark [9]. The system consists of 131,072 processors and 32,768GB of memory. The logs were gathered from the system over 215 days and consists of 4,747,963 log messages of which 348,460 are labeled as anomalies (~7.3%) [3]. Unlike the HDFS dataset, BGL does not provide any natural means of grouping messages together [2], and it is also labeled on a per message basis instead of per session. This could make BGL pose a more difficult setting than HDFS for many traditional methods. Examples of log messages in BGL can be seen in Figure 1.2.

```
1121727746 2005.07.18 R00-M1-N5-C:J03-U11 2005-07-18-16.02.26.154455
    R00-M1-N5-C:J03-U11 RAS KERNEL FATAL debug wait
    enable.................0

1123025954 2005.08.02 R23-M0-N7 2005-08-02-16.39.14.159918 R23-M0-N7 NULL
    HARDWARE SEVERE NodeCard is not fully functional

1125082154 2005.08.26 R62-M1-NA-C:J14-U01 2005-08-26-11.49.14.423511
    R62-M1-NA-C:J14-U01 RAS KERNEL INFO iar 003a90fc dear 00b35db8

1127464998 2005.09.23 R10-M0-N0-C:J06-U01 2005-09-23-01.43.18.801088
    R10-M0-N0-C:J06-U01 RAS KERNEL INFO total of 4 ddr error(s) detected
    and corrected
```

Figure 1.2: Examples of log messages from the BGL dataset. The messages are arbitrarily selected and the order does not match the source log.

1.3.3 Ericsson syslog dataset

The logs used for this thesis are system logs produced by several nodes in the packet core domain, described in Figure 1.3. The particular log files that will be used come from automated stability tests performed on several nodes in the Ericsson system in a virtual test environment. What we know about the structure of the logs is that they should contain four primary phases. The beginning of the log should represent the startup phase of the node, followed by a configuration phase. The majority of messages should be log messages produced when the node is alive and running "as usual", and if all goes well, there should be a shutdown phase at the end of the file. The final dataset used consists of 1243 log files with a total of 35,339,142 lines of log messages.

This data diverges from the other datasets in several different aspects. As already explained, the data is not uniform throughout the log files. As far as our knowledge goes, the other datasets only has logs produced during "ordinary" runtime, so no startup and configuration phases for example. Additionally, as can be seen in

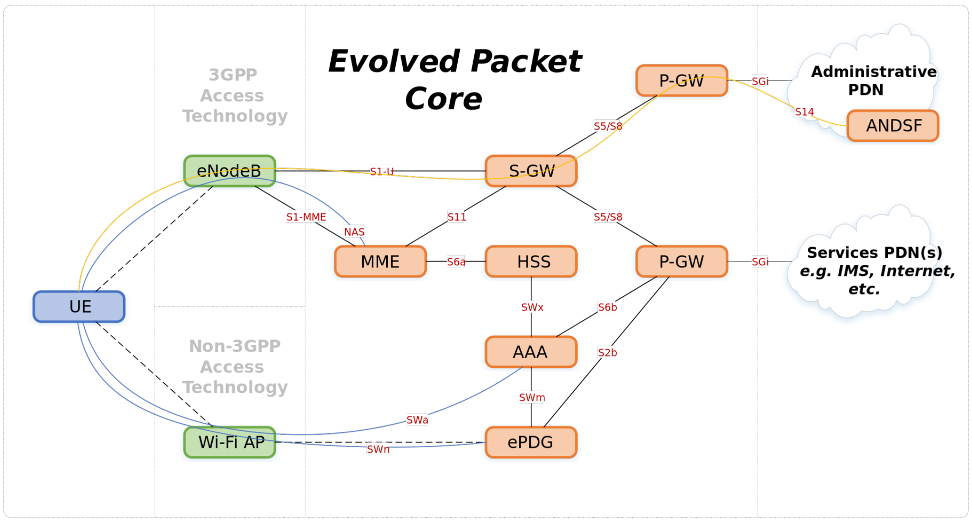

Figure 1.3: Diagram over the Evolved Packet Core, where the logs in the Ericsson dataset comes from several of the nodes. Image by Joe Deu-Ngoc / CC BY-SA (https://creativecommons.org/licenses/by-sa/4.0)

Figure 1.1 and 1.2, each log line tend to follow the same structure in many of the open datasets. This is to our understanding not always the case for many systems. In more complex systems, unified logging can fail due to several reasons such as reliance on third party software or different subsystems writing to the same log, and this is certainly an issue for the Ericsson dataset in particular. Most lines however, contains a timestamp and a loglevel, though these can occur in different places. On the same note, some log lines in the Ericsson data contains appendages such as configuration settings and other things that span over several lines.

Another major difference is that the Ericsson dataset is much more complex and contains vastly more different types logging statements than the open datasets. This makes the dimensionality of the problem much larger compared to the open datasets. Lastly, the Ericsson data is logged at a system level and the log messages come from many different components, however no natural means of grouping messages are available.

It is our belief that the structure of this dataset constitutes a fair representation for many logs in the industry, and potential means for anomaly detection considered in this thesis has been done with as little tailoring as possible to this specific dataset.

1.4 Related work

A very basic approach is to create a Finite State Automata (FSA) of the execution flow. Tan et al. proposed an algorithm based on this, which they named SALSA [10]. They automatically create state machines for each unique type of execution flow in the log file. To find anomalies, SALSA simply checks if the execution flows in the log files fit the corresponding state machine. Any operator that is to troubleshoot the program can then use the state machines to see where the error occured. A problem with this however is that they use session identifiers to untangle interleaved execution flows so they can make state machines of single execution flows. These session identifiers are not always present in log files, and to make an automaton of the whole log file with interleaved execution flows will make it very complicated and not very usable.

Another type of anomaly detection algorithm uses Invariants Mining (IM). Lou et al. [8] proposed an algorithm based on IM, where they mined invariants between templates in sequences of log messages. An invariant is a relation between the log events in the sequence. This could for example be that template A always appear the same amount of times as template B. The algorithm is trained to find many different invariants present in the data. To check if a new sequence is anomalous or not it simply checks if it breaks any of the invariants. A big problem with the invariants mining algorithm is that it is very computationally expensive to find all invariants in the training data. In their paper they managed to reduce the computational complexity by grouping logs together based on identifiers to reduce the number of templates to fit each invariant to.

One of the most successful methods right now is DeepLog [11] from 2017 which is a system that searches for many different types of anomalies. It is to our knowledge the first method to utilize a Recurrent Neural Network (RNN) for learning a normal execution path. By treating each log message as a word in a language of log events, it tries to predict next event based on the previous messages observed. If the next event is not among the top predictions of the model, it is treated as an execution path anomaly. Further more, Deeplog also use another RNN to find anomalies in variables or parameters parsed out from log messages. By learning a distribution of the parameter values of different log messages and given a sequence of values, it tries to predict the values of the next message. It also considers the timestamps of each message to learn a normal frequency of messages, in order to track the performance of the system. When an anomaly is detected, the system provides a trace of what happened and what went wrong by employing a workflow model. Deeplog's system for detecting execution path anomalies was tested on its own on the HDFS and OpenStack datasets, where it achieved good results.

LogAnomaly [12] is an even more recent algorithm which uses even more techniques borrowed from Natural Language Processing, such as word embeddings for clustering similar messages, in contrast to ordinary event clustering, and attention for anomaly detection. Like DeepLog they then use an RNN to predict the next log message given a sequence of messages, but they also use an RNN to learn quantitative relationships between log messages [12].

2
Background

Regardless of the model, the first step of any method is to transform the raw text data into a format useful for machine learning. There are a few different approaches to this which will be discussed briefly in this chapter, followed by descriptions of the different algorithms employed in this project.

2.1 Log parsing

The initial step is to transform a sequence of arbitrary text messages into a sequence of structured events [2].

As log messages are text based, some recent studies have tried employing word embedding techniques borrowed from natural language processing for clustering messages. In a study from 2017, Word2Vec [13] was used transform the words into vector representations, and a log message was encoded as the weighted average of all words it contained [14]. The rationale behind this is that similar messages should end up closer to each other in the latent vector space. An issue discussed in a later study from 2019 is that such approaches does not account for semantic differences caused by specific key words that acts like antonyms. For example, the messages `Interface ae3, changed state to down` and `Interface ae3, changed state to up` are almost exactly the same despite having the opposite meaning [12]. They instead propose a new embedding technique called Template2Vec which encodes additional domain specific information into the semantics. The results looks relatively promising, however requires additional domain knowledge and manual intervention.

Log sequence		Log templates	Event sequence
`Connected to 192.1.1.1`		`Connected to <*>`	`e1`
`Received 256 bytes from 192.1.1.1`		`Received <*> bytes from <*>`	`e2`
`Disconnected from 192.1.1.1`	$\Rightarrow$	`Disconnected from <*>`	`e3`
`Connected to 192.1.1.2`		`Connected to <*>`	`e1`
`Received 512 bytes from 192.1.1.2`		`Received <*> bytes from <*>`	`e2`

Figure 2.1: An illustration of log parsing.

A more conventional way of approaching this is to partition messages into a limited set of events, where each event relates to the print statement that produced it. This is typically done through the extraction of *log templates*. If we consider the last log message shown in Figure 1.1 of messages in HDFS, we can assume that it was produced by a logging statement that looks something as follows [5]:

```
LOG.info("Received block " + block + " of size " + size + " from " + addr)
```

If we regard all log messages produced by this print statement as being the same event, we realise that the unique fingerprint of each event is constituted by the constant string parts of the logging statements. Hence, we can think of template extraction as a reverse engineering of the original print statement, and for this one resorts to using a *log parser*. The output of the log parser given a sequence of log messages is a sequence of structured templates which are used to cluster events from the same origin. All messages produced by the print statement above would typically be parsed into `Received block <*> of size <*> from <*>`, where parameter values have been replaced by a generic variable token.

Preprocessing and pruning of the log messages based on domain knowledge tends to simplify the process of log parsing [6]. For most systems, logging is to some degree unified trough the use of a logging framework. In HDFS for example, the initial five tokens are always on the form `<Date> <Time> <Pid> <Level> <Component>`. Additionally for HDFS in particular, all messages are known to contain at least one session identifier (`blk_`) which can easily be found using regular expressions. These tokens are often removed all together prior to the parsing process as they do not contribute to distinguishing messages apart.

In a study from 2019, the performance of several different open source log parsers was compared across multiple different datasets [5]. The results showed that the parsers yielded different accuracies for each dataset, and no single parser came out on top for every case, however *IPLoM* and *Drain* showed the best *robustness*, yielding the best accuracies across most datasets.

2.1.1 IPLoM

IPLoM [15], short for Iterative Partitioning Log Mining, is an offline clustering based log parser that clusters log messages in three steps. Step 1 is to simply group together log messages that have the same word count. Next, the algorithm assumes that the word position with the fewest unique words within a cluster is most likely to contain constants. So it splits all clusters on the word position with the fewest unique words so all clusters have at least one word in common. In step 3 it searches for bijections between two word positions, *p1* and *p2*, which are computed based on their cardinality. A *1-1* relation, meaning that a word in position *p1* is always related to one word in position *p2* and vice versa, strongly suggests that the positions represent constants and shouldn't be split further. *1-M* and *M-1* relations mean that a word in position *p1* is related to multiple words in position *p2* (for *1-M* relations) or the other way around (for *M-1* relations). In these cases, the position with multiple words could either be variable, or there could be multiple templates in the cluster. The decision is made based on a the ratio between unique words in the set and the number of lines that have these values compared to a threshold provided by the user. *M-M* relations are when there are multiple unique words in both position *p1* and *p2*. In this case the algorithm splits the cluster into multiple clusters with *1-M* relations. Once all the steps are done the algorithm assumes that all messages in the same cluster correspond to the same log template. To find the variable parts it simply compares all word positions between the logs in the cluster and if all have the same word in the word position it is considered constant, otherwise it is marked as variable.

2.1.2 Drain

He et all. proposed an online algorithm for parsing logs based on a fixed depth parse tree which they named Drain [16]. First, it allows users to specify common patterns in the logs based on domain knowledge so it can mark them automatically as variable. It then takes the preprocessed log message and tries to match it by traversing the parse tree from the root node. The first layer in the tree groups logs depending on their word count. It then traverses to the following nodes by selecting the node matching with the next word in the message. Any words containing a digit are automatically assumed to be variable and matched to a special word marking it as variable. When it reaches a fixed depth d it is in a leaf node which contains groups of logs that are deemed similar. Drain tries to find the most similar group of logs by using a similarity function defined as:

$$simSeq = \frac{\sum_{i=1}^{n} equ(seq_1(i), seq_2(i))}{n} \tag{2.1}$$

where $seq_1(i)$ is the ith word of the log message and $seq_2(i)$ is the ith word of the template of the token group and equ is defined as:

$$equ(t_1, t_2) = \begin{cases} 1 & if \quad t_1 = t_2 \\ 0 & otherwise \end{cases} \tag{2.2}$$

When it has found the group with the highest $simSeq$ it compares it to a similarity threshold provided by the user to see if it is similar enough to the group to be part of it. If not, it creates a new log group, otherwise it is added to the log group and the template of the log group is updated to fit the new log message as well.

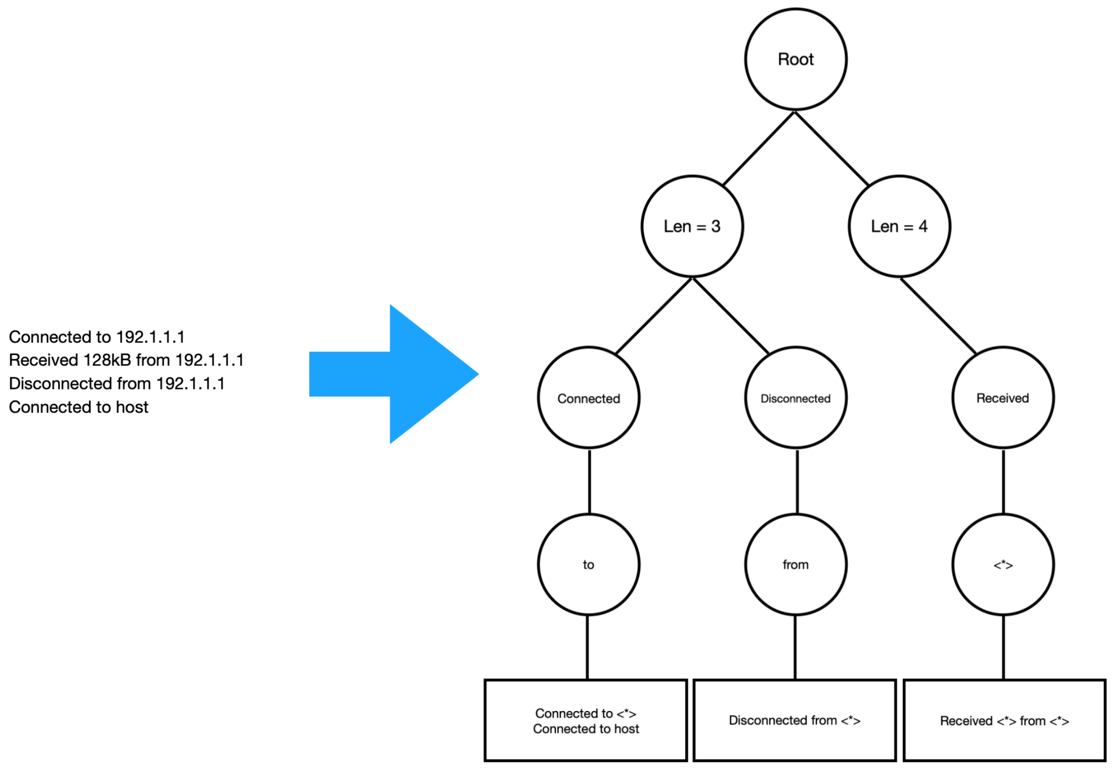

Figure 2.2: Example of how Drain constructs a parse tree from four log messages. In this example, the fixed depth is set to 3.

2.2 Feature-vector based models

Many anomaly detection models from other fields has been applied also to logs. Most of these methods requires numerical feature vectors to be extracted, typically on the form of event count vectors over smaller sequences of log messages referred to as *windows* [2]. How these windows are created, i.e the sequences are sliced, depends on the data. For logs like HDFS and OpenStack, session identifiers can be used to isolate the life cycles of individual sub tasks [11]. When no such identifiers are available, windows are instead created based on the order of the events in the logfile. Typically one uses a *sliding window* approach, of several partially overlapping slices. A *window size* is defined as a time frame for which events are grouped by, and a *step size* handles how often a new window is created. Prior work suggests that session windows tend to produce better results than sliding windows [2].
When count vectors have been created, it essentially becomes a classification problem of distinguishing normal from abnormal feature vectors. Further preprocessing such as weighting and normalization can be applied depending on the model.

Supervised methods such as Support Vector Machines perform well on this sort of problem, however they require annotated training data which is typically not available rendering them infeasible for many situations [2]. Unsupervised methods tend to be of more practical use and various different approaches have been proposed.

Clustering based methods are common examples of these, which aims to create clusters of similar log sequences and use the cosine similarity between a sequence and the created clusters to classify anomalies. There are a few methods employed on system logs in prior studies that are based on clustering [17, 18].

Another group of interesting methods are based on finding association rules between axes in the data. In log analysis this has been proposed as Invariant Mining, which tries to find linear relationships between the occurrences of events [8]. For example, one can consider two events `Connection to <*> open` and `Connection to <*> closed`. A reasonable assumption is that a connection should always be closed after it is opened, and if so there should be an equal number of both events across each sequence. If this is the case, Invariant Mining would find this relationship and treat any sequence that does not conform to this property as an anomaly. This approach has shown promising results on several datasets in prior studies [2, 11].
Yet another approach is based on Principle Component Analysis, which is a method to project multidimensional data to a space of lower dimension that captures most of the variance [2]. In anomaly detection, one makes use of the fact that a change in variable correlation causes a bad projection, and the distance can be used as an indicator [19]. This method was introduced to system logs in 2009 [4].

2.2.1 Principal Component Analysis

Principal Component Analysis (PCA) was first formulated back in 1901 by Pearson [20] and is a multivariate data analysis algorithm. Since then PCA has evolved and been applied to many different areas of research, such as geology, anomaly detection and chemistry. One interpretation of PCA is that it projects data of M dimensions to a lower dimension N which captures the dominant properties of the data.

PCA takes data as an $M \times N$ matrix where the rows are called objects and each row corresponds to a sample of data. The columns are known as variables and represent measurements that has been taken on each sample [21]. The algorithm considers each row in the matrix as a data point in an N dimensional space and tries to fit a hyper plane of A dimensions to the data, where $A < N$, by using a least squares method. Since the hyper plane can be seen as an A dimensional space, PCA can be thought of as projecting the data points in N dimensional space to a lower A dimensional space. Mathematically this can be described by the formula

$$X = TP' + E \tag{2.3}$$

where X is the original $M \times N$ matrix, T is the subspace to which the points are projected, P' is the projection matrix and E contains the deviations between the projected points and the original [21].

2.3 Sequential models

Another approach is to model execution paths based on the order of events. Some work has used automatons to create workflow models which can be used to identify abnormal executions [10, 22]. In order to retrieve workflow information in distributed systems however, these methods require the presence of identifiers to aggregate events by their corresponding task executions [22].

In most recent years, some promising methods has been proposed that uses recurrent neural networks (RNNs) [11, 6], which has previously been exposed a lot in various tasks related to natural language processing. The fundamental reasoning behind this is that a sequence of logs can be seen as a structured language as the logs are produced by the execution of structured source code. Even though this analogy might not hold when logs are interleaved, it has been shown to handle concurrent execution to a certain degree [11].

During training, only logs that are known to not contain any anomalous data is given to the network. The recurrent network is then trained to predict the next event from a sequence of log events, similar to what is done when suggesting words in a messaging app. The output of the RNN is a probability distribution over what the next event could be.

During evaluation, if the next event not among the top predictions for some threshold, the event is regarded anomalous.

2.3.1 Recurrent Neural Network

A recurrent neural network (RNN) is a network that is designed to handle sequences of data by feeding the output of the network back in to it together with new data. To update the weights of the RNN, an algorithm called Backpropagation Through Time (BPTT) [23] is often used. A problem with this algorithm is that it can lead to either exploding gradients, where the error becomes larger and larger the further it is propagated back, or vanishing gradients, where the error becomes exponentially smaller for each step back. Either of these scenarios lead to unstable and slow learning of the network.

To solve the issues of exploding and vanishing gradients, the Long Short-Term Memory (LSTM) was introduced [24]. LSTM differs from earlier RNNs by adding a memory cell with constant error flow that is protected by an *input gate* and an *output gate*. To ensure constant error flow in the memory cell, they give it the identity activation function ($f(x) = x$) and constant weights of 1. They refer to this as the *Constant Error Carousel* (or CEC). The input gate protects the memory cell from being modified by irrelevant data from the current LSTM block, while the output gate protects later LSTM blocks from being modified by irrelevant data in

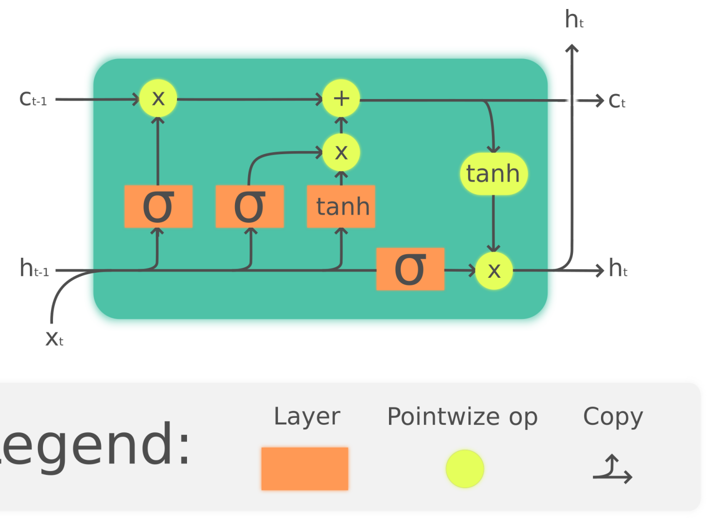

Figure 2.3: A figure of an LSTM block. x_i is the input, h_i is the output and c_i is the memory cell of LSTM block i. By Guillaume Chevalier / CC BY (https://creativecommons.org/licenses/by/4.0)

the current LSTM block. The equations for the forward pass of the LSTM are:

$$\begin{aligned} i_t &= \sigma_g(W_i x_t + U_i h_{t-1} + b_i) \\ o_t &= \sigma_g(W_o x_t + U_o h_{t-1} + b_o) \\ \tilde{c}_t &= \sigma_h(W_c x_t + U_c h_{t-1} + b_c) \\ c_t &= c_{t-1} + i_t \circ \tilde{c}_t \\ h_t &= o_t \circ \sigma_h(c_t) \end{aligned} \tag{2.4}$$

The LSTM has since its introduction been developed further. Gers et al. [25] for example noticed that the values of the memory cells tend to grow larger and larger when running on multiple blocks as more information is piled on. Their solution to this was to introduce the *forget gate*, which can be used to reset specific memory cells back to zero if the model doesn't need them anymore. The equations for the forward pass of the LSTM (equations 2.4) is extended by the equations:

$$\begin{aligned} f_t &= \sigma_g(W_f x_t + U_f h_{t-1} + b_f) \\ c_t &= f_t \circ c_{t-1} + i_t \circ \tilde{c}_t \end{aligned} \tag{2.5}$$

In 2009, an LSTM model won the ICDAR arabic handwriting recognition competition for best accuracy [26]. Another LSTM model also won the award for the fastest algorithm in the same competition. In 2015, Google started using an LSTM based model for speech recognition on the Google app and Android devices [27].

2.4 Active learning

Training a machine learning model, whether it is supervised or unsupervised, can often require lots of labeled data. If this data is very hard or time consuming to label, it can be very expensive to label all data. This is where *active learning* helps. In active learning the model has a large set of unlabeled data and can query the annotator (known as *oracle*) with datapoints it wants to get labeled. In this way the annotator doesn't need to label the whole dataset, but rather a subset of it that the model is most unsure of [28].

There are mainly three different strategies to active learning, namely *membership query synthesis*, *stream-based selective sampling* and *pool-based active sampling* [28]. In membership query synthesis, the model itself comes up with a datapoint it wants to get labeled. These datapoints don't have to be generated by the underlying distribution. Stream-based selective sampling however, is when the model draws a new datapoint from the distribution and sends it to the oracle for labeling. Lastly, in pool-based active sampling, the model has a large set of unlabeled data from which the model selects a small subset which is sent to be labeled and added to a small set of labeled samples.

3
Method

3.1 Environment

3.1.1 LogPAI Framework

LogPAI is an open source platform for automated log analysis[1]. It provides a collection of several public datasets gathered from other studies (i.e HDFS) as well as benchmarks for different log parsers and models on these datasets. It also provides a framework with Python implementations of some parsers and models published in previous research. These implementations are out of the box tailored to the the test data included for corresponding model, however they were used to establish base line measures to compare our results with.

3.1.2 Hardware

Testing and model training has been performed on a compute server at the Ericsson office. It is equipped with two Intel Xeon E5-2650 v4 CPUs with a total of 24 cores / 48 threads, 128 Gb of memory and two Nvidia Tesla P100 GPUs.

3.2 Collect data

For this thesis, we were granted access to a test portal containing the logs produced by runs of several artificial stability test cases on the system components. A set of tests, varying depending on the changes, are run whenever a new build for this system component is created, which happens on a daily basis. The logs for each test is stored for a limited period, approximately two weeks, before getting replaced by newer log files.

The individual log files are grouped by which among the several test cases it originates from, and a status verdict of each test can be retrieved from meta logs produced by the test system. As such, each log file for a particular test case is essentially marked as either a pass or fail, depending on rules established by domain experts. Reasons for a test to fail are memory leaks, CPU spikes, message drops and crashes. By what criterion exactly a test has failed is unknown to us, and there is no certainty that the cause for a failed test is represented by an execution anomaly in the logs

[1]http://www.logpai.com/

or that the verdict is observable at all from within the logs. These logs could also include faults that are not related to the packet core components themselves, but the surrounding environment and test system as well. The premise however, is that a passed test should be representative for a normal execution.

To build a sufficiently large dataset, the first step was thus to create a script that periodically scrapes the test portal of logs and stores to a local log bank, and categorize them depending on the verdict parsed from the meta logs. Logs from all test cases have been gathered, but in the end only the largest test case was used in order to keep the data as uniform as possible. It was also deemed to contain sufficiently many logs of both passes and fails to train and evaluate all implementations for this thesis.

3.3 Create a parser

As the arrangement of the Ericsson dataset is a collection of separate log files, as opposed to one large logfile (eg. HDFS and BGL), a system that can properly parse those are needed.

Other implementations, such as those provided by LogPAI, handles multiple files by parsing them separately and using a hash of the final log template as an identifier. This should in theory result in the same event ID's for the same log messages across all logs. Initial testing however, proved this insufficient for the Ericsson dataset. With each new log file, the number of extracted events increased way too much for what can be reasonably caused by truly unique events in each file. When inspecting the results, it became apparent that there were many templates that were mostly similar but had slight differences. We identified two possible reasons for this.

1. Many events are very rare and may occur only a few times or even only once in a file. Due to the nature of the parsing methods, these messages would be parsed to mostly constants as there are not enough occurrences per file to capture any variable tokens. As such, this would result in several different events when when the events of several files are combined.
2. There might be parameters for certain messages that stays the same across an entire file. Let's as an example consider a frequently occurring log message containing the current date. As the date stays the same across the entire log, the parsing algorithms would treat that as a constant in the final template. This would mean that two tests run at separate days would result in two different templates for the same log message.

Not only does this explosion of templates increase the complexity for any machine learning model, but also pose an immediate threat to the ability to learn any properties about the system's execution as there is a mismatch between extracted logging templates and actual system events.

Another issue regarding the parsing of the Ericsson dataset is its lack of unified logging statements. Open source parsers makes use of prior information to reduce the difficulty of the parsing task to a degree which is not possible with the Ericsson dataset. As such, efforts will be made to reduce the complexity without such information.

Aside from the various parsing algorithms provided in the LogPAI git repository, the source code of a variant of IPLoM written in C was also available for use at Ericsson. This implementation follows the same implementation and interface as the one from LogPAI, but without the means of providing a template for specific datasets. Due to the superior time performance of compiled code, and the amount of data needed to be parsed, we decided to use this IPLoM implementation as the foundation for our project.

3.4 Models

Two different anomaly detection models was evaluated, one RNN based model and one more traditional model based on feature vectors.

For the latter a model that already had an open source implementation was preferred. Invariant Mining has shown the most promising results in prior studies [2, 11, 12], however it does not scale well when the data becomes too large [8]. Instead, we went for a subspace model based on PCA which in essence is a very simple model and should thus not pose any major issues when run on the Ericsson dataset.

Due to the recent successes with using RNNs to learn a normal execution flow, this is also one of the methods that was used for this thesis. Aside from the promising results reported on other datasets [12, 11], there are a few other notable aspects with this approach that makes it suitable for our setup. As the Ericsson dataset is naturally divided by passed and failed tests, where passed tests should represent a normal execution, the situation is very favorable for using a model that can utilize this setup. Furthermore, as the Ericsson data is not uniform throughout the file, i.e contains different phases, we expect there to be at least some sequential relationship between log messages, something that most counting based methods might struggle with.

Python is used for all implementations, expect the parsing.

3.5 Evaluation process

The performance of the models was measured in *precision*, *recall* and *F-measure* since this is a common way of measuring detection accuracy. $Precision = \frac{TP}{TP+FP}$ and represent how many of the anomalies marked by the model actually are anomalies. $Recall = \frac{TP}{TP+FN}$ and represent how many of the total amount of anomalies the model found. $F\text{-}measure = \frac{2 \cdot precision \cdot recall}{precision + recall}$ is the harmonic mean between the two and is the metric that is to be optimized. By using F-measure as a score for how well the models perform, one can not get away with either just marking the whole file as anomalous to get the recall up since the precision will then be abysmal. Likewise one can not only mark anomalies that the the model is really certain about to get the precision up since that would miss a lot of anomalies and the recall would drop.

3.5.1 Baseline result

Baseline results has been established on HDFS and BGL. Both these datasets should pose a relatively simple setting compared to the Ericsson data, and both are annotated with anomalies.

HDFS was used mainly to validate the implementations of the various stages in the pipeline. Even though there are other annotated datasets available that closer resembles the Ericsson dataset, there are a few reasons to why HDFS was chosen in particular. First of all, LogPAI provides some utility code for HDFS specifically. This includes partitioning of the logs with respect to the session identifiers. This should facilitate the establishment of initial baseline results. The main reason however due to why HDFS was chosen, is that benchmarks on HDFS are available in almost all related literature. This is very good because the performance can be easily compared across all implementations on a single dataset. HDFS was also used to compare the performance of the new parsing algorithm against LogPAI's implementation of Drain, which is claimed to be the most robust parsing algorithm, yielding the best accuracy across most datasets. Previous studies suggests that the accuracy for Drain on HDFS is above 0.99 [5, 16]. For HDFS, each session is annotated.

BGL was chosen due to its lack of identifiers, which should make it more similar to the Ericsson dataset yet provide an easier setting. For BGL, anomalies are marked on a per message level.

As the subspace model and RNN are fundamentally different with regards to the input, there is a discrepancy in granularity of the results. For HDFS, the session labels are used for both models despite the RNN's ability to classify individual lines. For BGL, the RNN will classify individual lines, while the subspace model will classify the several lines encoded by each count vector. If a window truly contains an anomalous line, the entire window is treated as anomalous, which results in a different setting for the subspace method.

3.5.2 Ericsson dataset

When running the models on the Ericsson dataset, we assume that logs from successful test runs are representative of normal execution flow, i.e. that they do not contain any anomalies. This is important for the RNN which is only supposed to train on normal data.

Since the Ericsson data was not labeled prior to this work, domain experts helped us label a small subset of the dataset which was used to measure the performance of the models. Since it is very time consuming to label the log files, we used an active learning approach to help guide the experts in their labeling. The models marked what they think is anomalous in the files and the experts used these markings as guidance to where possible anomalies could be. Since all the unlabeled data was available from the start and the model only marked a small subset to be labeled, this is similar to pool-based active sampling as mentioned in 2.4. They tried to find anomalies outside what was marked by the model as well, but they focused most on what the model marked. Anomalies are labeled on a per line basis. When the dataset was labeled, we used it to tune and evaluate both models on it. The RNN needs to mark the exact line of the anomaly, while the subspace model needs to mark all windows containing the anomaly. For example, if 10 windows overlap the anomaly, and 8 are marked by the subspace model as anomalies, this counts as 8 true positives and 2 false negatives.

4 Results

4.1 Parser

In order to cluster events, a log parser was needed. As the open source implementations available did not suffice for the Ericsson dataset out of the box, some novel modifications were made. The foundation of the parsing algorithm is based on an implementation of IPLoM provided by Ericsson. Additionally, due to the size of the Ericsson dataset, and the way it is vectorized for the RNN model, it is infeasible to load everything into RAM at once during training. In order to support for only a few logs to be loaded at a time, the parser was split into two parts. A database creation part for creating an initial template database, and a template prediction part. The database creation part can also be used without the predict part by saving the parsed log file to disk, and we used this setup for the HDFS dataset as this allowed us to use some utility code from LogPAI for extracting sessions.

4.1.1 Database creation

The fundamentals of the database creation follows IPLoM closely with some pruning rules to reduce the number of templates in large amounts of unstructured data. This section won't go into detail about how IPLoM works (see 2.1.1), but rather how the provided version was modified to fit our needs.

At first the parser could only parse one log file at a time and as discussed in Section 3.3, our setup of logs could not be sufficiently parsed by parsing each log independently. To allow for multiple files to be parsed at the same time, the parser was modified so it could run step 1 of the IPLoM algorithm for each log file to add all messages to the model. Once the messages from all log files has been added, it continues with the rest of the steps as normal. This change reduced the number of templates massively, as can be seen in table 4.1. A database file is created with all extracted templates and their corresponding event IDs.

Another issue for the parsing process was the lack of uniformity between logging statements and the sheer amount of data needed to be included to capture enough templates. In an attempt to alleviate this problem, some very naive and aggressive pruning was done. In order for Drain to prevent a branch explosion in their parse tree, they simply assume all digit tokens to be variables [16]. This is in it self a fairly naive assumption, however we realised that this could reduce the complexity of the Ericsson dataset drastically. At first, this was explored by parsing all digits

Table 4.1: The effects of different means to reduce the number of templates. The results are from the parsing of all successful test cases in the final Ericsson dataset.

Optimization	# Templates	% reduction
Baseline	487,313	0%
(1) Collapse lists	482,217	1.0%
(2) Remove tokens with digits	7,233	98.5%
(3) Parse logs simultaneously	20,113	95.9%
(1) + (2)	6,960	98.6%
(1) + (3)	19,922	95.9%
(2) + (3)	4,188	99.1%
(1) + (2) + (3)	4,089	99.2%

to a special token `</d>` prior to further clustering, however this caused two issues to arise. As the `</d>` tokens were included into the parsing process, some messages were grouped by only this token surrounded by other variables. On the other hand, a digit token could also be treated as a variable together with other string tokens. In a second attempt, digits were replaced after the initial clustering, however this caused issues for templates that had been grouped only by digits. As both approaches resulted in too generic templates, essentially consisting of only variables that could capture very different messages, we decided to remove these tokens all together, similarly to what is often done in preprocessing based on domain knowledge (see Section 2.1). This is a bit dangerous as the position of digits are not tracked and a lot of information is essentially lost, however this enforces at least one unique string token to separate different events apart. As a last aggressive means of pruning, we expanded on this idea to remove any token containing a digit, even those who also contained other characters. The idea is that this would include for example IP-addresses and different types of identifiers which in theory should be parsed as variables. In order to not prune away too much, some additional splitting of tokens was done based on special characters.

A final amendment introduced was to collapse lists of variables. The idea is that this could help in rare instances when the variable parameters in a message are lists of variable length between messages of the same type.

It should be mentioned that due to these changes, the extracted templates do not represent the print statements in as a structured manner as previous parsing methods. In our case, we only use templates to cluster events, and thus we believe this to be a reasonable approach.

4.1.2 Predicting templates

Unlike the database creation mode, the prediction mode does not use IPLoM. Instead it uses a parse tree which is constructed using the database file. To do this it starts from the root node and takes a template from the database and for each word in the template it tries to find a child node related to it. If there is a child node it moves there and tries to find a child for the next word, otherwise it creates a new child node that will be related to the current word. When it has looped through all words in the template it has reached a leaf node and it will store the template. It continues this until all templates have been added to the tree. See Figure 4.1 for pseudo code.

When it later needs to look up a template for a new log message, it first removes all tokens containing digits in the line using regex, same as in the database creation phase. Then it tries to traverse the tree by starting in the root node and for each word in the message it checks if the node has a child node related to a variable. If it has it adds it to a First-In-Last-Out (FILO) queue with nodes that are to be searched. Then it also checks if the current node has a child related to the word position and adds it to the queue. It then pops a node from the queue and continues from there with the next word in the message. Since the queue is FILO and the algorithm adds the string node last, it will always prioritize constants before variables. Once it has gone through all words in the message it checks if the node is a leaf node with a template. If it isn't the algorithm pops another node from queue and continues. If the queue becomes empty without the algorithm finding a template it is assumed that the template is not present in the database. In this case, a 0 is returned which is a template for *out of vocabulary*, meaning that template wasn't found. See Figure 4.2 for pseudo code.

```
function build_parse_tree(template_database):
    for template in template_database:
        current_node = root
        for word in template:
            next_node = current_node.get_child(word)
            if next_node != None:
                current_node = next_node
            else:
                current_node.set_child(word)
                current_node = current_node.get_child(word)
        current_node.template = template
```

Figure 4.1: Pseudo code for creating the parse tree.

```
function find_template(message, parse_tree):
    remove_digits(message)
    to_search = [root]

    while to_search not empty:
        current_node = to_search.pop()

        if current_node.depth == len(message) and
                    current_node.template != None:
            return current_node.template

        # Since a variable can represent a list of words
        if current_node is variable_node:
            to_search.add(current_node)
        else:
            variable_node = current_node.get_child(variable)
            to_search.add(variable_node)

        string_node = current_node.get_child(word)
        to_search.add(string_node)

    return 0
```

Figure 4.2: Pseudo code for finding a template in the parse tree.

4.1.3 Performance on open datasets

The parser's performance was compared against the implementations of Drain and IPLoM provided by LogPAI. Both implementations was tuned according to LogPAI's suggested parameters for each dataset.

The default implementations make use of more prior information about the dataset than our variant of IPLoM. For HDFS, both parsers are given a description that all lines follows the structure:
`<Date> <Time> <Pid> <Level> <Component>: <Content>`.
This essentially means that the first five tokens should be disregarded from the actual parsing process, reducing the complexity of the problem. Likewise, BGL is given the template:
`<Label> <Timestamp> <Date> <Node> <Time> <NodeRepeat> <Type> <Component> <Level> <Content>`.
Our variant of IPLoM has a more generic template of only `<Date> <LogLevel> <Content>` (if present) as those tokens are included in most open datasets as well as the Ericsson dataset. Instead of assuming that they are always present in the same location, it uses regex to actively find the date and loglevel. Also, Drain uses regex to remove specific tokens which are known to be variables prior to the parsing. This for example includes all session identifiers and IP-addresses for the HDFS dataset.

Table 4.2: Number of extracted templates by different methods

Method	HDFS	BGL
LogPAI Drain w/o knowledge	46	546
LogPAI IPLoM w/o knowledge	376	12411
LogPAI Drain	45	1605
LogPAI IPLoM	156	793
Our IPLoM w/o knowledge	41	247

It also treats any token with digits as a variable, while our parser removes such tokens all together. Because of the differences in what is actually fed into the parser, the resulting templates look quite different as well and this prevents a one to one comparison.

A comparison in number of templates produced by each parser can be seen in Table 4.2. We also include measures for how well the parsers manage to parse data without prior knowledge, however these results should be taken with a grain of salt as they are not tuned thereafter.

When running the subspace model on HDFS parsed by both Drain and our IPLoM parser, we managed to get an equal F-measure of 0.791.

4.2 Models

The RNN model was implemented using PyTorch's library for machine learning and the implementation described in DeepLog [11] was used as a primary reference when designing the architecture. An overview of the architecture can be seen in Figure 4.3. The final model supports several stacked bi-directional LSTM layers, making it a deep RNN, followed by a single layer for multi-class classification. For predicting an event e_t at position t, a sequence of length s of the previous most recent events encoded as one-hot vectors is fed to the model. For backpropagation we compute the cross entropy loss between the prediction and the actual next event, as this is a conventional approach for multi-class classification problems. Furthermore, the prediction vector is passed through a softmax activation function to get a probability distribution of event e_t. The number of LSTM layers and the size of the hidden units are left as tuneable hyper-parameters. The hidden state and cell state vectors are initialized with zeros.

Previous implementations uses a fixed value for how many of the top predictions are regarded normal (topk), which we believe is sub optimal. When the model is very sure about the next event, fewer events should intuitively be regarded normal compared to a situation where the model is very uncertain. Consider a very simple case with a log with only 5 different events in the language $(e^1, e^2, ..., e^5)$, and two sequences (seq_1, seq_2) which is fed to the model. In the training data, seq_1 is always followed by either e^2 or e^3 while seq_2 is followed by either e^1, e^2 or e^4. The resulting

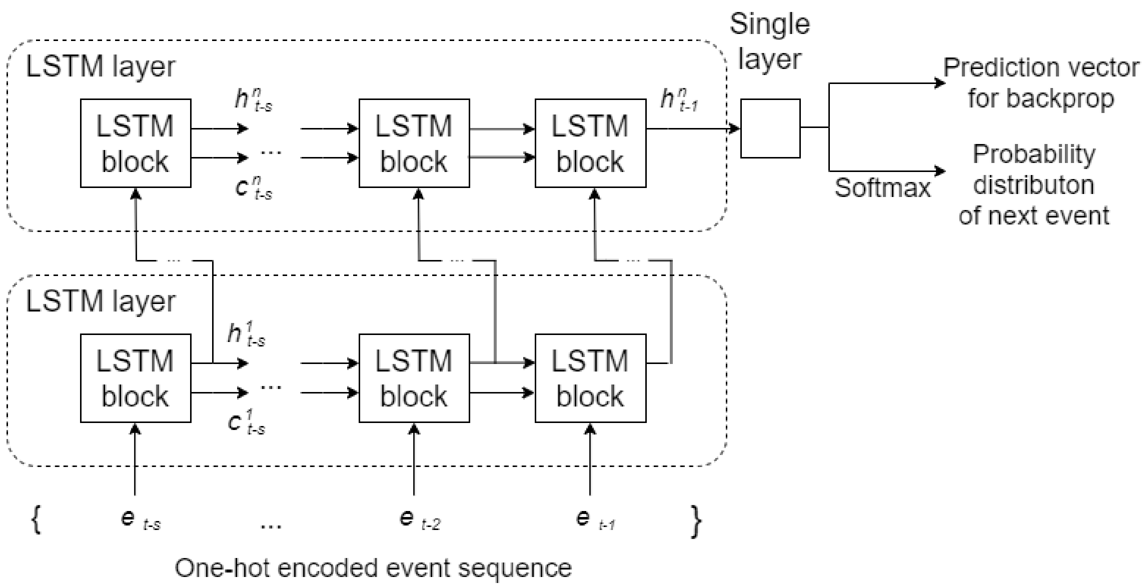
LSTM layer
LSTM block
h^n_{t-s}
c^n_{t-s}
h^n_{t-1}
Single layer
Prediction vector for backprop
Softmax
Probability distributon of next event
LSTM layer
h^1_{t-s}
c^1_{t-s}
{ e_{t-s} ... e_{t-2} e_{t-1} }
One-hot encoded event sequence

Figure 4.3: Architecture of the RNN model where n represents the number of LSTM layers, and h and c corresponds to the hidden and state vectors. While the LSTM layer is a recurrent unit, we have unrolled it for illustrative purposes. t denotes the line to predict, and e_i the one-hot encoded event at line i. s is the sequence length i.e. window size of the input.

probability distributions which the model will predict will as such be approximately $P(e_t|seq_1) = \{0, 0.5, 0.5, 0, 0\}$ and $P(e_t|seq_2) = \{0.33, 0.33, 0, 0.33, 0\}$. A fixed topk could not be tuned to properly mark all anomalies without falsely also marking normal rows. With a topk of 2, all anomalies would be marked for seq_1 however normal events are in the risk of getting marked anomalous in seq_2. With a topk of 3, abnormal events could be classified as normal in seq_1.
Two alternative approaches were tried. One implementation used a hard threshold, i.e. a fixed probability that the next event must exceed in order to be regarded as normal. In the example above, a threshold of 0.2 would consider e^2 and e^3 normal for seq_1 and e^1, e^2, e^4 normal for seq_2. This effectively works as a dynamic topk. This approach also allows the model to mark everything as anomalous, by for example using the threshold 0.6 in the example above. This method will be referred to as *RNN, threshold* in the following figures. The other approach tried uses an accumulative prediction threshold, and considers the top predicted events until their combined probability exceeds the specified value. For example, if a threshold of 0.7 is used, topk will be adjusted such that the joint probability of normal events reaches 70% confidence.
When evaluated, both methods produced better results than the fixed topk, but the fixed threshold implementation performed the best across all datasets. Additionally the threshold model allows all events to be predicted anomalous if the model is very uncertain. Comparisons between this approach and fixed topk are provided for each dataset.

For the subspace model, the fundamentals of the algorithm was taken from Log-

PAI's framework. It consists of a fit (or training) and a predict phase. Fit takes both normal and anomalous data as input and uses PCA to find a plane of $n < |\textit{unique events}|$ dimensions that fits to the input as well as possible. Based on an expected ratio r of anomalies, it then calculates a squared prediction error (SPE) threshold such that $r\%$ of data points with the furthest distance to this plane are segregated.

Predict projects the data points to be tested to the plane found by PCA during the fit phase and calculates the projection error for each data point. If the error is over the SPE threshold it is marked as an anomaly. Both fit and predict takes input data on the form of a count matrix of size $n \times d$, where n is the number of windows and d the number of unique event types. As such, each row represents a count vector over a sequence of logs. The LogPAI implementation uses a slightly different set of input parameters to automatically adjust the number of principle components and SPE threshold, however we report our results based on number of components and threshold.

As this model requires all training data to be available at once, slight modifications to the implementation was done in order to support more data. If the window size is kept relatively small compared to the number of event types, most entries will be zero, and thus sparse matrices were used to save memory. After vectorization, all vectors are weighted by inverse document frequency as rarer event should have a greater impact on the result [4]. In some earlier work, each vector has also been normalized by zero-mean [19], however this was deliberately left out as that would had required dense matrices.

Both models use our version of IPLoM to parse log messages.

4.2.1 Baseline results

The models were first tested on the HDFS and BGL datasets in order to validate the implementations. A template database for event clustering was created on the entire dataset, including test data, however during vectorization, only templates in the training data was included into the vocabulary. If a new event occurs in the predict / test phase, it is treated as a 0 or out of vocabulary. After vectorization, 50% of all normal and abnormal data is randomly selected and put into a test set. The F-measure for each model and dataset can be viewed in Figure 4.4

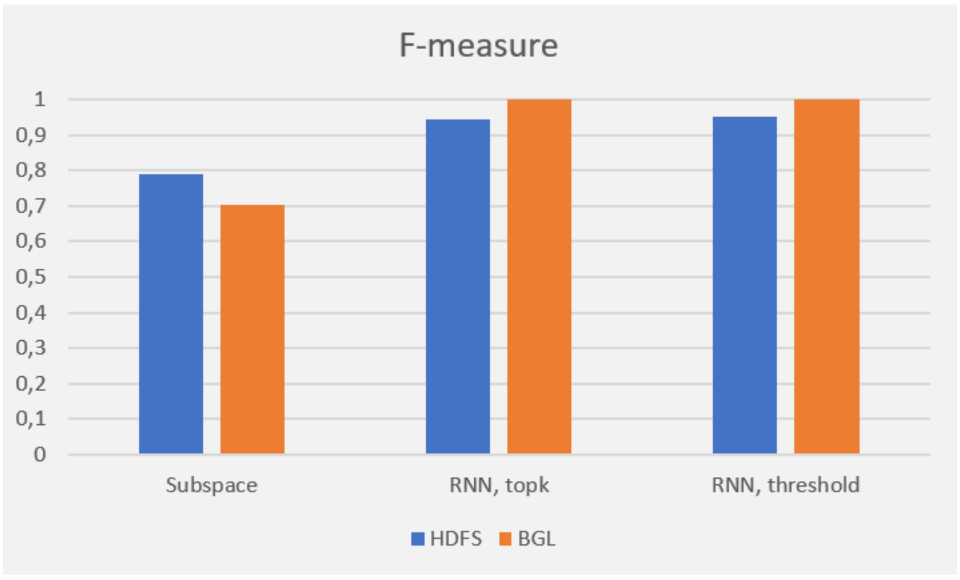

Figure 4.4: F-measures for all models on HDFS and BGL

4.2.1.1 HDFS

After event clustering, all events were grouped by their corresponding session identifiers. Out of the total 575,062 sessions extracted, 50% of all normal sessions and 50% of all anomalous sessions where randomly sampled into a test set.

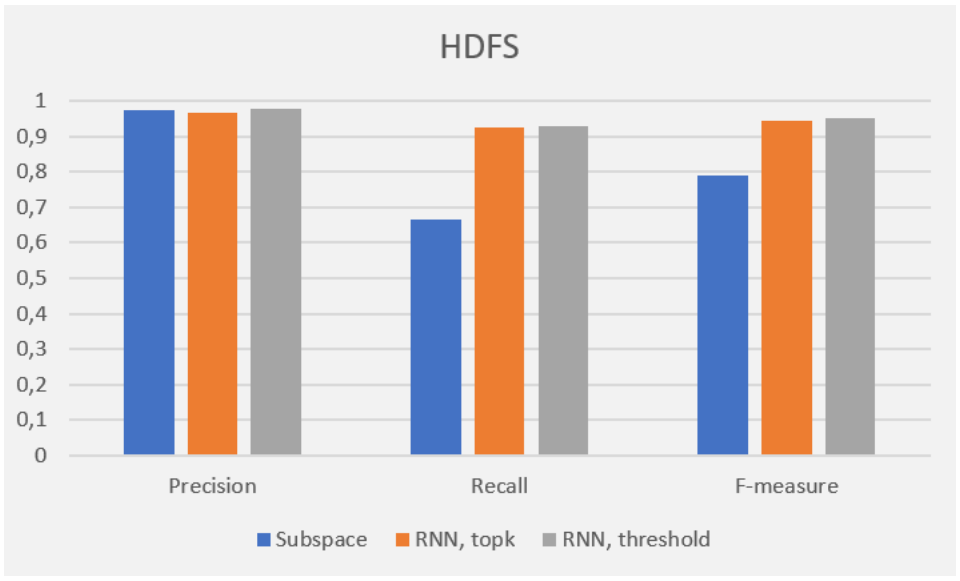

Figure 4.5: The best results of the models on the HDFS dataset compared against each other.

For the subspace model, all remaining sessions were put into the training/fit set, and event count matrices were created for both sets respectively. As the subspace model in essence is an offline method, we used the score reported on the training data for tuning the model parameters. The best results was produced by using 5 principle components and an SPE threshold of around 6.56, which on the test set yielded a precision of 0.98, recall of 0.67 and F-measure of 0.79 which is on par with previous studies.

Table 4.3: Precision, Recall and F-measure for different values of the parameter *topk* on the HDFS dataset.

topk	Precision	Recall	F-measure
9	0.99398	0.88193	0.93461
7	0.98221	0.89833	0.93840
6	0.96544	0.92576	0.94519
5	0.89121	0.95356	0.92133

For the RNN model, all remaining anomalous sessions were removed from the training data and only the normal sessions were used for training and validation. 15% was put into a validation set which was only used to track the loss, and all remaining data (a total of $42,5\%$ of the normal sessions) was used for training. For each session, a sliding window of size 10 was employed to extract input sequences, followed by the next event to predict. As each session can be fairly short, they were padded with a sequence of $1's$ as placeholder events such that all events in the sequence can be predicted. Topk and threshold is not relevant for the training process and can as such be tuned afterwards. If the RNN considers any event in the session anomalous, the entire session is treated anomalous. The best results were achieved using two LSTM layers and a hidden size of 32. With a fixed topk of 6, the model managed to get an F-measure of 0.945 at best, and the results of different values for topk can be seen in Table 4.3. When instead evaluating the same model with a certainty threshold of 0.00009, the model achieved an F-measure of 0.953 (an improvement of 0.85%), which suggests that this approach is slightly better. Most notably, both the number of false positives and false negatives decreased, as can be seen in Figure 4.6

	P'	N'
P	5,598	2,821
N	137	278,975

(a) Subspace model

	P'	N'
P	7,794	625
N	279	278,833

(b) RNN, topk

	P'	N'
P	7,825	594
N	173	278,939

(c) RNN, threshold

Figure 4.6: Confusion matrices for the different models on the HDFS dataset. Rows corresponds true labels and columns predicted labels.

4.2.1.2 BGL

As no session identifiers are present in BGL, the data was vectorized by a sliding window approach for both models. The number of windows is dependant on the size and step used for each model and as such the total number of windows are not the same for each run. Still, 50% of all normal and anomalous windows respectively was put into the test set.

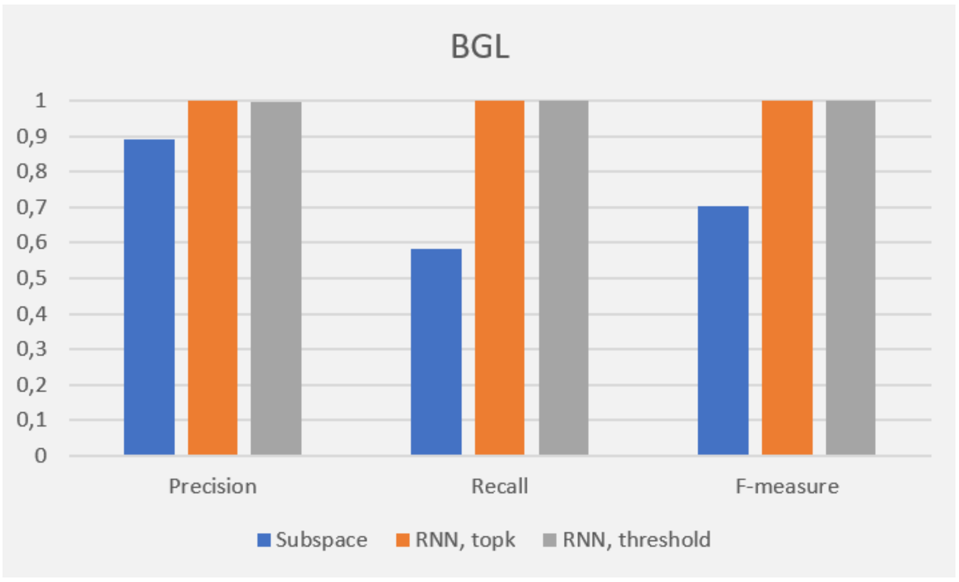

Figure 4.7: The best results of each model on the BGL dataset compared against each other.

For the subspace model, all remaining data was again used for training. As the purpose of this thesis is to provide an aid for fault analysis, the window size was initially kept low in order to increase the granularity. Unfortunately, the results were very low, and for a window size of 10 we never managed to get an F-measure that exceeded 0.1. We deem this reasonable as it is probably not enough to capture any relational properties between the events. When comparing against previous work [2] it became apparent that their score was reported on a window span of 9 hours, which translates to approximately 8000 log lines. When increasing our window size to 8000 and letting a window start every 1000*th* line, we managed to approximate their setup and match the scores of 0.71 which still is fairly low. These results were achieved using 131 components and an SPE threshold of 8.03. Despite the differences in how windows are created, we are sceptical to whether this setup is useful at all for our use case. We realized that for such a large window and step size, a third of all windows contain anomalies. Additionally, we can conclude that a window size of 8000 is not feasible in our case as the granularity of the marking becomes way too low to provide useful hints in fault analysis.

The RNN model on the other hand, managed to yield a perfect recall of 1.0 without the need for much hyper parameter-tuning, meaning that it found all anomalies right away. With our initial run using a window size of 10, a step of 1, two LSTM-layers,

Table 4.4: Precision, Recall and F-measure for different values of the parameter *topk* on the BGL dataset.

topk	Precision	Recall	F-measure
1	0.62608	1.0	0.77005
9	0.99123	1.0	0.99559
15	0.99449	1.0	0.99724
150	0.9997	1.0	0.99985

	P’	N’
P	465	332
N	57	1,517

(a) Subspace model

	P’	N’
P	174,230	0
N	53	2,199,364

(b) RNN, topk

	P’	N’
P	174,230	0
N	371	2,199,376

(c) RNN, threshold

Figure 4.8: Confusion matrices for the different models on the BGL dataset. Rows corresponds true labels and columns predicted labels. Note that the total number of datapoints are less for the subspace model due to differences in window size.

a hidden size of 64 and topk of 9 we managed to get a near perfect F1-measure of 0.996. When investigating the results, we realised that the event ID of each anomalous line was 0, meaning that they were never present in the training data. This makes the task of anomaly detection very simple for the RNN, as long as the size of the training data is sufficient to cover all normal events. Because of this, the topk parameter can be increased a lot to increase the precision, which is shown in Table 4.4. However, the precision is already very good at the fairly low topk of 9. When using the certainty threshold instead, there isn’t much of a difference in score since it already is very good. With a certainty threshold of 0.00001 it produced an F-measure of 0.997, which is roughly the same as when using a fixed topk of 9. By further lowering the threshold it produces better and better results until it starts marking anomalies as normal. In cases like these, where anomalies are constituted only by rare events, anomaly detection can be done already on an event clustering level. Yet, we deem these results good as they suggest that both the parsing algorithm and RNN works as intended.

4.3 Anomaly detection on the Ericsson dataset

For evaluation on the Ericsson dataset, only one of the gathered test cases was used in an attempt to keep the data as uniform as possible and keep the training time bearable. It was still deemed sufficiently many logs for a proof of concept. As the

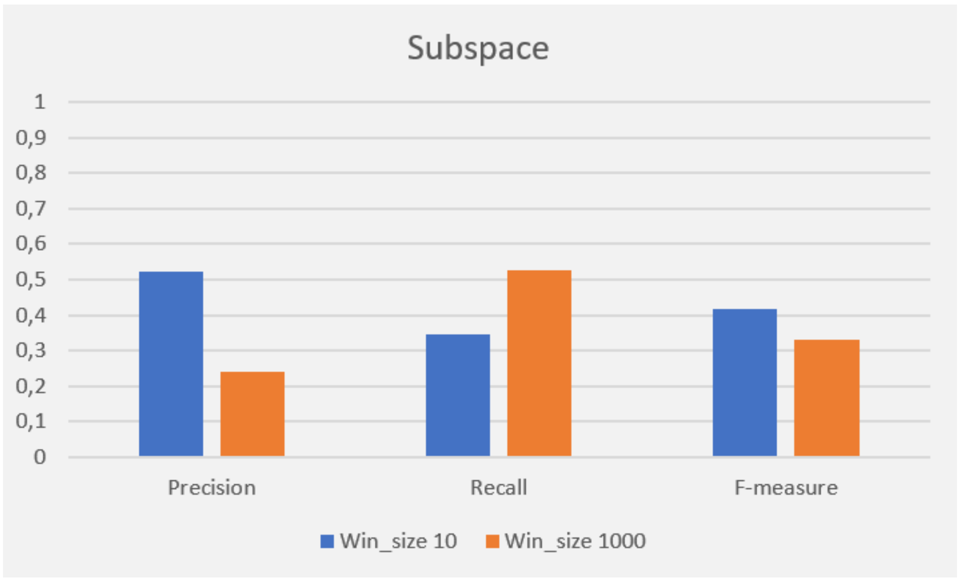

Figure 4.9: Precision, recall and F-measure of the subspace model on the Ericsson dataset with window size 10 and 1000

Ericsson data contains lines in the logs which themselves are appendages of other log events, i.e. stack traces and prints of configuration settings, we realised that some parsed templates would not correspond to actual templates in the Ericsson source code. In an attempt to resolve parts of this issue, we made one preprocessing step based on domain knowledge, namely to append all lines that did not start with a date to the end of the previous line. The desired result would be that tokens from these lines would be treated as variables and later collapsed, resulting in only a few generic templates. This covers far from all such issues, as multi line messages from some components are formatted with a date as well. A final test dataset was created with a total of 10 logs, 8 of which had failed. These were later sent for manual annotation.

4.3.1 The subspace model

Since the subspace model needs to be trained on both normal and anomalous data, it was trained on a final set of 1009 logs, 544 passes and 465 fails, which contained a total of 4789 unique templates. To start, we tried running the model with a window size of 10 and window step of 1 in hope that it would mark any anomalies more accurately to help potential users with debugging. By using 2907 components and an SPE threshold of 0.81, it achieved a precision of 0.52, recall of 0.35 and F-measure of 0.42.

Since it helped to increase the window size massively when running on BGL, we decided to try larger windows on the Ericsson dataset as well, even though this means less accurate pinpointing of potential errors. By using a window size of 1000 and step size of 100 instead, the model managed to get a score of 0.33 when using 387 components and SPE threshold of 214.28. It turns out that the subspace model performs worse on Ericsson dataset when using the larger window size, which is interesting since a larger window size performed better on BGL. As seen in Figure 4.9, when using a small window size, the precision is high while the recall is low, but when using the larger window size it is the other way around.

4.3.2 RNN

For the RNN model, the dataset was divided into training and validation sets of 640 and 113 normal logs respectively, which should all be representative of a normal execution. Initially, a template database was created only on the training set for which we saw very poor performance. We suspected that there was a risk of messages being improperly captured by the wrong template in the parsing, and to reduce the risk of this, we created a new template database on all messages, including the failed test cases. The total number of templates found then increased from about 4000 to 5000, i.e. with about 25%. Very little improvements to results were however seen. A distribution over the occurrences of each template in the training data can be seen in Figure 4.10, which suggests that there are many templates which occurs very seldom.

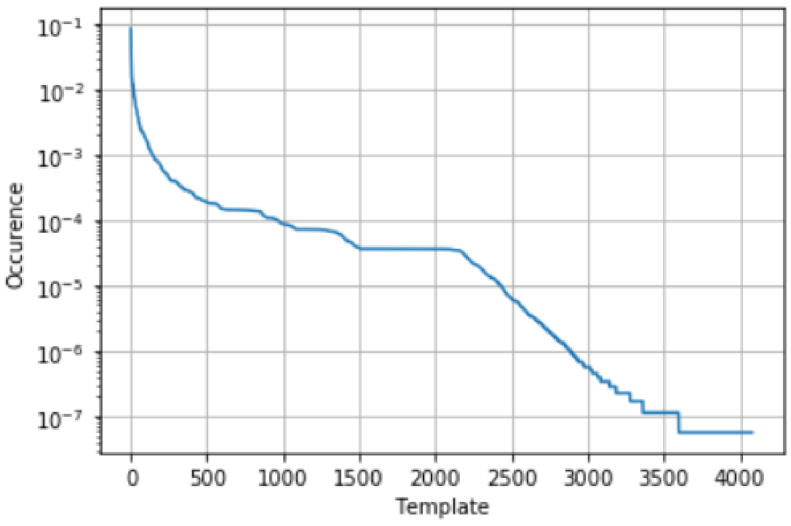

Figure 4.10: Distribution of events in training data, sorted based on occurrence. Logarithmic scale is used.

Parameter tuning of the RNN was done on a slightly reduced subset of the training data, where we used the cross entropy loss of the validation set as an indicator of how well the model learned to predict normal data. The model state was saved after each epoch that yielded a lower validation loss. Previous work suggests that increasing the size of the history sequence for which the RNN bases its predictions never hurts the results but only the time it takes for the model to learn. By increasing the window size, the predictions should only improve until a point where the model learns to disregard information from too far back in time [11]. As such we ended up using a window size of 25 as it showed very little improvements compared to 20,

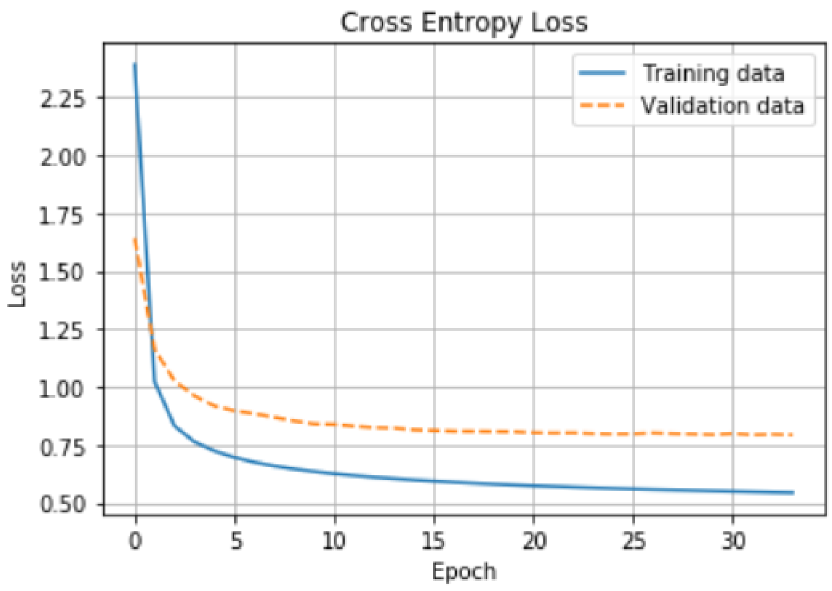

Figure 4.11: Training and evaluation losses over 35 epoches on the Ericsson dataset.

but still did not decrease the time performance by a lot.

The best results were achieved using a hidden size of 64 and three LSTM layers, for which we then trained the model on all training data. The cross entropy losses for the training and validation set can be seen in fig 4.11. Topk was the only parameter tuned with respect to the test set. Running with these parameters achieved a best F-measure of 0.11 at topk 1, which is very bad. The scores of other topk can be seen in Table 4.5. Notable is that with topk 1, the recall still is very low, at 0.22. This means that there are many anomalies which the model predicts with highest probability, which implies that the anomalies are also common in the training data.

Table 4.5: Precision, Recall and F-measure for different values of the parameter *topk* on the Ericsson dataset.

topk	Precision	Recall	F-measure
1	0.07783	0.22188	0.11524
10	0.07439	0.03615	0.04865
100	0.25456	0.02241	0.0412
1000	0.55019	0.0176	0.03411

When using the probability threshold instead, with the same model, it managed to get an F-measure of 0.23 which is much better. It is mainly the recall that drags the score up, but the precision increased slightly as well. However, the score was achieved with a threshold of 0.9, meaning that the model needs to be 90% sure that the next event is the one observed for it to be normal. A threshold that high is more strict than a topk of 1, since there cannot be two events with a probability of 0.9 as the probabilities of all events sum to 1. So not only must the event be the

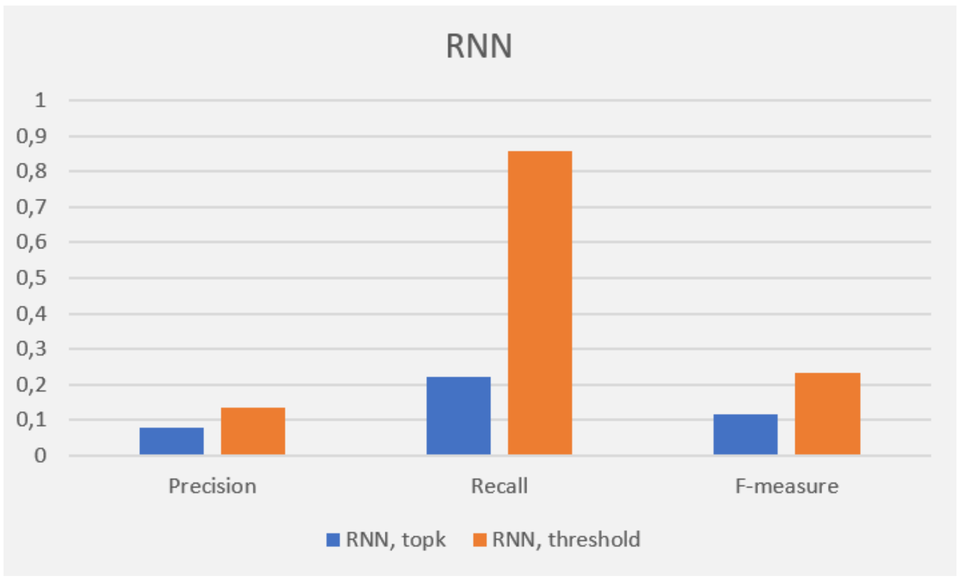

Figure 4.12: Precision, recall and F-measure scores on the Ericsson dataset by the RNN models.

top most predicted, it also needs to be very certain that the event is correct. Even though the F-measure doubled when using the probability threshold, it is still far from being useful in real world anomaly detection.
When running the model on the test set, anomalies were found in both passed and failed tests. Though more anomalies on average were detected in the failed test cases, the variance were larger within each category than between them. This strongly suggests that the data is very dissimilar between individual tests. The final score of the model can be seen in Figure 4.12. Confusion matrices for all models on the Ericsson dataset can be seen in Figure 4.13.

	P'	N'
P	8,025	15,221
N	7,289	244,118

(a) Subspace model

	P'	N'
P	3,732	13,088
N	44,217	213,466

(b) RNN, topk

	P'	N'
P	13,463	3,357
N	84,655	173,028

(c) RNN, threshold

Figure 4.13: Confusion matrices for the different models on the Ericsson dataset. Rows corresponds true labels and columns predicted labels. Note that there is a discrepancy between the subspace model and the RNN models due to the differences in vectorization and labeling.

5 Conclusion

5.1 Discussion

5.1.1 The data

The Ericsson dataset is very unstructured and a major problem is that there are many occurrences of log messages that stretch over multiple lines. This could be memory stack dumps or even log messages that has been broken so that parts of it appears on the next line. This can introduce problems with the parsing of templates as one event will be split into several parts and create inaccurate templates. There are also messages where words have not been printed correctly, for example there are several messages where the word CONNECTING is printed as NNECTING, which leads to similar problems where the correct message and the broken message get two different templates. To mitigate the problem with multiline messages, newlines were removed from suspected multiline messages. However, this required the use of domain knowledge (the observation that all messages start with a date) to find the suspected messages, which goes against the thesis of finding a general method for anomaly detection.

Many systems today are designed to take care of issues in a safe way without it causing any problems. These systems are known as fault tolerant systems [29] and could produce logs that look anomalous, but never actually caused any errors. This could for example be a client trying to connect to a host and fail, but then tries again and succeed. A log message related to failing to connect here would not be considered anomalous, however if it tries more times and fails it could be an indication of an anomaly. This is apparent in the Ericsson dataset, as there are many false positives produced by the models, which could be anomalies such as warnings and fails, but according to domain experts are normal for example during startup and shutdown phases.

5.1.2 The parsing

Since the amount of templates had to be reduced heavily in order to reduce the dimensionality of the problem, one could argue that a lot of important information was lost from the templates. From table 4.1 it can be seen that the number of templates was reduced by over 99%. However, the baseline measure in the table should probably not be viewed as baseline, but rather that the number of templates parsed when parsing all logs simultaneously should be considered baseline. This is due to the fact that the only difference between the two is that the algorithm has more data to work with when it parses all logs simultaneously which should lead to more accurate clustering. The fact that there is such a great difference between them is likely due to inaccurate parsing in the baseline case, such that some templates receive slightly different templates in each log file which leads to many templates in the database being slightly different when they should be the same.

The idea to collapse several variables into one is something worth discussing whether it is good or not. The reason behind this idea was that some messages contain lists of variables but originate from the same print statement. We have concluded two issues with this method. Firstly, as IPLoM starts by clustering based on length of the message, two such messages will be clustered into different events even from the beginning, and there is a high risk that they will not be collapsed into the same template anyways. Another problem is that since variable lists could represent any amount of following variables, they would sometimes create very generic templates which could match many other templates. We believe this to be especially problematic in the parse tree used in the prediction phase. When comparing the database creation mode and the prediction mode on the same data, we could see that a small fraction of messages were parsed into different templates. The absolute worst case we discovered was a template which was only a single string token followed by a `<*>`. In the prediction mode, any message that starts with the same string token would be captured by this template despite there being templates that would match it better. This is due to the parser naively searches the tree for the first match, which is not necessarily the best match. Searching the whole tree for the best match would increase time complexity drastically. If variables would not be collapsed, messages would at least have to match length.

5.1.3 The models

Though the RNN performed really well on the public datasets, it showed very poor performance on the Ericsson dataset. We suspect that the order of messages are too arbitrary due to the several components and parallel execution, and that this is a too difficult setting for the model to learn sequential relationships properly.

The fact that the RNN does not find very many unseen events in the test data of the Ericsson dataset and that many of the anomalies actually are the top predicted events might suggest that anomalies are also present in the training data. This would also explain why the subspace model works better as anomalous data is accounted for in the fit phase. The anomalies in the training data is likely due to how the test system the logs were extracted from works. It first has a startup phase, then it runs the test and then it has some exit phase. In one of the files that had passed the test, there were many true anomalies marked by the domain experts in the startup phase, but since this is not part of the test it still got a pass.

By using only a window size of 10 for the subspace model, it is very doubtful that it manages to capture any mayor relationships between the events, especially if we compare to the results from running on BGL. Yet it provides a better score than the RNN. We assume that the presence of a few events contribute very much to the projection error, such that it essentially performs anomaly detection mostly on an event level. If this is the case, it would be interesting to try an anomaly detection method that only focuses on event clustering, however this would defeat the purpose of this thesis.

5.1.4 Threats to validity

Here we briefly state a few reasons that can affect the validity of the results.

- The methods implemented were only tested on two open datasets, which is not enough to make any specific claims on performance on other datasets.
- For the Ericsson data, the test set was very small consisting of only 10 log files, containing approximately 275000 log lines and 16,820 rows marked as anomalous. This is not a sufficient amount to give a reliable result of the performance, especially considering the dissimilarity between each file.
- Though the results look promising for our parsing method, this has not been individually evaluated against the "true" templates and we do not know how well our events relates to true the logging statements.
- Since the logs are labeled manually in an active learning manner, there is a risk that there are mislabels. It becomes increasingly difficult to verify as our methods only mark anomalous segments in the logs and does not provide additional information about what went wrong. To get more accurate labels, more experts would have to inspect the files and thoroughly inspect every line, and not use the results of the models as indicators as this could introduce a bias.

5.1.5 Future work

A suggestion on future work could be to try and find other means of grouping messages together, even if explicit identifiers are not present. One could for example try groupings based on arbitrary variables parsed out in the parsing process, such as for example IP-addresses or other variables that can occur for different templates. For the Ericsson dataset, this could not be done for all messages as most messages do not share any suitable parameters, however it could perhaps be combined with a sliding window approach to provide additional analysis to certain parts of the logs.

There is also a need for more robust log parsing that can handle things like broken lines or words. These things seem to be quite common in the Ericsson dataset and thus probably also common in many other datasets. These problems make the parsing more unpredictable and the complexity of the model increases since there are more templates. For example, the parse tree could be modified so that if a message matches part of a template, it could see if the next line matches the continuation of the template. Broken words could be detected by trying to match the word to the template and see if it could be a substring of a word in the template.

Another thing to test could be to combine different models and do several runs on the data with the different models. The idea is that if the results are combined in some clever way, they could together produce better results. Different models seems to work well on different datasets (mostly notable on the BGL dataset), which suggests that if the results are combined they could draw from each others strengths.

5.2 Conclusion

We have approached the task of execution path anomaly detection on a new system log dataset composed at Ericsson. These logs are composed by four primary states (i.e they are not uniform throughout the files), and there are several processing nodes writing to the same file concurrently. Previous studies on anomaly detection on distributed logging has to a great extent been done on the assumption that session identifiers are present, and can be used to aggregate log messages belonging together, something that is not the case for our dataset. Additional discrepancies to our dataset are the setup of multiple different files and a severe lack in uniformity between log messages. We have attacked this problem by clustering log messages into a finite set of events, where we apply heavy pruning rules to the clustering process to reduce the number of events, and we have evaluated two different anomaly detection methods applied in an sequential manner over the logs. One of the models is a subspace model based on Principal Component Analysis and captures quantitative relationships between events, and the other is based on a Recurrent Neural Network which captures sequential relationships. Our setup has been evaluated on our dataset as well as the two public datasets HDFS and BGL. Overall, the results are very poor for our dataset, and neither model achieved a result good enough to be used in practice.

We can conclude that for datasets where logging templates are not well standardized, an alternative pruning method could be to remove tokens containing digits as these seem to often relate to variables. While this reduces the number of templates extracted massively, and the positional information of these tokens are lost, we still managed to cluster messages accurately enough to produce equivalent results to prior studies with our models on both HDFS and BGL. While we cannot make any major conclusions based on the results on the Ericsson dataset, we deem this approach a necessary step in the process of parsing our logs, and we think that it can be useful in other scenarios as well.

Given the results achieved for HDFS and BGL, we saw that an RNN method tends to produce better results. For HDFS we conclude that this is because of the well structured format of isolated session pose, as a normal execution can be easily learned. For BGL, all anomalies comprise of unique events never seen in the training data, and as such the RNN can be tuned to be very selective in what it regards anomalous. Additionally, we can conclude that using a probability threshold instead of a fixed topk used in previous literature improves the accuracy of the RNN model even further.

Unfortunately, neither of the models tested are feasible for finding anomalies in the Ericsson data, and we question whether any model is. Due to the findings in our test set, we have concluded that most causes to why a test fails does not show up as statistical outliers in our data. There is a difference between a statistical outlier and an actual cause of error since just because a sequence of events is very rare, doesn't mean that it automatically is a cause of failure. We don't think that a more complex model will work better, but that the key to anomaly detection is in the preprocessing as the problems seems to be inherent in the data. Since the distribution of events is skewed, as seen in Figure 4.10, there are some events that occur very rarely. These will be hard for the model to predict as they are not shown to the model frequently during training and risks being labeled as anomalous.

We can conclude that an RNN in the way it was implemented in this thesis does not work well on the Ericsson dataset due to the fact that there are anomalies present in the data from successful test cases which we have regarded normal. The subspace model produces somewhat better results, perhaps since it operates on both normal and anomalous data in the training phase, and does not suffer from being exposed to anomalies such as the RNN. The best results by the subspace model was however achieved on a very small window size, arguably not large enough to capture many real relationships between the events.

Our final conclusion is that fully automatic anomaly detection on logs is very difficult using current methods. It is very reliant on preprocessing and exploitation of specific properties of the individual datasets. We hope however that this thesis can provide some useful insights for future studies on the topic.

Bibliography

[1] Adam Oliner, Archana Ganapathi, and Wei Xu. Advances and challenges in log analysis. *Communications of the ACM*, 55(2):55–61, 2012.

[2] Shilin He, Jieming Zhu, Pinjia He, and Michael R Lyu. Experience report: System log analysis for anomaly detection. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, pages 207–218. IEEE, 2016.

[3] Adam Oliner and Jon Stearley. What supercomputers say: A study of five system logs. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*, pages 575–584. IEEE, 2007.

[4] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I Jordan. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 117–132, 2009.

[5] Jieming Zhu, Shilin He, Jinyang Liu, Pinjia He, Qi Xie, Zibin Zheng, and Michael R Lyu. Tools and benchmarks for automated log parsing. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 121–130. IEEE, 2019.

[6] Pinjia He, Jieming Zhu, Shilin He, Jian Li, and Michael R Lyu. An evaluation study on log parsing and its use in log mining. In *2016 46th annual IEEE/IFIP international conference on dependable systems and networks (DSN)*, pages 654–661. IEEE, 2016.

[7] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. 2004.

[8] Jian-Guang Lou, Qiang Fu, Shengqi Yang, Ye Xu, and Jiang Li. Mining invariants from console logs for system problem detection. In *USENIX Annual Technical Conference*, pages 1–14, 2010.

[9] Top500 supercomputing site. Highlights - june 2006. https://www.top500.org/lists/top500/2006/06/highlights/. (accessed: 2020-05-11).

[10] Jiaqi Tan, Xinghao Pan, Soila Kavulya, Rajeev Gandhi, and Priya Narasimhan. Salsa: Analyzing logs as state machines. *WASL*, 8:6–6, 2008.

[11] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1285–1298, 2017.

[12] Weibin Meng, Ying Liu, Yichen Zhu, Shenglin Zhang, Dan Pei, Yuqing Liu, Yihao Chen, Ruizhi Zhang, Shimin Tao, Pei Sun, et al. Loganomaly: Un-

supervised detection of sequential and quantitative anomalies in unstructured logs. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19. International Joint Conferences on Artificial Intelligence Organization*, volume 7, pages 4739–4745, 2019.

[13] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[14] Christophe Bertero, Matthieu Roy, Carla Sauvanaud, and Gilles Trédan. Experience report: Log mining using natural language processing and application to anomaly detection. In *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, pages 351–360. IEEE, 2017.

[15] Adetokunbo AO Makanju, A Nur Zincir-Heywood, and Evangelos E Milios. Clustering event logs using iterative partitioning. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1255–1264, 2009.

[16] Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R Lyu. Drain: An online log parsing approach with fixed depth tree. In *2017 IEEE International Conference on Web Services (ICWS)*, pages 33–40. IEEE, 2017.

[17] Qingwei Lin, Hongyu Zhang, Jian-Guang Lou, Yu Zhang, and Xuewei Chen. Log clustering based problem identification for online service systems. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 102–111. IEEE, 2016.

[18] Z. Liu, T. Qin, X. Guan, H. Jiang, and C. Wang. An integrated method for anomaly detection from massive system logs. *IEEE Access*, 6:30602–30611, 2018.

[19] Ricardo Dunia and S Joe Qin. Multi-dimensional fault diagnosis using a subspace approach. In *American Control Conference*, 1997.

[20] Karl Pearson. Liii. on lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 2(11):559–572, 1901.

[21] Svante Wold, Kim Esbensen, and Paul Geladi. Principal component analysis. *Chemometrics and intelligent laboratory systems*, 2(1-3):37–52, 1987.

[22] Xiao Yu, Pallavi Joshi, Jianwu Xu, Guoliang Jin, Hui Zhang, and Guofei Jiang. Cloudseer: Workflow monitoring of cloud infrastructures via interleaved logs. *ACM SIGARCH Computer Architecture News*, 44(2):489–502, 2016.

[23] Ronald J Williams and David Zipser. Gradient-based learning algorithms for recurrent. *Backpropagation: Theory, architectures, and applications*, 433, 1995.

[24] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[25] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with lstm. 1999.

[26] Volker Märgner and Haikal El Abed. Icdar 2009 arabic handwriting recognition competition. In *2009 10th International Conference on Document Analysis and Recognition*, pages 1383–1387. IEEE, 2009.

[27] Kanishka Rao Françoise Beaufays Haşim Sak, Andrew Senior and Johan Schalkwyk. Google voice search: faster and more accurate.

https://ai.googleblog.com/2015/09/google-voice-search-faster-and-more.html, 2015-09-24. (accessed: 2020-05-07).

[28] Burr Settles. Active learning literature survey. Computer Sciences Technical Report 1648, University of Wisconsin–Madison, 2009.

[29] A Aviziens. Fault-tolerant systems. *IEEE Transactions on Computers*, 100(12):1304–1312, 1976.