AI-assisted anomaly detection from log data

Master of Science in Technology Thesis
University of Turku
Department of Computing
Software Engineering
2023
Teemu Pusa

UNIVERSITY OF TURKU
Department of Computing

Teemu Pusa: AI-assisted anomaly detection from log data

Master of Science in Technology Thesis, 99 p.
Software Engineering
October 2023

As the production of software continues to increase, the volume of log data being generated is also on the rise. This surge in data has made it impractical for human operators to manually review each log line produced by software systems. This necessity has led to the development of automatic anomaly detection methods. Automatic anomaly detection methods would allow system operators to respond to incidents more quickly and improve the quality of the software.
In the past, anomaly detection from log data relied heavily on predefined rules. However, with the complexity of modern software systems, finding experts for every system component to write these rules has become difficult. Additionally, it is very labor-intensive to implement these rules. This has spurred interest in unsupervised anomaly detection methods.
The purpose of this thesis is to research which kind of methods can be used for automatic anomaly detection, what is required to use them in a production system, and how well deep learning-based methods would work with log data produced by hundreds of embedded devices. The thesis begins with a literature review to explore the various methods used for anomaly detection from log data. It then outlines the required infrastructure for efficient anomaly detection and concludes by testing the DeepLog Deep Learning method on real log data from a production system.
The key findings suggest that the DeepLog model performs effectively for anomaly detection when trained in an unsupervised fashion. However, it is essential to ensure that anomalous samples do not dominate the training data. This can be achieved either by completely excluding them from the training set or by ensuring that no single anomalous sample overwhelms the entire dataset, which could lead to overfitting. Moreover, the training dataset can be continuously refined by eliminating recognized anomalous sequences and subsequently retraining the model.

Keywords: Anomaly detection, Deep learning, DeepLog, BERT, NLP, PCA

Contents

List of Figures

List of Tables

1 Introduction

Nowadays almost all software systems produce somekind of logdata which can be used to determine what kind of actions the software system was doing while it was running. Exceptions for this could be some lowlevel embedded devices which do not have resources and hardware capabilities to produce logdata. This logdata can be used to diagnoze system errors.

Tradionally logdata analysis has been human labour intensive work but with the increased amount of software systems and logdata they produce it is not possible for humans to examine all of the logdata to find errors. There has been incresing amount of efforts to use automated methods to detect anomalies from the logdata.

First loganomaly detection methods used data mining and data analysis methods to used datamining and analysis methods to find anomalies in the logs but more recently the focus has shifted to more machine learning and with the rise of deep learning success in other fields there has been many efforts of using deep learning for automatic loganomaly detection.

Even though deep learning methods produce superior results compared to more tradional data analysis based methods they also tend to have a higher computational requirements. Depending on size of the organisation it may make deep learning inpractical to be used in those organisations. Deep learning methods also generally require some separate training step with labelled data. Supervised Deep Learning methods may require a lot of human labour for labeling the training dataset which can be constly but if the training objective can be transformed to self supervised style of learning it can greatly decrease the required human effort for training. Generally exchanging human worktime to machine worktime is profitable for the business.

In order for the company to do automatic log anomaly detection effectively it requires them to have some kind of pipeline which considers different aspects of loganomaly detection like how to collect the logdata, how to do required preprossessing, how to train the models and how to visualise the results so that human operators can underestand.

This thesis seeks to answer following research questions:

- **RQ1:** What kind of methods are currectly being used for log anomaly detection.
- **RQ2:** What kind of infrastructure does machine learning methods need to be used for anomaly detection.
- **RQ3:** What kind of methods are economically viable to use in a small company.
- **RQ4:** How to visualise anomalies found from the system so that the human maintainer can investigate the anomaly efficiently.
- **RQ5:** Does automatic log anomaly detection decrease the overall worktime required for system maintenance.

The main objective of this thesis is to evaluate whether the logdata produced by the company's software system can be used to increase system reliability by finding anomalies anomalies earlier and whether these methods will reduce the human effort required for anomaly detection and anomaly investigation.

The rest of the thesis is structured as follows:

Chapter 2 provides reader basic idea on what is log anomaly detection and what concepts and terms are used when discussing about automatic loganomaly detection.

Chapter 3 provides more detailed explanation for some concepts choosen from literature.

Chapter 4 describes more what kind of system the logdata is collected from and gives brief plan on how the practical side of this thesis is executed.

Chapter 5 provides description what kind of pipeline should be build in order to do loganomaly detection effectively.

Chapter 6 provides description of the dataset structure and how testing was concluded with some realworld logdata and anomalies.

Chapter 7 analyses the results of the experiments and how well the methods would work in realworld usecases.

Chapter 8 provides the final conclusions and discussions about the results of the practical sections. The research questions are answered in this chapter.

In this thesis, AI tools such as GPT-4 were employed solely for the purpose of enhancing the flow of the text, and each suggestion was rigorously examined by the author.

2 Log Anomaly Detection

Anomalies within software systems often manifest as irregularities in log data. By investigating logs, one can discover the root causes of many issues. For instance, software running as a backend service may log every HTTP request and its corresponding payload. Such logs offer invaluable insights during post-issue investigations. However, this reactive approach means that users might already have experienced the problem by the time it's diagnosed. In the worst scenarios, users, discouraged by anomalies, might abandon the service entirely. Without efficient log analysis, such problems could frequently recur, diminishing user trust.

To alleviate this, there's an increasing interest in automated anomaly detection techniques. These tools aim to identify irregularities without necessitating manual log examination. A correctly identified anomaly is termed a 'true positive'. Conversely, if the system flags a non-existent issue, it's deemed a 'false positive'. Instances where actual anomalies are missed by the system are known as 'false negatives'. An ideal system should adapt over time, minimizing erroneous predictions.

Different applications might prioritize different aspects of detection. While some might tolerate false negatives to reduce false positive disturbances, critical systems, such as those in healthcare, demand high accuracy due to the stakes involved.

2.1 Types of Anomaly Detection Systems

2.1.1 Rule-Based Systems

The most rudimentary anomaly detectors employ Regex filters or intricate rule systems. Although potentially effective, they demand consistent maintenance and domain expertise for rule formulation. Such systems excel at identifying known anomalies but fail when faced with unforeseen disturbances.

2.1.2 Data Analysis and Clustering

Focusing on deciphering log patterns, these techniques aim for anomaly detection even without explicit domain-specific knowledge. They first necessitate the transformation of raw log data into a comprehensible format. Subsequently, data science methods analyze the processed logs. While these approaches can discern simple patterns, they may struggle with intricate log sequences, especially those exhibiting parallel activities.

2.1.3 Deep Learning

Deep learning, a relatively recent entrant in the field, captures more complex log patterns, improving anomaly detection efficacy. Depending on the data preparation, deep learning techniques can be categorized into:

Supervised Learning

For this approach, human annotators label data segments, designating them as either ’normal’ or ’anomalous’. While these methods can deliver impressive results, labeling vast datasets for deep learning training is labor-intensive and costly. Consequently, techniques like data augmentation, commonly used in domains like image recognition, are explored to circumvent the data scarcity challenge.

Unsupervised Learning

Removing the need for human-annotated data, unsupervised learning solely relies on the inherent data patterns. This eliminates the labor and cost associated with manual annotation. However, defining suitable learning objectives can be challenging. For log anomaly detection, it’s often assumed that a majority of system-generated logs are ’normal’. Leveraging this presumption, deep learning models can be trained to predict how normal log sequences look like. Upon training, the model is trained to predict normal log structure and even if the training data includes anomalies it should not affect models normal prediction accuracy too much. During detection phase model can be used to predict normal log structure and then by comparing it with the ground truth determine whether it is anomalous or not.

2.2 Concepts

2.2.1 Parsing

Within the realm of log data anomaly detection, "parsing" typically refers to the processing of raw log data to make it suitable for use in an anomaly detection system. Rule-based systems may not necessitate any preprocessing, as they operate directly on the raw log data. However, if the log data possesses any inherent structure, such as log line timestamps, preprocessing can be beneficial.

Methods leveraging machine learning demand that the data is presented in a numerical format. How the raw log data is numerically represented hinges on the specific method employed. The most straightforward approach would involve treating each log line as a unique number. However, this could yield an excessive number of distinct numbers for every possible log line permutation. For instance, given a log line like "sleep for 30 seconds," this method would generate a new number for each distinct sleep duration. Even more problematic, these numbers lack semantic meaning. Consequently, two nearly identical log lines might differ only in their parameter values, but the parser fails to indicate any similarity between them.

To address this issue, parsers like Spell [1] and Drain [2] attempt to identify the longest common subsequences to determine which segments of the log line represent the log template and which denote parameters. Ideally, for the previously mentioned example, the parser would produce: "sleep for * seconds," with the asterisk symbolizing a variable parameter value. Nonetheless, such an approach does have a downside: semantically similar log lines could yield different log templates. Potential solutions include employing semantic word vectors, as suggested by HitAnomaly [3], or relying on tokenization while allowing the model to discern the log data's structure, as proposed by LanoBERT [4].

Prior to parsing, some methods implement preprocessing to eliminate elements like IP addresses, which might otherwise result in too many of log templates.

2.2.2 F-score

The F-score is a prevalent metric for evaluating the efficacy of binary classification models. It represents the harmonic mean of precision and recall. The optimal F-score is 1.0, signifying flawless model predictions, while the lowest attainable score is 0, indicative of either zero precision or recall. Precision and recall are derived from the counts of true positives (TP), false positives (FP), and false negatives (FN). The F-score is defined as:

$$F_1 = \frac{2 \times \text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$$

Where:

$$\text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

and

$$\text{recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

One inherent limitation of the F-score is that it affords equal weighting to both FP and FN. This can be problematic, as different systems might have varying tolerances or requirements regarding these metrics.

3 Analysis of Detection Methods

This chapter will provide an overview of technologies that have been presented in the literature for anomaly detection, giving the reader an idea of the methods that have been used in the past and the current state-of-the-art methods. The chapter is organized in chronological order, with a focus on the development of new technologies over time. By examining the evolution of anomaly detection technologies, a deeper understanding of the current state of the field can be gained, and areas for further research and development can be identified.

3.1 Swatch

Introduced by Hansen et al. [5], Swatch represents one of the earliest methods employing console logs for automated anomaly detection. Developed as a command-line tool, Swatch is designed for finding issues manifested in console logs. Real-time log scanning can be achieved through the "swatch -t /var/log/authlog" command, while analysis of an entire log file is facilitated by "swatch -f /var/log/syslog.0". Implementation of Swatch was performed using the Perl programming language.

Swatch's operation is guided by a configuration file containing instructions for both matching log lines and specifying subsequent actions. Regular expressions are employed to match each log line against the patterns found in the configuration file. Upon a successful match, actions configured in the configuration file are executed. A time interval instruction is also included in the configuration file, aimed at minimizing redundant notifications. For instance, a five-minute interval can be set to significantly reduce repetitive alerts when log lines are generated at a high frequency.

Several actions are recognized by Swatch upon matching a log line, which include:

- The **echo** action, used for relaying the line to Swatch's controlling terminal.
- The **bell** action, responsible for sending a bell signal (Ĝ) to the controlling terminal.
- The **ignore** action, directing Swatch to bypass the current input line.
- The **write** and **mail** actions, which enable the transmission of the line to a designated user list via write and mail commands, respectively.
- The **pipe** and **exec** actions, offering additional flexibility. The pipe action permits the utilization of matched lines as command input, whereas the exec action facilitates the execution of system commands, optionally using selected fields from the matched line as arguments.

It is noted in the paper that Swatch has proved beneficial in detecting system probing by intruders, which could have otherwise gone unnoticed in the absence of centralized logging and Swatch. Experiences of averting system failures during off-hours due to air conditioning malfunctions are also reported. As experience with Swatch's configuration file has grown, its utility has correspondingly increased.

3.2 Logsurfer

Logsurfer, as presented in [6], is designed for the examination of messages in a log file in relation to other messages. Phenomena in the log file that can be uniquely attributed to specific issues are termed as "signatures." The capability to make use of contextual information enables log analysis programs to identify and act upon a greater number of signatures. With its dynamic rule set and the ability to compile related messages into structures called "contexts," Logsurfer effectively identifies these signatures.

Logsurfer comprises four main components: messages, rules, actions, and contexts.

- **Messages** are defined as units of incoming data from a single data source.
- **Rules** serve to identify messages and initiate actions based on those identifications.
- **Actions** are initiated either when a rule identifies a message or when a context is terminated.
- **Contexts** are structures that accumulate and store messages in memory.

3.2.1 Rules

In Logsurfer, a rule is composed of six or seven fields. The first field is defined by a regular expression that identifies the messages to be matched by the rule. The second field also uses a regular expression to specify exceptions to the first field, meaning that messages are excluded from matching if they align with this expression.

The third field contains a regular expression that indicates the conditions under which the rule should be removed. Similarly, the fourth field uses a regular expression to state exceptions to the third field, specifying situations where the rule should not be deleted.

The duration for which the rule remains active is indicated by the fifth field, expressed in seconds. A value of 0 signifies that the rule should remain active indefinitely.

An optional sixth field may include the keyword "continue." This implies that when a message matches the rule, the following rules in the sequence should also be evaluated against the message. In the absence of this field, messages matching the rule are not subject to further rule checks.

The action to be executed when the rule matches a message is dictated by the seventh field.

3.2.2 Actions

In Logsurfer, actions are defined to specify what should be executed when a particular message or a context of messages is encountered. An action is triggered either when a rule matches a message or when a context surpasses one of its defined limits. Typically, the action results in the execution of an external program, taking the specific message or context of messages as its input.

The actions supported by Logsurfer are listed as follows:

- **Ignore**: The message is disregarded. This action is commonly employed to filter out messages that do not require any action.
- **Exec**: An external program is executed.
- **Pipe**: An external program is executed, with the existing message or context serving as its standard input.
- **Open**: A new context is opened. No action is taken if the context already exists.
- **Delete**: A context is deleted. The specific context for deletion is identified by the match regular expression of the context.
- **Rule**: A new rule is created. The first argument indicates the position within the ruleset where the new rule should be inserted and must be one of: before,

top, or bottom.

3.2.3 Contexts

Contextual information is crucial when analyzing a log file. A single symptom may not provide enough information to determine whether there is a problem. Messages in a log file may be related if they come from the same process on the same host, or they may have more complex relationships.

Contexts collect all the relevant data for diagnosing a particular problem into one location, which is a structure stored in memory for reporting later if necessary. By grouping related messages together, contexts provide a better understanding of the problem and help in identifying the root cause of the issue. They can also help in tracking the progress of a problem and provide a history of events leading up to it. Therefore, contexts are essential for effective log analysis.

3.3 Pattern Mining from Event Logs

A critical role in system management is played by event logging and log files, which serve as key sources of information regarding the health of the system. Tools like Swatch [5] and Logsurfer [6] have been developed for log file monitoring.

Monitoring techniques are generally divided into two categories: fault detection and anomaly detection. In fault detection, a database of fault message patterns is created by a domain expert. A limitation of this approach is its inability to detect faults unknown to the expert. Thus, messages corresponding to new fault conditions in the log file may be overlooked. Moreover, finding a person with sufficient knowledge of the system can be difficult.

In contrast, anomaly detection involves the creation of a system profile that outlines normal system activity. Alarms are triggered by messages that deviate from this profile. Although this method can identify new fault conditions, manual profile creation is laborious and susceptible to errors.

A common format is often missing in log files, complicating their use for monitoring. Although certain elements like timestamps or device numbers are usually present, the plain text part of the message may not adhere to strict requirements. Event types are also frequently missing from log lines. However, event types can be deduced from the log files since similar events typically follow a particular line pattern. For instance, the following lines:

```
Router myrouter1 interface 192.168.13.1 down
Router myrouter2 interface 10.10.10.12 down
Router myrouter5 interface 192.168.22.5 down
```

are indicative of the "interface down" event type and correspond to the line pattern "Router * interface * down". Manual identification of these patterns is feasible only for small log files [7].

Clustering algorithms offer a solution to these challenges by grouping similar entries while separating dissimilar ones. Outliers, or those entries that don't fit into any cluster, are considered for further investigation. Upon identifying clusters, association rule algorithms can be employed to find temporal relationships between event types. Clustering also helps in recognizing line patterns typical of normal system activity, which can be added to the system profile. Outliers formed by the algorithm may indicate unknown fault conditions or unexpected behavior, warranting further examination to ascertain their root causes and ensure system reliability [7].

3.4 Source Code Analysis

In the preceding section, the conversion of unstructured log data into a structured format through clustering algorithms was discussed. However, an alternative method was proposed in [8], where logging statements in the source code were used for determining the types of log messages. The authors argued that, although logs may seem to be in free-text format, they are actually generated by a limited set of log printing statements embedded in the system. One benefit observed from using the source code as a schema for parsing is the ability to interpret all possible log messages, including those that are rarely encountered. Although this method necessitates the availability of source code for parsing logs, it was asserted that the prevalent use of open-source software in various systems makes this requirement more of a practical solution than a limitation.

3.4.1 Structure of logs

In the work presented in [8], it was observed that logs often exhibit a common structure, comprising a constant part (for instance, "xact") and multiple variable parts (such as "325/326" and "COMMITTING/ABORTING"). The constant part is referred to as the message type, and the variable parts are termed message variables by the authors. Through the analysis of logs from diverse systems and interviews with developers, two significant types of message variables were identified: identifiers and state variables.

Identifiers are variables that are utilized to distinguish the object being manipulated by the program. On the other hand, state variables are labels that list the potential states in which the object might exist (for instance, COMMITTING and ABORTING). With the identification of these types of variables, an algorithm was developed to automatically extract the schema of log messages. This schema can be employed for parsing and analyzing log data.

3.4.2 Problems

If the software was written with language like C it is likely that the template can be directly infered from the printf statements which generate the log messages. Example of an logline produced by a program made with C langauge could be fprintf(LOG, "starting: xact %d is %s") fprintf(LOG, "starting: xact %d is %s"). However when software system is developed with object-oriented langages such as Java it is more difficult. In Java it is common to use logging objects for logging so it can be difficult to figure out what object is producing the final log message. It is common to have some kind of interface like "toString" which is used for converting the object internal state to string which can be printed to the console. This means that it is not trivial to find out what kind of string is printed to console.

3.4.3 Solution

In the study presented in [8], a solution to the problems previously mentioned is proposed. The code is parsed into an Abstract Syntax Tree (AST) format and static analysis is performed to identify all method calls on objects of specific classes. Each identified call provides only a partial message template because the template might include interpolations of non-primitive types. To resolve this issue, all toString() calls in all classes are enumerated by the authors. The string formatting statements in these calls are then examined to deduce the types of variables in message templates. This type information is substituted back into the partial templates, and the process is recursively repeated until only primitive types are interpolated in all the templates.

3.4.4 PCA Anomaly Detection

In the paper [8] authors use PCA to find anomalies from logs. Their method has three parts. The first part is log parsing like was mentioned in previous section. The second part is feature creation and the final is the anomaly detection with PCA.

Feature creation

Two key features are utilized by the system: the state ratio vector and the message count vector, which are based on state variables and identifiers, respectively. The state ratio vector is formulated to encapsulate the system's behavior over a specified time frame, whereas the message count vector is designed to identify issues at the level of individual operations.

For the construction of a state ratio vector, state variables are grouped within a time window, and a vector is generated based on these variables. Each dimension in the vector corresponds to a unique state variable value, and the value within that dimension reflects the frequency of the state value within the time window. Inclusion in the vector is restricted to state variables that are reported a minimum of $0.2N$ times, where N represents the total number of messages. The size of the time window is automatically set such that each variable occurs a minimum of $10D$ times in 80% of all time windows, with D standing for the number of distinct values. It was found that varying the parameters for the state ratio vector's construction did not substantially impact detection outcomes.

A message count vector is created by categorizing messages based on the same identifier values and generating a vector for each group. Each dimension within the vector corresponds to a distinct message type, and the value in that dimension indicates the number of occurrences of that specific message type within the message group.

Anomaly detection

PCA (Principal Component Analysis) is employed to discern anomalous feature vectors that stray from the anticipated normal patterns. As depicted in Figure

3.1, the normative behavior is situated along the reference normal line. Elevation above this line indicates heightened system activity, while descent reflects reduced activity. Conversely, the red data points that diverge from this line signify instances of anomaly.

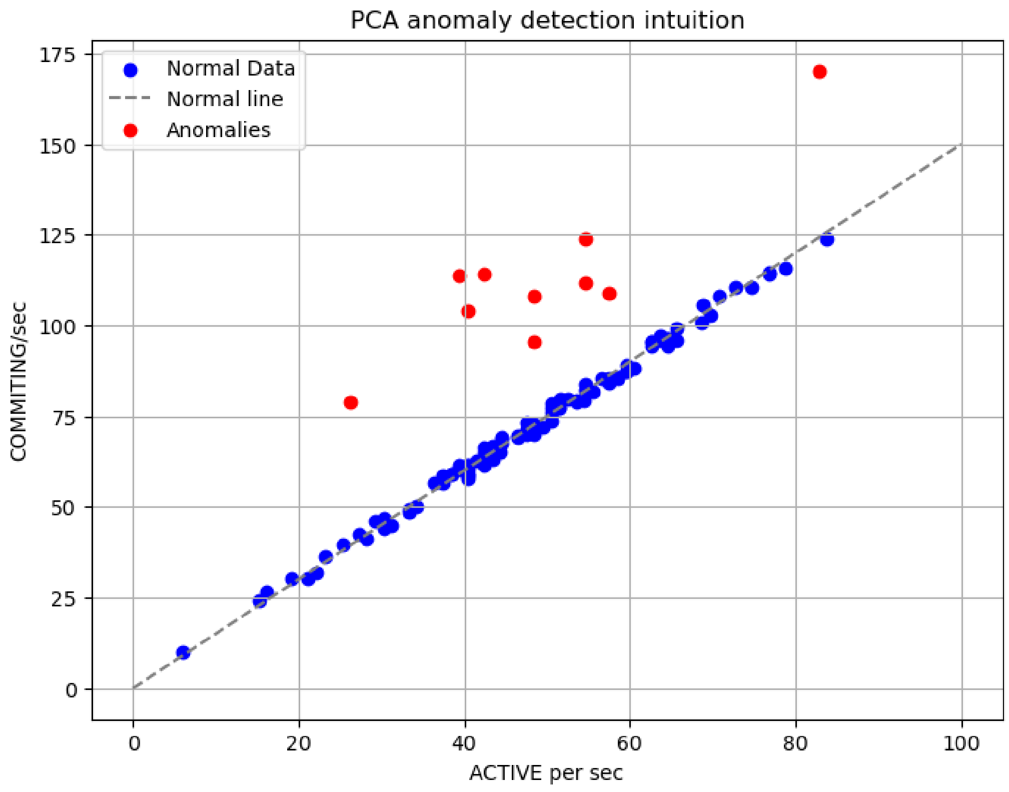

Figure 3.1: Intuition with PCA anomaly detection.

3.5 Unsupervised Streaming Parser

Untill this point the chapter has been going though methods which are being used for anomaly detection and usually they require separate parsing step. This parsing is done in offline fashion meaning it will happen after the logs has been collected. Many methods required either some kind of domain knowledge and regular expressions. Swatch [5] used simple regular expression based which simply match loglines and do actions defined in configuration file. Logsurfer [6] used more complicated configuration rules which could remove or add more rules and there was concept of context which means it can capture more complicated behaviour. Both of these methods required lot of effort for creating and maintaining the rules. Clustering method introduced in [7] used clustering approach to convert unstructured logdata to structured form. Method in [8] used source code as a schema for logs to parse them to structured format. Even though this method may require good accuracy it can also be inpractical in usecases where there is a lot of external dependecies which may even use different logging practices.

This section introduces unsupervised streaming parser which means it can parse logdata realtime even without any domain knowledge. However in some cases the performance of algorithm is better if some simple preprosessing is done like IP addresses or dates which usually have common format is easily detected using simple regular expressions without special domain knowledge.

3.5.1 Spell

Spell [1] is a structured Streaming Parser for Event Logs using an LCS (longest common subsequence) based approach. It parses unstructured log messages into structured format in streaming fashion. The goal with Spell parser is to find different message types produced by logging statements and separate them from parameters used in logging statements.

For instance, the log printing statement "printf('HTTP request took %d milliseconds', time)" can generate various kinds of log entries, such as "HTTP request took 500 milliseconds" or "HTTP request took 10,000 milliseconds." Here, the numbers 500 and 10,000 are considered parameters of the logging statement. Consequently, the message type would be "HTTP request took * milliseconds." This distinction is useful because a limited number of message types exist, and they can be employed to identify anomalies in the execution path. Additionally, parameter values such as

500 and 10,000 can be utilized to detect abnormal parameter values. Specifically, a 10,000-millisecond request time could be identified as an anomalously long request time.

Algorithm

The Spell algorithm consists of three parts. The first part is the LCSMap which contains all the LCSObjects. An LCSObject is comprised of two parts - LCSseq, which is the tokenized version of the logline, and lineIds, which have produced the LCSObject. Tokenization is done by splitting the log text using a delimiter such as a space or a colon. When a new logline arrives, it is first tokenized, and then the longest common sequence (LCS) is calculated between it and the existing LCSObjects in the LCSMap. The maximum value for LCS and its index position is saved to a variable. If the largest LCS value is greater than the threshold value, it means that the new logline and the existing LCSObject have the same message type. The algorithm then uses backtracking to set a mark "*" for every token that is different between the chosen LCSObject. If there are two consecutive adjacent "*", then they are merged into one "*". For example, if there are two loglines "Command Failed on: node-127" and "Command Failed on: node-235 node-236", then the LCSseq of the two would be: "Command Failed on: *". If none of the LCS values with other LCSObjects is higher than the threshold, then a new LCSObject is created with LCSseq as itself.

Performance

When a new logline arrives, the algorithm's primary task is to compute the length of its largest common sequence with each existing logline. However, based on their observations, over 99% of the new log entries are already present among the currently parsed message types. Consequently, instead of calculating the Longest Common

Subsequence (LCS) between the new logline and every existing message type, they implement a pre-filtering step to determine if the message type already exists. The new logline is denoted as σ, and the current loglines are represented as a set of strings: $strs = \{str1, str2, ..., str_m\}$. The goal is to identify the longest str_i for which $|LCS(\sigma, str_i)| \geq \frac{1}{2}|\sigma|$ holds. The authors propose two approaches to achieve this:

1. **Simple Loop Approach**

   This straightforward method involves iterating through the strings in $strs$. For each str_i, two pointers are maintained: p_i, which tracks the current position in str_i, and pt, which tracks the current position in σ. If the characters pointed to by p_i and pt match, both pointers are advanced. Otherwise, only the pt pointer is advanced. Once the pt pointer reaches the end of σ, the algorithm checks whether the p_i pointer has also reached the end of str_i. Strings in $strs$ that have a length less than $\frac{1}{2}|\sigma|$ are pruned.

2. **Prefix Tree Approach**

   In the prefix tree approach, a prefix tree (also known as a trie) is utilized to minimize the number of candidates that need to be examined. For instance, given $strs = \{ABC, ACD, AD, EF\}$ and $\sigma = ABPC$, the algorithm starts by comparing each character in σ with the characters in the tree. This process leads to the pruning of various branches from the tree. Characters that don't match those in the tree are marked as parameters. In the example depicted in Figure 3.2, ABC is identified as the message type for σ, and P is recognized as its parameter.

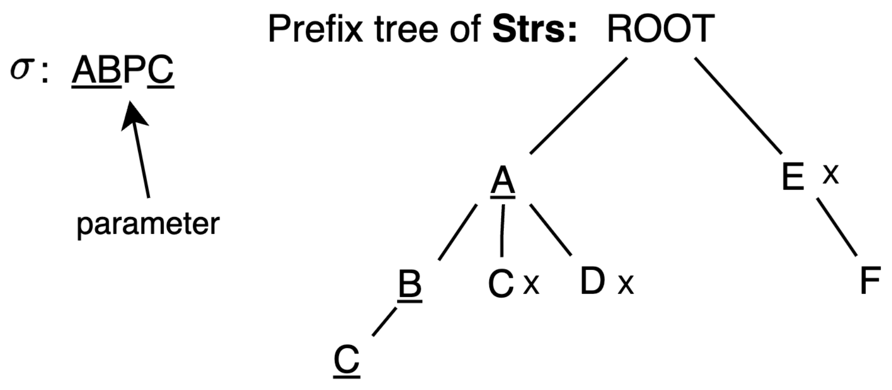

Figure 3.2: Finding subsequence of σ using Prefix Tree.

However, it's important to note that the prefix tree approach might not always return the longest common subsequence. For instance, if $\sigma = DAPBC$ and $strs = \{DA, ABC\}$, the prefix tree will identify DA as the message type instead of ABC. Nevertheless, in practical scenarios, the authors find that although this approach doesn't guarantee finding the longest message type, the returned message type is quite similar to the result obtained using the simple loop method. This phenomenon can be attributed to the fact that parameters within each log record typically appear towards the end.

3.5.2 Drain

The Drain parser, introduced by He et al. in their work [2], is a fixed-depth tree online parser designed for a similar purpose as the Spell parser discussed in the previous section. The motivation behind the development of the Drain parser stems from the challenges posed by modern large-scale service systems, which generate massive amounts of logs, often reaching 50GB per hour (equivalent to 120 200 million lines). Given that these systems are comprised of numerous components authored by hundreds of global developers, crafting parsing rules becomes a formidable task, as no single developer possesses a comprehensive understanding of the entire system's functioning. Moreover, the dynamic nature of internet systems demands parsing algorithms that can adapt to evolving log formats.

The Drain parser employs a fixed-depth parse tree structure, as depicted in Figure 3.3. In this example, the tree's depth is set to 3. The first layer encompasses nodes for log lines of varying lengths. Log line length is determined by the number of tokens produced through tokenization. The subsequent layer selects nodes based on the first token in the sequence. For instance, if the log line is "Receive from node 4," the corresponding node is "Receive." If the initial token is numeric, a special "*" node is designated to prevent excessive branching. The final layer of the tree contains log groups associated with the branch. A log group embodies patterns like "Receive from node *," where "*" serves as a variable. These log groups include log IDs that reference the line numbers producing the respective events.

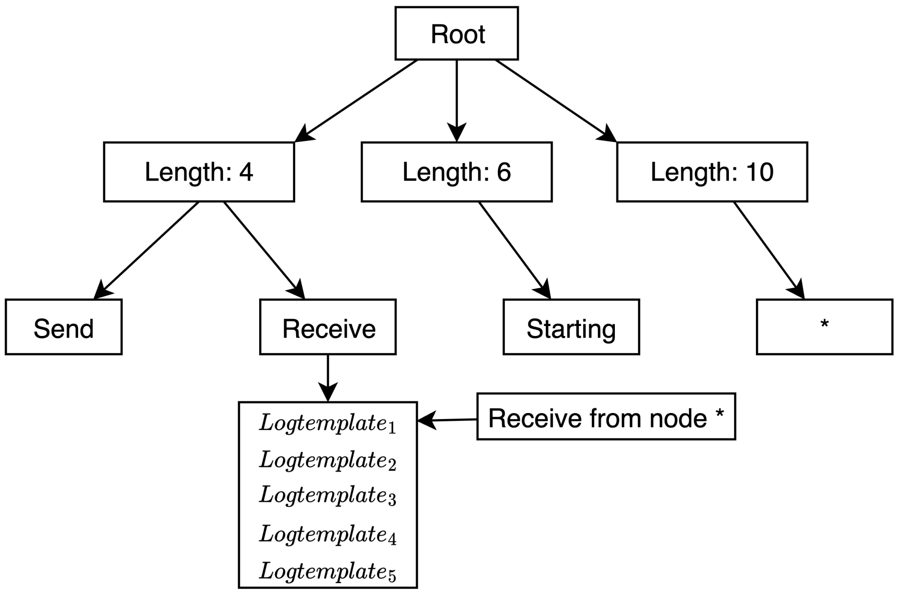

Figure 3.3: Structure of Parse Tree in Drain.

The Drain algorithm employs the simSeq algorithm to determine similarity between a given log line and others within the group. It subsequently identifies the most suitable matching log line, provided its simSeq value surpasses a predefined threshold. The simSeq metric is defined as follows:

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

Here, seq_1 and seq_2 represent log messages, $seq(i)$ denotes the i-th token in the sequence, and n signifies the sequence's log message length. The function equ is defined as:

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

Where t_1 and t_2 denote two tokens. The Drain algorithm identifies the log group with the highest simSeq value and compares it to a predefined similarity threshold, denoted as st. If simSeq $\geq st$, Drain designates the group as the most appropriate log group. Otherwise, a new log group is created. In cases where an existing log group is found, it is updated by comparing each token in the sequence. Tokens that do not match are replaced with "*" tokens like illustrated in Figure 3.4.

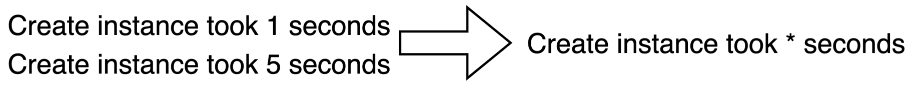
Create instance took 1 seconds
Create instance took 5 seconds
Create instance took * seconds

Figure 3.4: Example of how tokens can be marked as parameters.

In the cases where first token of logline does not match with with any in the parse tree new branch is created as illustrated in Figure 3.5.

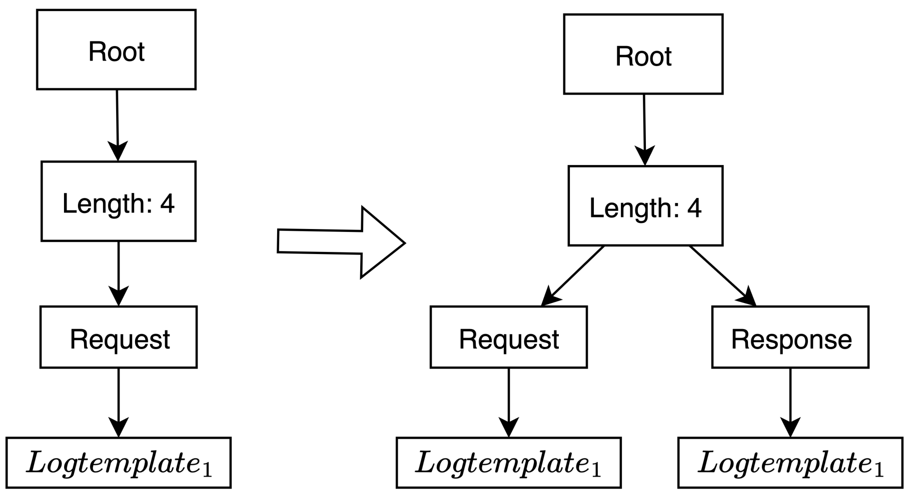

Figure 3.5: Example of Drain parse tree update.

3.6 DeepLog

DeepLog [9] marked a significant departure from previous approaches by treating log anomaly detection as more akin to a natural language task than a binary classification problem. The authors argued that log data exhibits similarities to natural language, suggesting that it could be approached with techniques used in natural language processing. Unlike earlier methods that heavily relied on manual feature extraction, DeepLog introduced a novel paradigm.

One of the most remarkable features of the DeepLog model is its capacity to be trained solely on normal log data. This means that extensive manual labeling of log lines as normal or anomalous is unnecessary. The underlying rationale here is that during regular system operation, the majority of log lines are expected to be normal, overshadowing the anomalous ones. Consequently, training on a comprehensive set of log lines allows the model to learn from the prevailing normal patterns, mitigating the impact of anomalous instances on prediction accuracy.

The initial step of the DeepLog workflow involves parsing raw log lines into log templates and associated parameters. This parsed information subsequently serves as the foundation for generating log key sequences and parameter value vectors. Each log line is parsed into a numerical representation corresponding to its log type, while the log's parameters are extracted to construct parameter value vectors. Figure 3.6 provides an illustration of this parsing process.

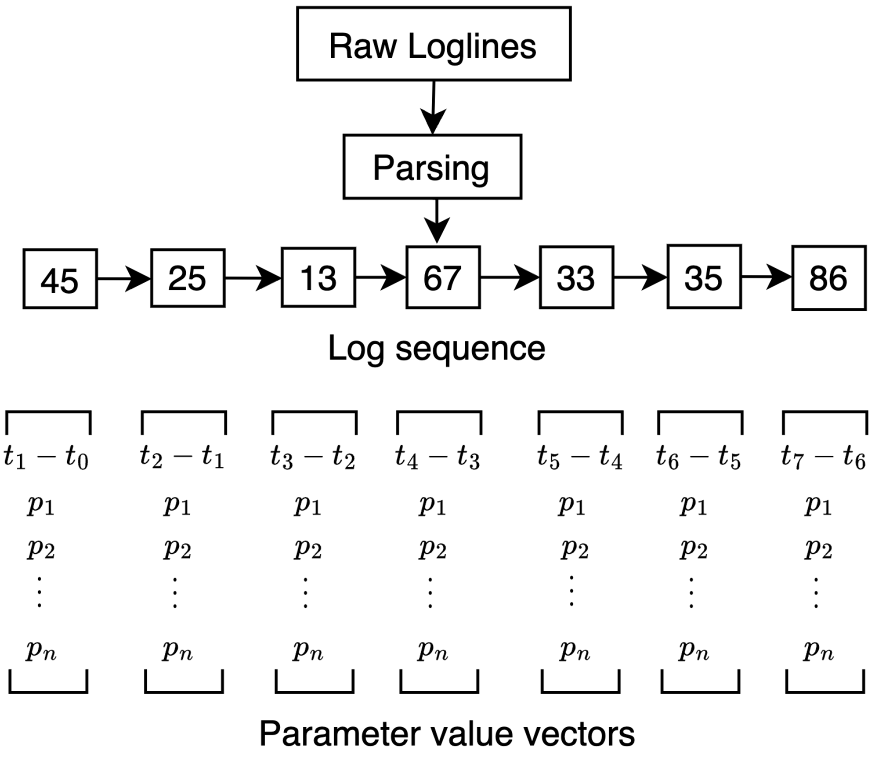

Figure 3.6: Parsing process in DeepLog.

Upon completing the parsing phase, DeepLog produces two distinct models: the log key anomaly detection model and the parameter value anomaly detection model. These models collectively contribute to the identification of anomalies in log data. Additionally, DeepLog includes a workflow model that empowers operators to diagnose anomalies and refine the models in cases of false positives. The architecture of DeepLog is depicted in Figure 3.7.

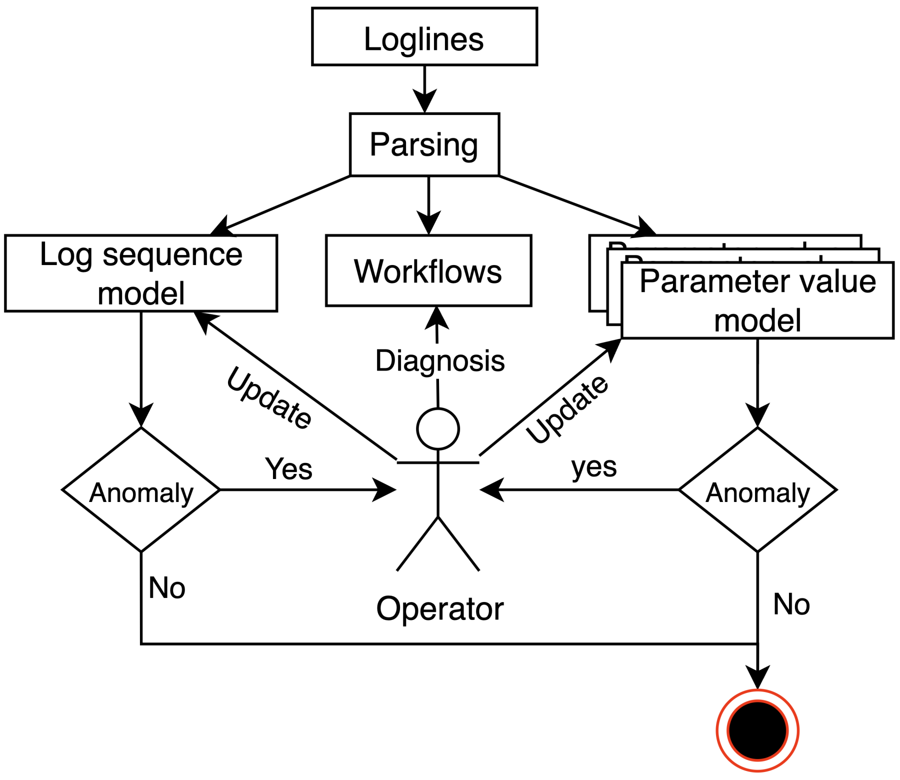

Figure 3.7: Architecture of DeepLog.

3.6.1 Log Key Model

The primary objective of the Log Key Model is to predict the subsequent log key in a sequence of log keys. The prediction task can be illustrated as follows: Let's take a sequence of log keys, $k_{22}, k_5, k_{11}, k_9, k_{11}, k_{26}$. With a window size of $h = 3$, the input-output pairs for training the Log Key Model are constructed as: $k_{22}, k_5, k_{11} \rightarrow k_9$, $k_5, k_{11}, k_9 \rightarrow k_{11}$, and $k_{11}, k_9, k_{11} \rightarrow k_{26}$. This concept is depicted in Figure 3.8.

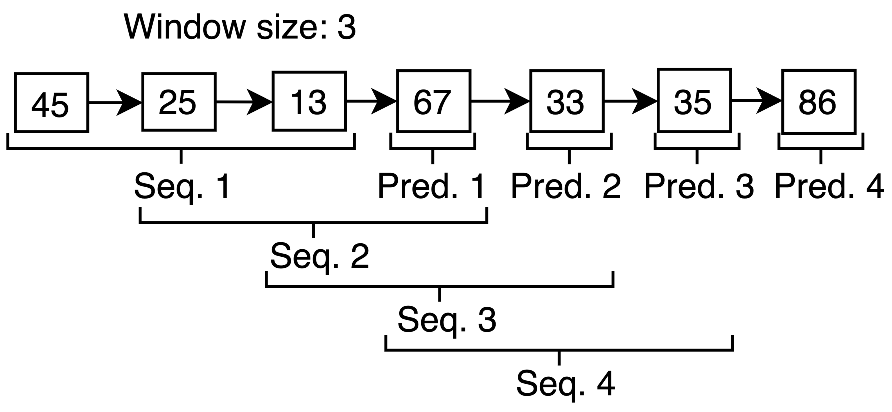

Figure 3.8: DeepLog log sequence prediction.

To assess whether a log key m_t is normal or abnormal, the model takes a window $w = \{m_{t-h}, ..., m_{t-1}\}$ as input. The output comprises a probability distribution denoted as $Pr[m_t|w] = \{k_1 : p_1, k_2 : p_2, ..., k_n : p_n\}$. This distribution indicates the likelihood of each log key from the set K appearing as the next log key value, given the historical context. In practical terms, multiple log keys might correctly follow as the next in sequence. Consequently, the Log Key Model arranges potential log keys from set K based on their probabilities $Pr[m_t|w]$. A key is considered normal if it ranks among the top g candidates. This procedure is outlined in Figure 3.9.

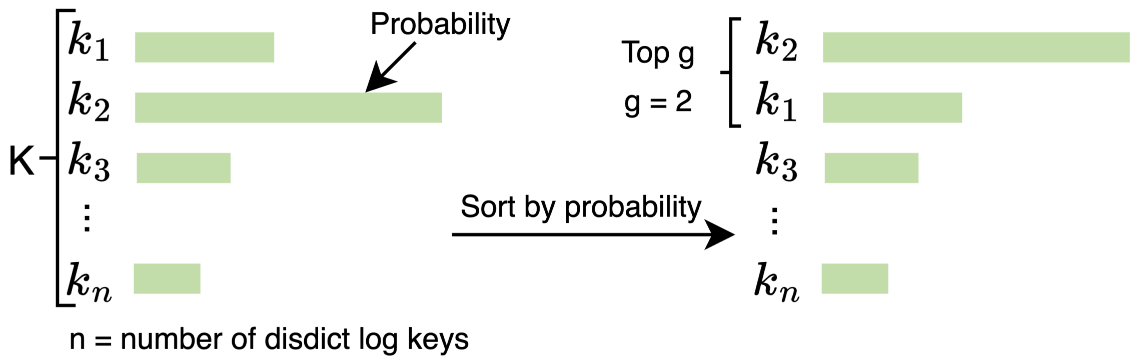

Figure 3.9: DeepLog log sequence top-g prediction.

3.6.2 Parameter Value Prediction Model

For each distinct log key, an individual Parameter Value LSTM (Long Short-Term Memory) network model can be tailored. This model is designed to anticipate forthcoming parameter values based on historical patterns. Constructing the input vector involves a two-step process. Let's illustrate this with examples: consider log lines such as "Took 0.61 seconds to deallocate network" and "Took 1 second to deallocate network." The input for the model is a parameter value vector (e.g., time duration) extracted from the corresponding timestamp. The output, as depicted in Figure 3.10, is a prediction for the subsequent parameter value vector.

To ensure uniformity and comparability, each value within the vector is normalized using the mean and standard deviation calculated from all values belonging to the same parameter position within the training data.

$$\text{Input: } \left[t_2 - t_1, p_1, p_2, \ldots, p_n\right]$$

$$\text{Output prediction: } \left[t_3 - t_2, p_1, p_2, \ldots, p_n\right]$$

Figure 3.10: DeepLog Parameter Model Input and Outputs.

This model generates an output that is a vector of real values, forecasting the next parameter value vector based on a sequence of parameter value vectors from recent historical data.

For anomaly detection, the Mean Squared Error (MSE) between the predicted and observed value vectors is calculated. However, rather than arbitrarily setting an error threshold for anomaly detection, a more robust approach is taken. The training data is divided into two distinct subsets: the model training set and the validation set. Following the model's training phase, the validation set is employed to make predictions for each time step. The MSE between the predictions and the actual observations is then modeled as a Gaussian distribution.

During production usage, if the error between the predicted and observed vectors falls significantly above the Gaussian distribution's confidence interval, the incoming log entry's parameter value vector is considered normal otherwiese it is considered abnormal.

3.6.3 Updating parameters

As system behavior evolves over time, training data might not cover all potential normal patterns. To accommodate this, the ability to modify model weights for adapting to new log patterns is essential.

When users report model false positives, DeepLog improves accuracy without complete retraining. It updates weights using new input-output pairs from reported anomalies. This doesn't demand retraining from scratch, saving time and resources.

DeepLog efficiently uses reported false positives to update. New training data adjusts weights to minimize error between model output and actual observations in these instances. DeepLog learns and integrates new patterns while retaining existing knowledge.

Updating parameters and weights lets DeepLog adapt to emerging log patterns while retaining responsiveness and accuracy. This intelligent process ensures continual improvement without exhaustive retraining.

3.7 Robust Log Anomaly Detection

While DeepLog has proven effective for log-based anomaly detection, it has a significant limitation: it requires prior knowledge of all existing log keys during model construction. This drawback means that the model will not function with previously unseen log keys. In the paper by Zhang et al. [10], the authors conducted an empirical study and found that real-world log data exhibits instability, with new but similar log keys frequently emerging. One major factor contributing to this phenomenon is the evolution of logging statements. A study by Kabinna et al. [11] observed that approximately 20% $\sim$ 45% of logging statements changed over their lifetime. As a result, log-based anomaly detection methods must accommodate the evolution of log data.

To address the issue of log instability, Zhang et al. proposed LogRobust in [10], a novel log-based anomaly detection approach that achieves accurate and robust anomaly detection on dynamic and noisy real-world log data. Instead of directly using log keys, LogRobust transforms each log key into a fixed-dimensional semantic vector. This enables the model to comprehend new but similar log keys that emerge due to evolving logging statements and parsing errors. The authors utilized a Bidirectional Long Short-Term Memory Neural Network (Bi-LSTM) classification model to detect anomalies. They evaluated LogRobust using public log data from Hadoop, introducing varying levels of changes, and demonstrated its effectiveness. The experiments showed that LogRobust remained robust, with the F1-score decreasing only slightly from 0.96 to 0.89 as the injection rate increased from 5% to 20%.

3.7.1 Architecture

The LogRobust architecture, outlined in Figure 3.11, consists of three layers. The first layer involves log parsing, which converts raw log data into a structured format using the Drain parser. The second layer transforms each log-event sentence into a semantic vector, where semantically equivalent words yield similar vectors. The final layer is an attention-based Bi-LSTM classifier that outputs the probability of a log line being anomalous.

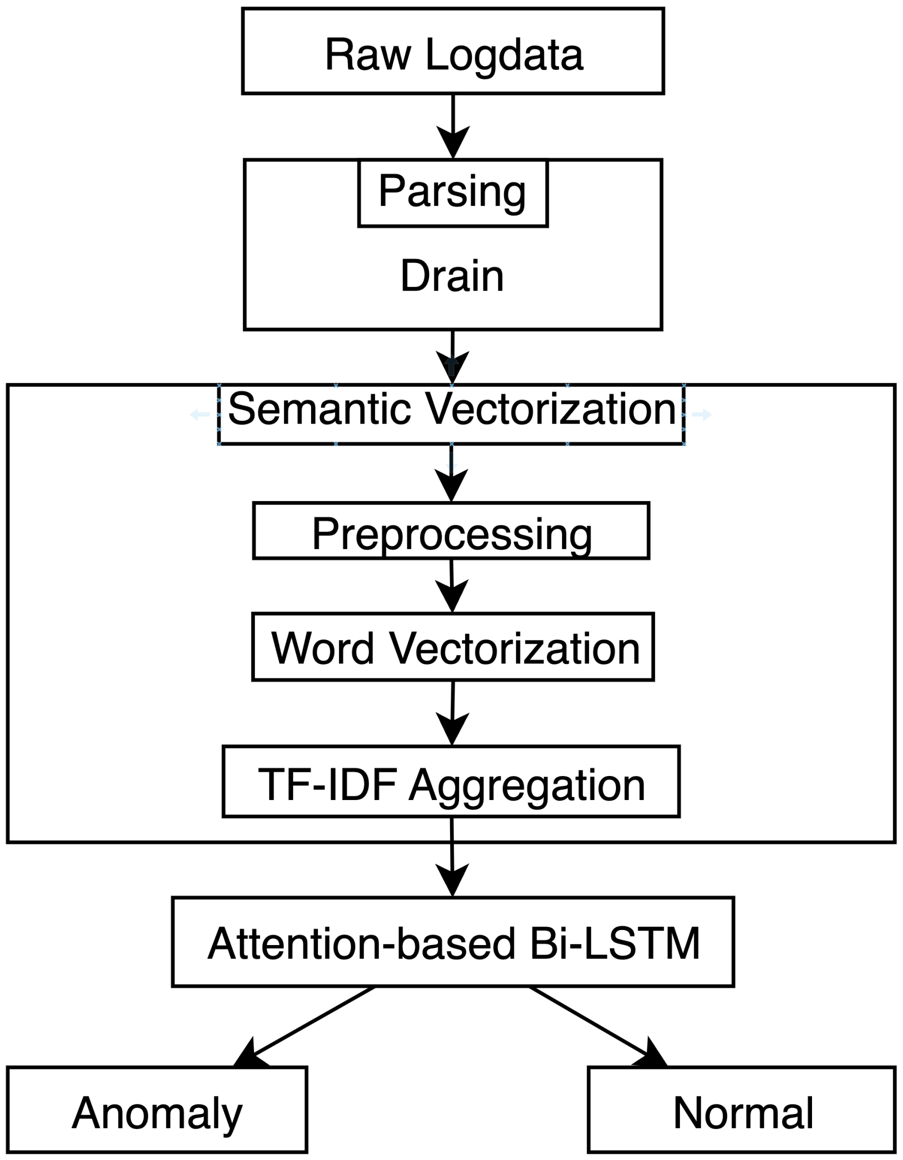

Figure 3.11: LogRobust architecture.

3.7.2 Semantic Vectorization

Parsed log lines undergo semantic word vector transformation, as depicted in Figure 3.12.

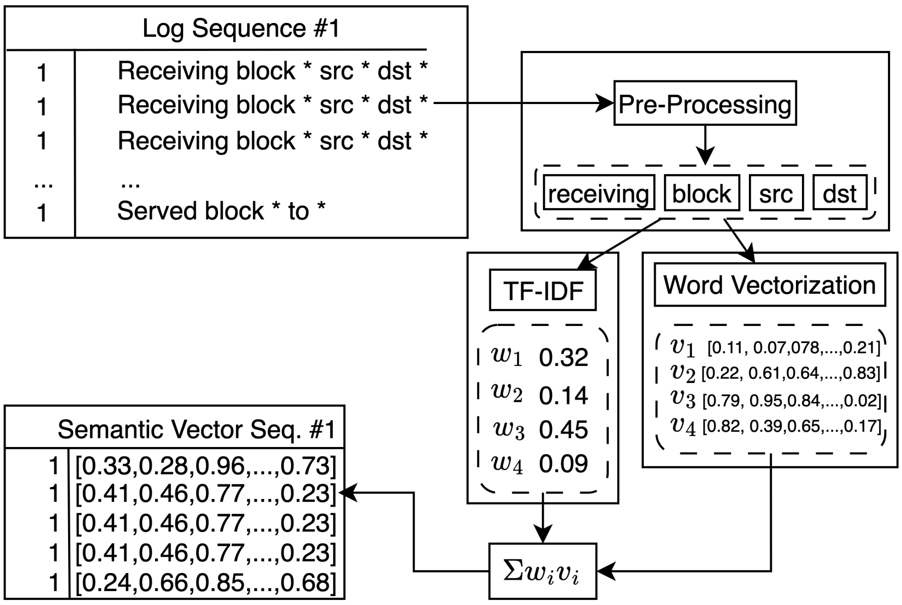

Figure 3.12: LogRobust semantic vectorization.

Preprocessing

LogRobust treats a log line as a natural language sentence, with most tokens being valid English words carrying distinct meanings. However, non-character tokens and variable names are also present in logs. To preprocess each log-event sentence, LogRobust first removes non-character tokens such as delimiters, operators, punctuation marks, and number digits. Subsequently, stop words like "a" and "the" are removed. Additionally, some variable names in log events are composed of concatenated words, such as "TypeDeclaration" containing "type" and "declaration," or "isCommitable" consisting of "is" and "commitable." To handle this, LogRobust splits these composite tokens into individual tokens using Camel Case.

Word Vectorization

Following preprocessing, LogRobust transforms each log-event sentence into a semantic vector. To satisfy two requirements—discrimination and compatibility—LogRobust utilizes off-the-shelf word vectors pretrained on the Common Crawl Corpus dataset using the FastText algorithm [12]. After replacing words with corresponding vectors, a log-event sentence S is transformed into a list of word vectors $L = [\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_N]$, where $\mathbf{v}_i \in \mathbb{R}^d$, $i \in [1, N]$, denotes the word vector, and N is the number of tokens in the log-event sentence.

TF-IDF Aggregation

To obtain a fixed-sized semantic vector representing a log event, LogRobust aggregates the N word vectors in L, accommodating varying word counts across log events. Achieving this goal involves using TF-IDF, which measures word importance in sentences. LogRobust calculates the Term Frequency (TF) of a word as $TF(\text{word}) = \frac{\text{count(word)}}{\text{total events}}$, where count(word) is the number of occurrences of the word and total events is the total number of log events. However, if a word, such as "Block," appears in all log events, it becomes less discriminative, warranting a weight reduction. Inverse Document Frequency (IDF) is employed to address this, defined as $IDF(\text{word}) = \log\left(\frac{\text{total events}}{\text{events with word}}\right)$, where events with word is the number of log events containing the target word. For each word, its TF-IDF weight w is computed as $w = TF \times IDF$. Ultimately, the semantic vector $\mathbf{V} \in \mathbb{R}^d$ representing a specific log event is obtained by summing the word vectors in L weighted by their TF-IDF values, following the equation: $\mathbf{V} = \frac{1}{N}\sum_{i=1}^{N} w_i \mathbf{v}_i$.

3.7.3 Anomaly Detection

LogRobust employs an attention-based Bi-LSTM neural network for anomaly detection, illustrated in Figure 3.13.

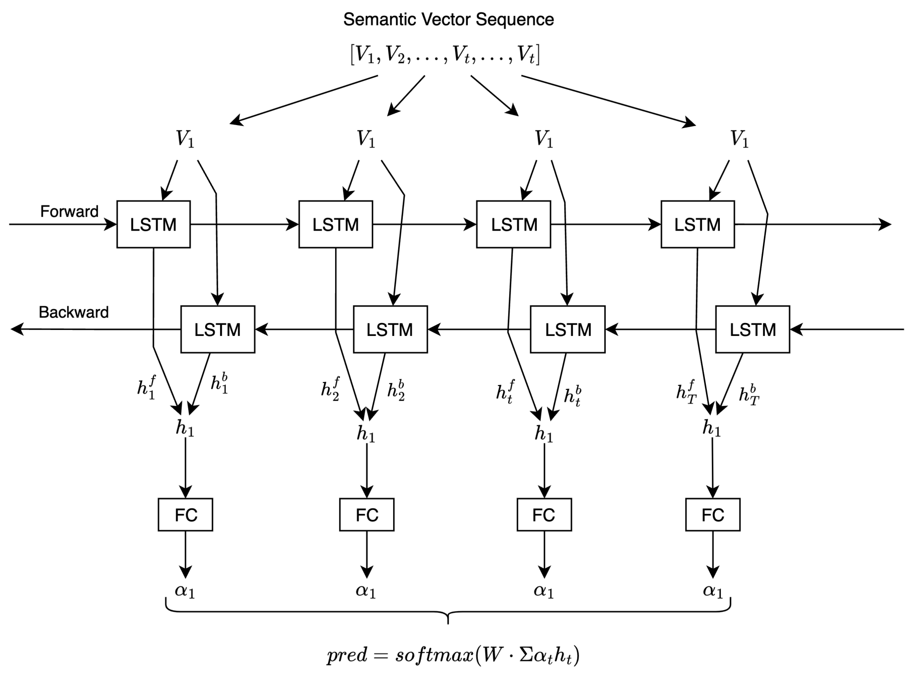

Figure 3.13: LogRobust classification model.

3.8 HitAnomaly

The Transformer architecture, introduced in the seminal paper by Vaswani et al. [13], has outperformed the traditional LSTM models in various natural language processing (NLP) tasks, making it the go-to architecture for state-of-the-art NLP models. For instance, BERT [14], one of the most successful pre-trained language models, is built on top of Transformer.

However, despite the success of the Transformer models, they have not been extensively used for anomaly detection. In the context of anomaly detection, the previous approaches primarily relied on the log key sequences, with the exception of DeepLog [9]. However, DeepLog's approach of using separate parameter value models for each distinct log key may not be sufficient for detecting anomalies that arise from abnormal parameter values.

Consider the following two log lines: "Took 10 seconds to build instance" and

"Took 600 seconds to build instance". These log lines have the same log key, which is "Took * seconds to build instance", but the parameter values are significantly different. If building an instance takes too long, it could be considered an anomaly. While DeepLog's parameter value vector could work in this case, it may not be effective in scenarios where anomaly detection requires knowledge of the execution path and parameter values.

To address these limitations, Huang et al. [3] proposed HitAnomaly, which uses hierarchical Transformer models instead of hierarchical LSTM models for anomaly detection. By leveraging the strengths of Transformers, HitAnomaly can capture complex relationships between log keys and parameter values, leading to better anomaly detection performance.

3.8.1 Architecture

The HitAnomaly architecture, as outlined in 3.14, consists of three primary steps: pre-processing, encoding, and anomaly detection. The pre-processing step resembles those employed in prior approaches, with the main objective being the conversion of raw log text into parameters and log keys. HitAnomaly utilizes the Drain algorithm [2] due to its speed and accuracy, outperforming many alternative log parsing methods. However, it is worth noting that Drain [2] may occasionally introduce noise, as it can produce inaccurate results. According to [15], log parsing errors can lead to a significant performance degradation in log mining, sometimes by an order of magnitude.

Following pre-processing, the next phase in HitAnomaly is encoding. This stage employs two distinct encoders: Log Sequence Encoding and Parameter Value Encoding. Both encoders utilize a hierarchical transformer to convert log key sequences and parameter values into fixed-size vectors. These vectors enable the model to effectively capture semantic information and manage contextual knowledge within the logs.

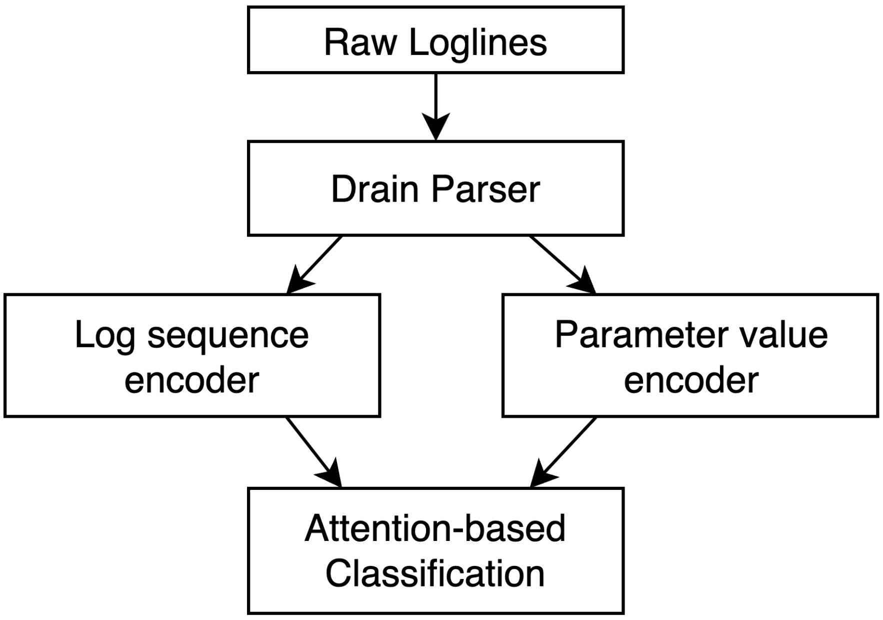

Figure 3.14: HitAnomaly architecture

3.8.2 Encoding

Logging Key Sequences

The log encoder involves four essential steps: 1) Word2Vector; 2) Self-Attention; 3) Feed Forward Neural Network (FFN); 4) Pooling Layer.

1. **Word2Vector:** The initial phase of the encoding process revolves around converting each word in the log template into a corresponding vector, denoted as $\mathbf{x}$. By utilizing one-hot encoded vectors for each word, a lookup operation is performed within an embedding matrix to retrieve the corresponding word vector. In this context, HitAnomaly utilizes word vectors derived from BERT and adopts the WordPiece [16] tokenization strategy. WordPiece functions as a text segmentation technique that dissects words into subword units, allowing for a more nuanced representation. For instance, an IP address such as

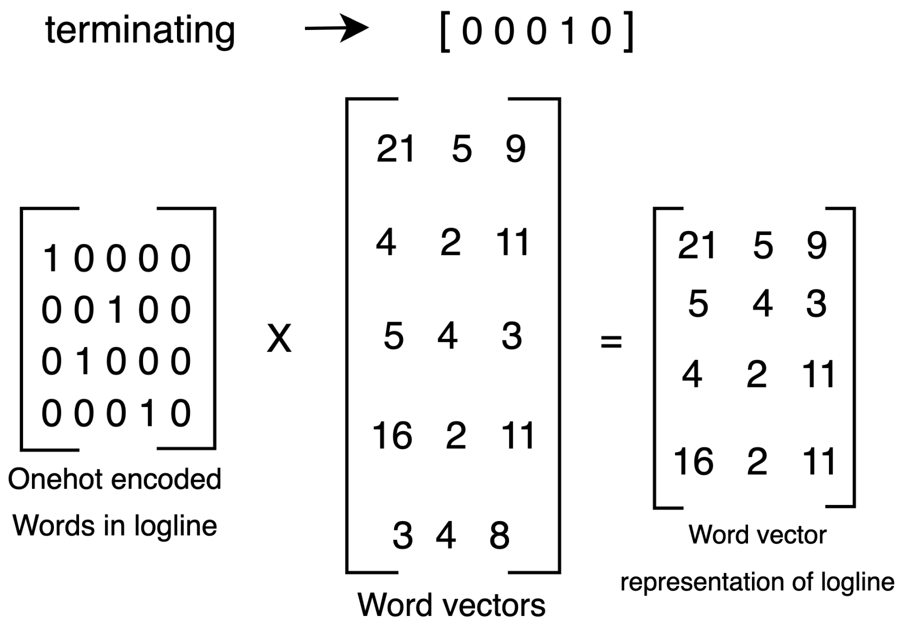

Figure 3.15: Visualization of the conversion process from a log line to a word vector list.

"10.251.122.79:50010" is tokenized into segments like ["10", ".", "251", ".", "122", ".", "79", ":", "500", "10"]. Substituting words with their respective vectors results in the transformation of the log line S into a vector list denoted as $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n]$, as visually depicted in Figure 3.15.

2. **Self-Attention:** The vector list $\mathbf{X}$ flows through the self-attention layer. This layer calculates self-attention using three vectors for each vector $\mathbf{x}$: the query vector $\mathbf{Q}$, the key vector $\mathbf{K}$, and the value vector $\mathbf{V}$. These vectors are computed using the following formulas:

$$\mathbf{Q}_i = \mathbf{x}_i \mathbf{W}_Q$$

$$\mathbf{K}_i = \mathbf{x}_i \mathbf{W}_K$$

$$\mathbf{V}_i = \mathbf{x}_i \mathbf{W}_V$$

Here, $\mathbf{W}_Q \in \mathbb{R}^{d \times d_q}$, $\mathbf{W}_K \in \mathbb{R}^{d \times d_k}$, and $\mathbf{W}_V \in \mathbb{R}^{d \times d_v}$ are parameter matrices, and d_q, d_k, and d_v represent the dimensions of the query, key, and value vectors respectively. The output of the self-attention layer, denoted as $\mathbf{z}_i$, is computed using the formula:

$$\mathbf{z}_i = \text{softmax}\left(\frac{\mathbf{Q}_i\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V}$$

In practice, these calculations are performed in parallel, and the results are organized into matrices $\mathbf{Q}$, $\mathbf{K}$, and $\mathbf{V}$. In the context of HitAnomaly, a multi-headed attention mechanism is employed. This means that instead of using a single set of query/key/value weight matrices, there are multiple sets. In their transformer model, twelve attention heads are used. The final computation can be expressed as:

$$\text{head}_h = \text{softmax}\left(\frac{\mathbf{Q}_{i,h}\mathbf{K}_h^T}{\sqrt{d_k}}\right)\mathbf{V}_h$$

$$\mathbf{z}_i = \text{concat}(\text{head}_1, ..., \text{head}_h)\mathbf{W}_O$$

Here, $\mathbf{Q}_{i,h} = \mathbf{x}_i\mathbf{W}_{Q,h}$, where $\mathbf{W}_{Q,h}$ is the parameter matrix for attention head h. $\mathbf{K}_h$ and $\mathbf{V}_h$ are the key matrix and value matrix respectively for attention head h. $\mathbf{W}_O \in \mathbb{R}^{hd_v \times d}$ is a trainable parameter used for the concatenation operation. The multi-headed attention mechanism enhances the model's ability to focus on different positions and learn diverse representations.

3. **Feed-Forward Network:** Subsequently, the outputs from the self-attention layer are propagated through the feed-forward network. The feed-forward network is defined by the following equation:

$$\mathbf{r}_i = \max(0, \mathbf{z}_i\mathbf{W}_1 + \mathbf{b}_1)\mathbf{W}_2 + \mathbf{b}_2$$

Here, $\mathbf{W}_1 \in \mathbb{R}^{d \times d_{\text{ff}}}$, $\mathbf{W}_2 \in \mathbb{R}^{d_{\text{ff}} \times d}$, and $\mathbf{b}_2 \in \mathbb{R}^d$ are the trainable parameters in the feed-forward network layer. The dimension of the feed-forward network is denoted as d_{ff}.

4. **Pooling Layer:** In the architecture, the concluding layer is the pooling layer, which plays a pivotal role in weight reduction and the conversion of a template into a standardized vector of fixed dimensions. This process offers a consolidated representation while maintaining the essence of the original data. The computation is as follows:

$$R_l = \frac{1}{n} \sum_{i=1}^{n} r_i$$

Here, $r_i \in \mathbb{R}^d$ signifies the output vector produced by the second transformer block at the position i within the sequence. After the sequence of log encoding, a list of log representations $R_L = [R_{L1}, R_{l2}, ..., R_{LN}]$ is generated, where R_{LN} corresponds to the log representation at the N-th position.

Upon applying a subsequent transformer layer to the sequence of log template representations R_L, we obtain refined and context-sensitive log template representations denoted as $Z_L = [z_{L1}, z_{L2}, ..., z_{LN}]$. These enriched representations capture intricate relationships and dependencies among the log templates.

Finally, to culminate the process, an average pooling layer is employed to seamlessly transform the log template representations Z_L into a comprehensive and cohesive log sequence representation, aptly named R_s. This final representation encapsulates the essential features of the entire log sequence, facilitating effective analysis and anomaly detection with improved accuracy.

Parameter value vector

The sequence of log templates plays a crucial role in identifying anomalies in program execution paths. However, certain anomalies manifest not as deviations from a typical template sequence, but rather as unusual parameter values. DeepLog introduced a parameter value detection model that treats individual log key parameters as distinct time series. However, this model exclusively handles numerical values and generates a parameter value vector using the mean and standard deviation of all values. This approach encounters issues in scenarios like distinguishing between "Took 600 seconds to build instance" (possibly anomalous) and "Took 600 ms to build instance" (normal event).

In contrast, the HitAnomaly approach employs a hierarchical transformer structure to better model parameter values. Initially, parameter values sharing the same log template are grouped together, resulting in log template parameter groups. Each group shares a common template and encompasses a series of associated parameters. Similar to the methodology applied to log sequence encoding, parameters undergo encoding. This process yields a list of parameter value representations: $[r_{v1}, r_{v2}, ..., r_{vM}]$. These representations are then fed through an average pooling layer, resulting in a consolidated vector $R_{v1} \in \mathbb{R}^{1xd}$.

Mathematically, the transformation of the representations list into the vector R_{v1} is expressed as:

$$R_{v1} = \frac{1}{M} \Sigma_{i=1}^{M} r_{vi}$$

The template vector R_{L1} is used to signify the template, and a linear layer is introduced to model the interplay between templates and parameter values. The comprehensive equation for a single group is computed as:

$$R_{v1} = R_{L1}\mathbf{W}1 + Rv1\mathbf{W}_2 + b$$

Here, $\mathbf{W_1} \in \mathbb{R}^{dxd}$, $\mathbf{W_2} \in \mathbb{R}^{dxd}$, and $b \in \mathbb{R}^{1xd}$ denote trainable parameters that are updated during training.

Ultimately, an additional average pooling layer is employed to convert the list of parameter representations $[R_{V1}, R_{v2}, ..., R_{VN}]$ into the final representation of the parameter sequence, denoted as R_p.

Attention-Based Classification

In the context of log analysis, the transformation of a log sequence involves generating two distinct vectors: the log sequence representation denoted as R_s, and the parameter sequence representation denoted as R_p. Since these representations, R_s and R_p, exert varying influences on the classification outcome, an attention mechanism is harnessed to construct a classification model. This mechanism assigns different weights to the two representations, allowing for a more nuanced understanding of their importance. The attention scores are calculated as follows:

$$\alpha_s = \frac{exp(f(\mathbf{R_s}))}{exp(f(R_s))+exp(f(R_p))}$$
$$\alpha_p = \frac{exp(f(R_p))}{exp(f(R_s))+exp(f(R_p))}$$

The computations of $f(R_s)$ and $f(R_p)$ are as follows:

$$f(R_s) = v^T \tanh(R_s \mathbf{W} \alpha)$$
$$\text{f}(\text{R}_p) = v^T \tanh(R_p \mathbf{W} \alpha)$$

Here, W_α and v are parameters to the attention layer. Subsequently, a softmax layer is introduced to calculate the probability distribution for anomalies:

$$y = \text{softmax}(\alpha_s R_s + \alpha_p R_p)$$

Training of the model is executed using the Stochastic Gradient Descent (SGD) algorithm [17], with the cross-entropy loss serving as the chosen loss function.

3.9 LanoBERT

In the realm of anomaly detection from log data, conventional methods often rely on log parsers to transform unstructured logs into a structured format. Notably, the Drain parser [2] has gained significant popularity and was discussed in previous sections. While effective log parsers contribute to robust anomaly detection, their performance is closely linked to the accuracy and efficiency of the parsing process. A significant challenge emerges from software updates, which frequently lead to alterations in logging statements. Even minor changes can impede anomaly detection when semantically similar log concepts are expressed differently but retain the same meaning. To address this challenge, alternative solutions such as LogRobust [10] and HitAnomaly [3] have been developed.

These approaches leverage word vectors and fixed-size semantic vectors to enhance their ability to withstand variations in logging statements. The primary objective is to maintain consistent performance as software ecosystems evolve over time.

In response to the limitations of traditional methods, a novel solution called LanoBERT [4] has been introduced. LanoBERT breaks away from reliance on log parsers by harnessing the power of the BERT [14] language model. Unlike methods tied to parsers, LanoBERT employs BERT as a versatile language model that has been trained from scratch specifically for anomaly detection.

A notable characteristic of LanoBERT lies in its parser-free methodology, which involves minimal preprocessing exclusively for entities such as Numbers, IPs, and dates. After this preprocessing, LanoBERT makes use of the standard tokenizer WordPiece [16], a foundational component used by BERT. Notably, the tokenizer used for log data has also undergone training from scratch. This meticulous approach ensures that the vocabulary of log data from each distinct system can be thoroughly learned and integrated into the anomaly detection process.

In summary, LanoBERT represents a significant advancement in anomaly detection by liberating itself from the constraints of log parsers. By harnessing the capabilities of BERT and employing a well-crafted tokenizer, LanoBERT provides a robust and versatile solution capable of adapting to the changing nature of software systems.

3.9.1 Architecture

The architecture of LanoBERT, as depicted in Figure 3.16, closely resembles the model architecture of BERT. However, a significant difference exists: LanoBERT omits the Next Sentence Prediction (NSP) training objective. Instead, it exclusively employs the Masked Language Model (MLM) technique [18], which aligns with the pre-training approach used by BERT.

The training parameters within LanoBERT closely mirror those utilized by the original BERT model, differing primarily in the masking probability. LanoBERT employs a masking probability of 20

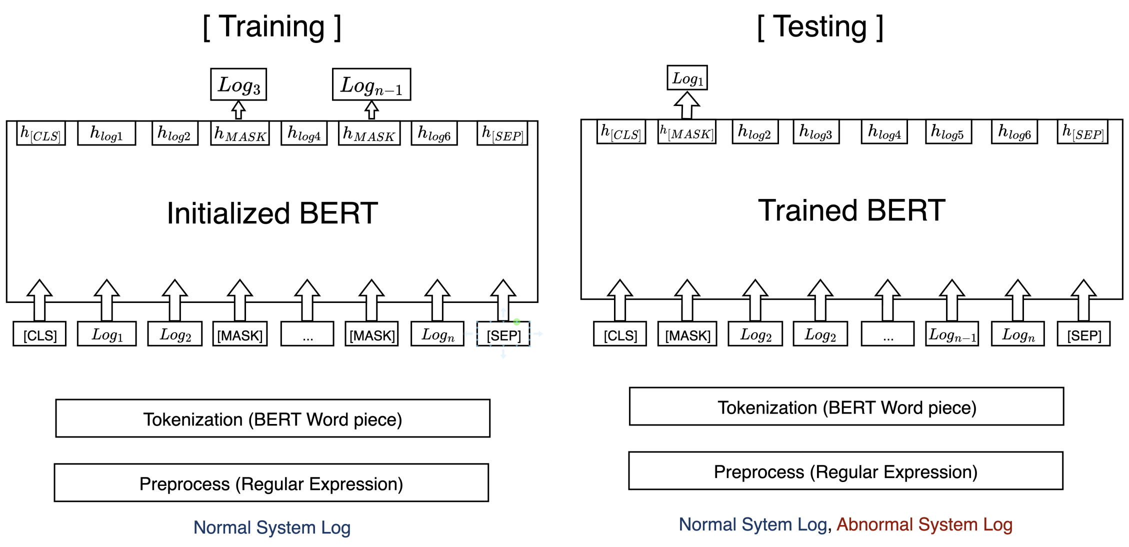
[Training]
[Testing]
Log_3
Log_{n-1}
Log_1
$h_{[CLS]}$ h_{log1} h_{log2} h_{MASK} h_{log4} h_{MASK} h_{log6} $h_{[SEP]}$
$h_{[CLS]}$ $h_{[MASK]}$ h_{log2} h_{log3} h_{log4} h_{log5} h_{log6} $h_{[SEP]}$
Initialized BERT
Trained BERT
[CLS] Log_1 Log_2 [MASK] ... [MASK] Log_n [SEP]
[CLS] [MASK] Log_2 Log_2 ... Log_{n-1} Log_n [SEP]
Tokenization (BERT Word piece)
Preprocess (Regular Expression)
Normal System Log
Normal Sytem Log, Abnormal System Log

Figure 3.16: LanoBERT architecture

3.9.2 Training

LanoBERT employs normal log data for training, as previously mentioned, and does not rely on a separate parser for logs. Instead, it harnesses the standard WordPiece tokenizer [16], which is also utilized by BERT. The Masked Language Model (MLM) approach is applied during both the training phase and anomaly detection. Given that the majority of log data consists of regular log entries, this strategy proves effective for pre-training the BERT model. With a sufficient volume of data, LanoBERT acquires the ability to learn contextual and structural features from the log data, thereby enhancing its generalization performance in detecting abnormal logs by understanding the characteristics of a normal log system. Notably, LanoBERT does not require labeled log data for training; it solely relies on normal data.

3.9.3 Anomaly Detection

LanoBERT's central assumption is rooted in its primary training with normal logs, leading to the inference that it yields low errors and high predictive probabilities when applying MLM to normal log data inputs. Commonly observed logs with regular patterns are recognized as familiar contexts. Conversely, larger errors and lower predictive probabilities are generated when applying MLM to abnormal log data inputs, as depicted in Figure 3.17.

Abnormal Score

LanoBERT computes an abnormal score for anomaly detection, derived from individual abnormal scores. This strategy is motivated by the similarity in features often exhibited by normal and abnormal logs. The prediction error introduced by LanoBERT can be defined as follows: given a log sequence $L = (w_1, w_2, ..., w_N)$, a sequence is generated for each log key, and predictive probabilities and prediction

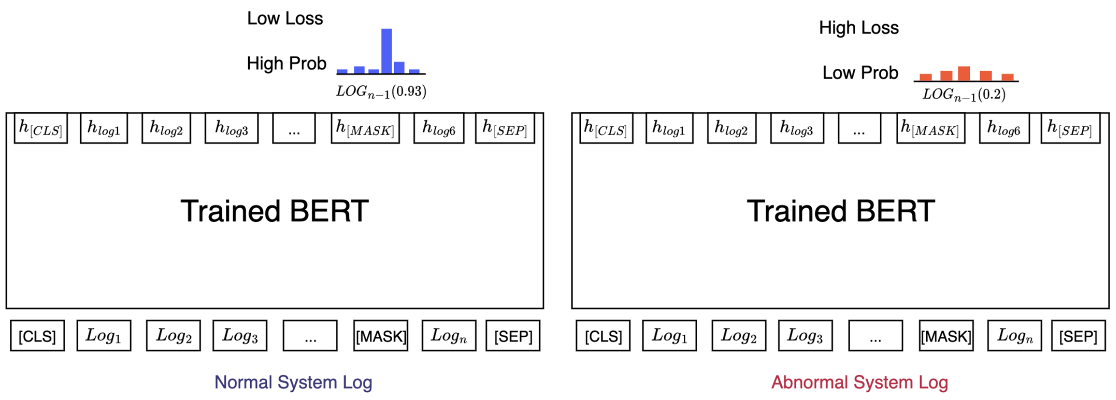
Low Loss
High Prob
$LOG_{n-1}(0.93)$
High Loss
Low Prob
$LOG_{n-1}(0.2)$
$h_{[CLS]}$ h_{log1} h_{log2} h_{log3} ... $h_{[MASK]}$ h_{log6} $h_{[SEP]}$
Trained BERT
[CLS] Log_1 Log_2 Log_3 ... [MASK] Log_n [SEP]
Normal System Log
Abnormal System Log

Figure 3.17: LanoBERT normal vs abnormal logs.

errors are computed. Tokens are repeatedly replaced with the [MASK] token to facilitate error and probability calculation for each logline. The top k values are selected. The count of predictions equals the length of the log sequence, as illustrated in Figure 3.18. The final abnormal score is calculated using these values, as depicted in Equations 3.9.3 and 3.9.3.

$$abnormal_{error} = \sum_i^k \frac{error_1 + error_2 + ... + error_k}{k}$$

$$abnormal_{prob} = \sum_i^k \frac{prob_1 + prob_2 + ... + prob_k}{k}$$

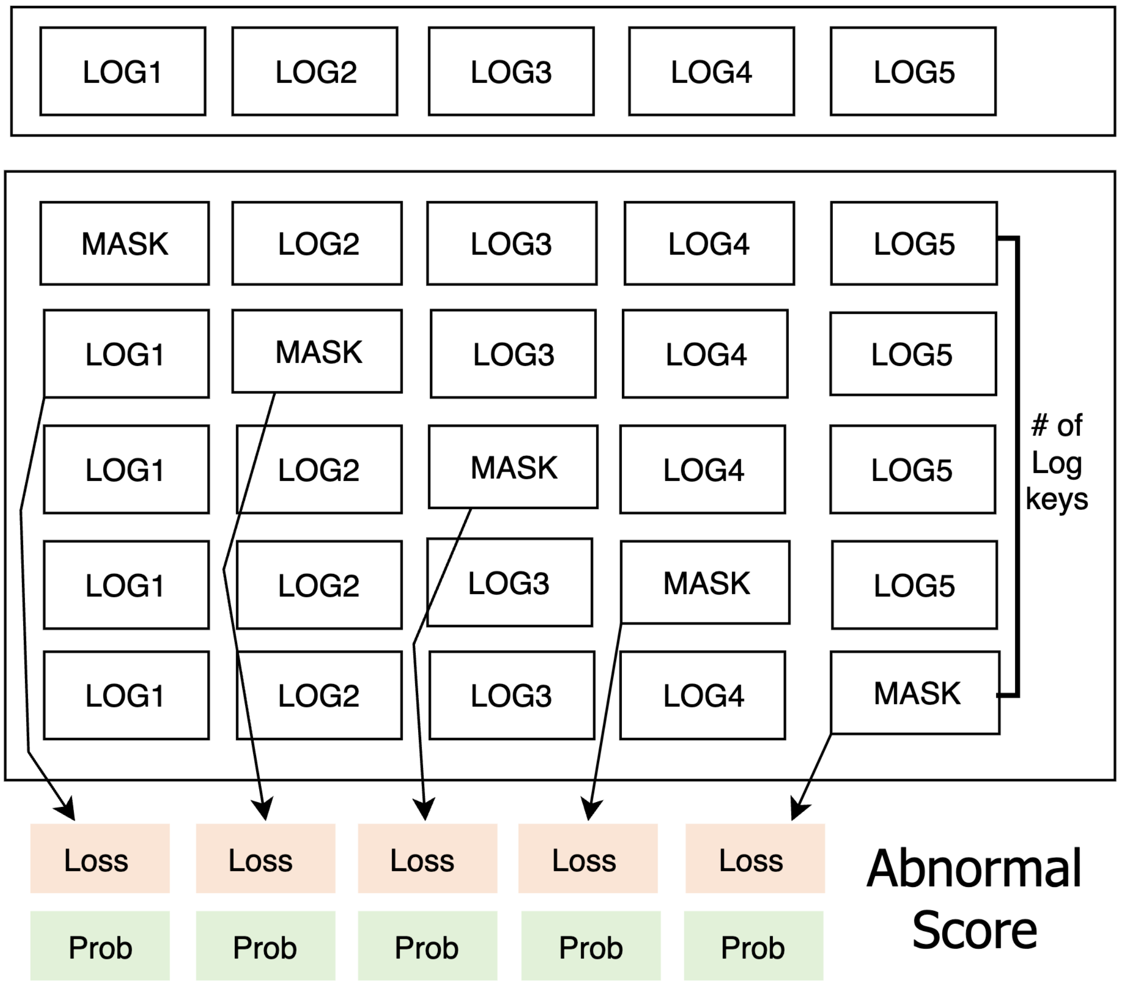

Figure 3.18: Testing Phase of LanoBERT Model.

3.10 Summary

The first log anomaly detection method, introduced in the 90s, was Swatch [5], which employed simple regular expression rules to identify anomalous logs. However, this method was unable to detect anomalies based on log sequences. Logsurfer, the second method discussed in this chapter, also used regular expressions for anomaly detection, but featured a more complex rule system capable of detecting anomalies from sequences of messages. This was achieved through the use of context and dynamic rule addition and removal. Despite their power, these rules required considerable effort to develop and maintain.

Subsequent methods aimed to automate anomaly detection more effectively. Data mining techniques were employed to cluster logs into relevant groups, followed by the application of association rule algorithms to detect temporal associations between event types. The authors in [8] parsed logging templates from raw log data using source code, which were then used to parse raw log data into log events. Principal Component Analysis (PCA) was employed to determine when a log was considered anomalous. This approach, however, had limitations in detecting complex errors from log sequences and faced challenges related to parsing different programming languages and source code availability. Consequently, Spell and Drain parsers were developed to parse raw log data into log templates using unsupervised techniques.

DeepLog [9] was the first method to utilize Deep Learning for anomaly detection, using Long Short-Term Memory (LSTM) networks. The model detected execution path anomalies, such as a log line appearing in an incorrect place. However, some anomalies could not be detected by this model, and DeepLog struggled with evolving logging statements. To address these issues, LogRobust [10] transformed log data into fixed-size semantic vectors that were more resilient to changes in logging statements. By using off-the-shelf word vectors pre-trained on the Common Crawl Corpus dataset with the FastText algorithm, LogRobust was better equipped to handle changes in log line semantics.

HitAnomaly [3] employed a similar approach but utilized a transformer model and considered both log key sequences and their parameter values for anomaly detection. Most previous Deep Learning approaches relied on a log parser to convert raw text from log data into log templates, which could be used with Deep Learning models. LanoBERT [4] eliminated the need for a separate log parser, using a standard tokenizer with the BERT language model and applying Masked Language Modeling (MLM) to detect anomalies. This approach made the model more robust to changes and less dependent on parser performance.

Anomaly detection methods have evolved towards Deep Learning techniques with minimal pre-processing. Modern methods are more robust to changes in logging statements and can detect more complex anomalies by capturing intricate relationships within log data. This trend is also observed in other fields, such as computer vision and natural language processing. However, the exponential increase in computational power has enabled the use of advanced Deep Learning methods, which would not have been possible in the 90s or even the early 2000s. While these powerful methods are highly effective, they may be inaccessible or impractical for many organizations due to computational requirements. Moreover, they necessitate large amounts of training data, although modern systems can typically generate sufficient data.

4 Case Study

This thesis focuses on the log data generated by 1,000 to 2,000 Linux-based embedded payment terminals. These terminals are primarily installed in buses, with some found in swimming halls. Capable of running multiple programs concurrently, they generate overlapping log entries. The objective of this research is to identify log anomaly detection techniques from established literature and evaluate their suitability for this type of log data. The ideal technique would:

1. Reduce human effort in ensuring system stability.
2. Demand limited human oversight for maintenance.
3. Be cost-effective, considering the constrained resources.

The effectiveness of the chosen technique will be assessed by its ability to detect previously identified issues in the system.

Furthermore, this thesis will present strategies and delve into the specifics of enhancing the efficiency of the log collection and analysis process. Currently, logs are retrieved from the terminals using a script that employs SSH. This method, however, falls short when targeting offline terminals and often results in the collection of redundant logs, leading to unwarranted storage consumption.

4.1 Plan

4.1.1 Tooling

To effectively apply the methodologies gleaned from the literature to the log data produced by these devices, it's imperative to have the data in an easily accessible format. There are known anomalies from the past that can serve as benchmarks for the methodologies' efficacy. However, the current state of the log data, due to its inaccessible format, poses a challenge in extracting valuable log lines for model training and evaluation, despite their known existence. Thus, the first step in the testing and implementation phase of this thesis involves developing tools that gives more straightforward access to the logs.

Many anomaly detection models for log data necessitate a specific format—primarily converting textual log data to a numerical form. Drain [2], which permits log parsing in an unsupervised and online streaming manner, stands out as a paramount method in this conversion process. Integrating Drain into the tool will not only simplify the generation of datasets for training and evaluation but also be indispensable for real-time anomaly detection.

4.1.2 Testing

Initially, the model will be evaluated using a dataset augmented with injected anomalies. This approach helps establish a baseline performance in scenarios with optimal data quality. Following this, the model will be tested against real-world log data to gauge its practical applicability.

Subsequent to testing, analysis will be conducted to assess the model's performance. This will also provide insights into potential areas for future enhancement and refinement.

5 Designing the Data Processing Pipeline

In previous chapters, various methods for anomaly detection from log data were presented. The methods that yield the best results are based on deep learning, but they require a considerable amount of training data. Moreover, it is crucial that alerting thresholds can be adjusted to reduce false positives and false negatives when anomalies are detected. To achieve this, a carefully designed pipeline could be employed, which includes the ability to label data as anomalous or non-anomalous, enhancing the overall efficiency of the process.

5.1 Feedback Loop

To optimize the performance of anomaly detection methods, it is essential to incorporate feedback into the entire process. Without proper feedback mechanisms, these methods may detect anomalies but also generate a high number of false positives. This abundance of false positives can significantly undermine the system's credibility.

False positives can arise from inaccurately set detection thresholds or erroneous predictions made by the deep learning model. Typically, when the model identifies anomalies, a human operator must verify whether the detected anomaly requires further action or if it was falsely flagged.

To address this challenge, it is vital to establish a mechanism through which the human operator can provide feedback to the system regarding incorrect predictions. This feedback loop enables the system to learn from its mistakes and refine its future predictions. The specifics of how this learning is implemented vary depending on the particular anomaly detection method being used. By incorporating effective feedback loops, the system can continually improve its performance and reduce the incidence of false positives, thereby enhancing its overall accuracy and reliability.

Another form of learning involves creating new anomaly detection rules for previously unseen issues. These rules can be simple, like identifying log lines that should not exist at all or those that occur too frequently. Additionally, labeling log keys as anomalous can facilitate model retraining.

Since it is impossible to anticipate all potential problems that may arise in a system, the best approach to handling anomalies is to minimize their occurrence in the future. This begins with addressing the root cause of the anomaly, such as fixing software bugs. The next step is to implement preventive measures to avoid its recurrence. In software systems, automatic and manual testing is commonly used for this purpose. Anomaly detection rules serve a similar purpose, but instead of testing how the code works, they assess its behavior based on the logs. These rules can be applied even in testing environments to catch issues before they reach production.

The feedback loop, as illustrated in 5.1 of this thesis, includes two sources of anomaly detection: those reported by customers and those identified by automatic anomaly detection rules. All anomalies are reviewed by a human operator who determines their validity. If a detection is a false positive, the operator is guided to modify the rules to prevent similar occurrences in the future. If the anomaly is genuine and was automatically detected, no changes are required. However, if it was reported by a customer, a new rule should be added to enable the system to

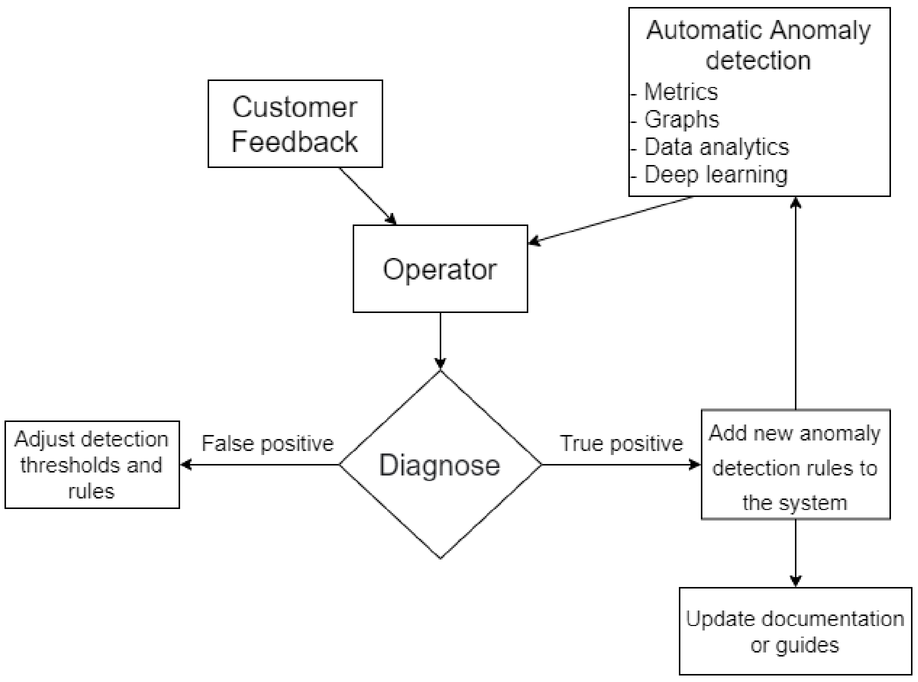
Customer Feedback
Automatic Anomaly detection
- Metrics
- Graphs
- Data analytics
- Deep learning
Operator
Diagnose
False positive
Adjust detection thresholds and rules
True positive
Add new anomaly detection rules to the system
Update documentation or guides

Figure 5.1: Diagram of anomaly detection feedback cycle.

detect similar anomalies in the future. This iterative feedback process enhances the effectiveness of the anomaly detection system.

5.2 Dataflow

To enable easy experimentation and building of anomaly detection rules, data accessibility is essential. The data goes through various processing stages before it can be used with the model. However, not all methods require extensive preprocessing. For example, simple text search does not need complex preprocessing, but parsing easily extractable information like timestamps or program names can be helpful.

Data processing can be parallelized to improve system performance. Each device has its own processor responsible for handling the logs it receives. The processor checks for anomaly detection rules and obtains model predictions. If any anomalies are found, they are added to the database. The operator can then check these anomalies to confirm if they are real anomalies, allowing adjustments to the model

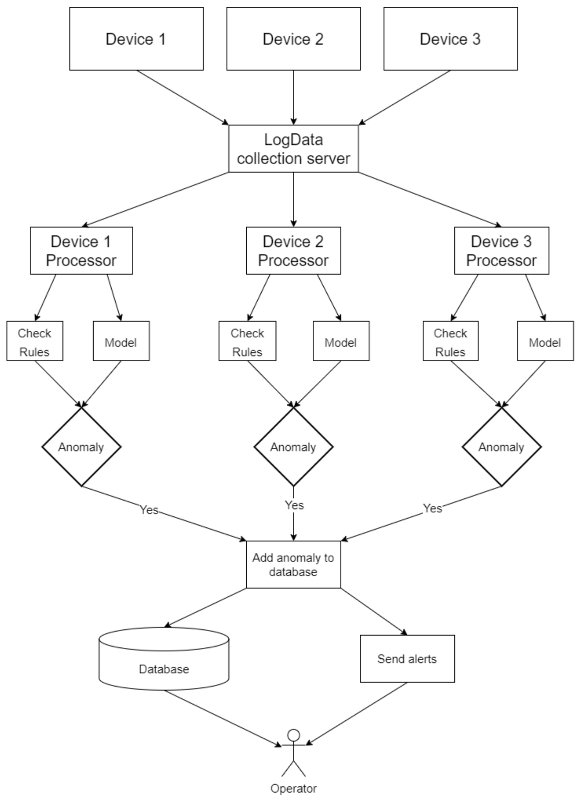

Figure 5.2: Dataflow graph how the data flows through the system

and detection rules accordingly (refer to Figure 5.2).

5.3 Alerts

In order to ensure quick response to anomalies, it is essential to establish a reliable notification system for the operators. The most common and convenient methods of notification include email and Slack. However, even if the alarms are sent correctly to the operators, there is a risk that they might be ignored. One significant reason for this is the excessive number of false positives generated by the system. When users receive numerous false alerts, it becomes challenging to take the system seriously. To address this issue, it is crucial to facilitate easy modification and testing of new rules and models. Some log-based monitoring software, such as Google Stackdriver, allow the addition of detection rules based on log lines. However, testing these rules can be difficult. The only viable method of testing is attempting to replicate the same anomaly in a test environment. The subsequent section will delve further into potential solutions for mitigating this problem.

5.4 Test Suites

In order to evaluate the efficacy of log anomaly detection rules, it is imperative to establish a mechanism that can feed historical anomaly data into the system and assess its ability to accurately identify it as anomalous. Consequently, this approach can be instrumental in the development of comprehensive test suites; these suites ensure that even in the event of modifications to the detection rules, the capacity to recognize past anomalies remains unimpacted.

Undertaking such a form of testing necessitates the existence of a method to replay past loglines. This would aid in confirming the satisfactory performance of the detection rules as per their initial design and intention. An improvement in the level of confidence in anomaly detection rules will resultantly reduce the incidence of false positives. Moreover, it mitigates the likelihood that the anomalies flagged by the system will be overlooked. Through these measures, the overall reliability and trustworthiness of anomaly detection rules can be significantly augmented.

5.5 User Interface

Delivering a user interface explicitly designed for human operators significantly streamlines the task of anomaly investigation, while also facilitating system status monitoring. This interface should incorporate a detailed list of open and resolved anomalies, coupled with a feature that empowers human operators to designate particular anomalies as examined.

Furthermore, this user interface should present an array of metrics for review, enabling the investigation of diverse aspects such as the volume of log data a specific device generates or the frequency of certain log line occurrences. A search function, designed to promote the discovery of log lines produced within a selected timeframe or by distinct devices, shall also enhance operational efficiency during anomaly inquiries.

The primary objective of the user interface is to offer users a platform through which they can formulate, edit, and observe rules for anomaly detection. The establishment of an intuitive process for deploying new rules and subsequently amending them when they result in an excess of false positives or negatives is vital. With this system, the operator is given leeway to tag individual log lines as anomalous; this information can then be used for refining the model, subsequently enhancing its capability in accurate anomaly detection.

5.5.1 Visualisations

Error Count

Figure 5.3 illustrates a common way to visualize errors in a software system. The y-axis represents the error count, while the x-axis represents time. A red line is drawn to indicate the chosen threshold that the system operator deems as an excessive number of errors, prompting them to be alerted.

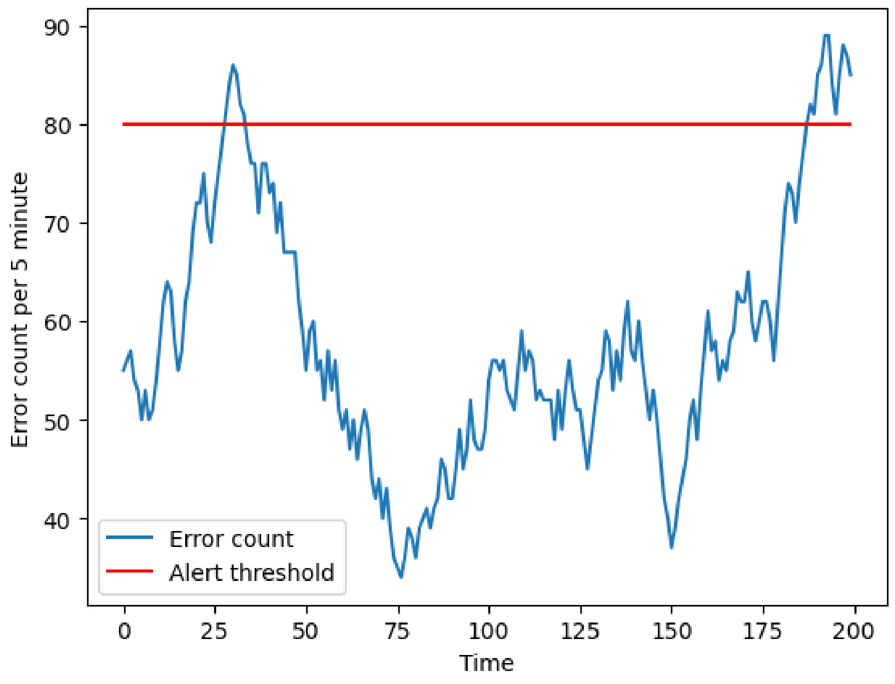

Figure 5.3: Error count with the alert threshold indicated by the red line. A higher value indicates more errors in the system.

It is uncommon for the threshold value to be set to zero because errors are expected to occur even under normal system functioning. This is especially true as the usage of the system increases. With a significant number of users, it is highly likely that someone will always encounter some form of error, even if it is relatively minor.

The interpretation of the error count itself can vary. In its simplest form, it could represent the number of loglines that match a specific text search, such as "HTTP request error." Alternatively, the count could be a combination of many different types of errors. More sophisticated error detection methods, such as those utilizing deep learning, may also contribute to the error count based on their own decision-making processes.

Furthermore, the error count is typically measured over a specific alignment period, which signifies the number of errors that have occurred within a given time frame, such as five minutes.

Outlier detection

When using clustering methods for anomaly detection, the PCA algorithm can be employed to reduce the dimensions of the data. This reduction allows the data to be plotted in a 2D scatter plot, where the proximity of points indicates their closeness. By visualizing the data in this way, it becomes easier to detect outliers or anomalous loglines that deviate from the cluster.

Figure 5.4 visually represents this concept, where the anomalous loglines are depicted as points that are farther away from the rest of the data points. This visualization aids in the identification and detection of anomalies.

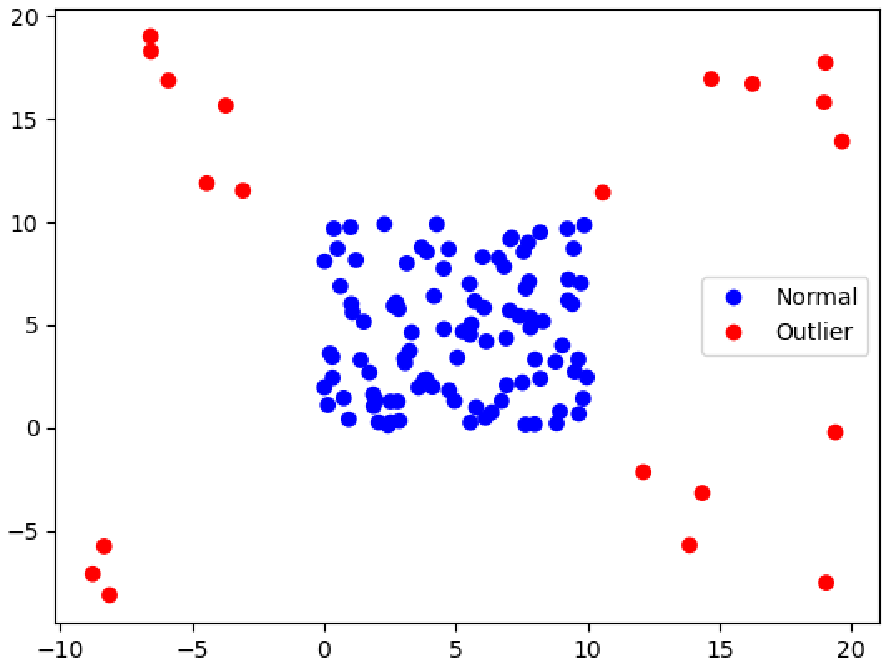

Figure 5.4: Scatterplot of normal and anomalous loglines.

Model confidence

Anomaly detection models, such as DeepLog [9], function by comparing their predictions for the next logline with the actual logline. DeepLog generates a probability indicating how likely the real next logline is based on its own prediction. This probability information can be utilized to visualize the model's confidence at any given point. When an operator is investigating a problem, they can identify areas in the log data where the model has lower prediction confidence. This could indicate potential issues in those areas.

Picture 5.5 displays the logline index on the x-axis and the model's confidence level on the y-axis. The red line represents the threshold chosen by the operator for alerts. The chart indicates that the model exhibits lower prediction confidence between logline indexes 45 and 65, suggesting the presence of an anomaly.

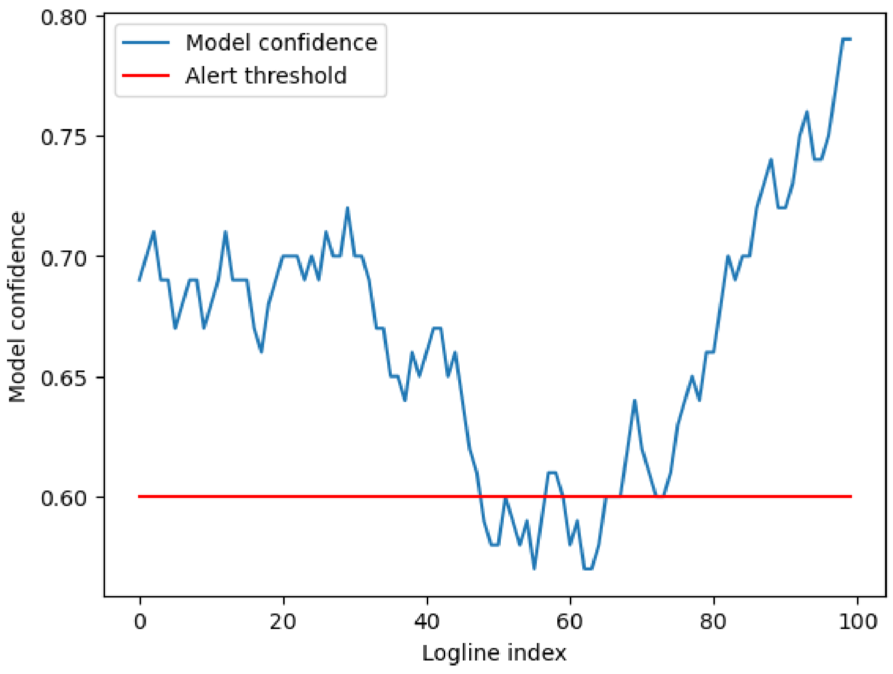

Figure 5.5: Model confidence for loglines with alert threshold indicated by the red line. A lower value means the model is less confident about its predictions.

6 Implementation

The selected model for this thesis's evaluation is DeepLog [9], notable for being the first method to leverage deep learning in anomaly detection. As elaborated upon in Chapter 3, DeepLog utilizes a Long Short-Term Memory (LSTM) network [19] to predict subsequent loglines.

In the original paper [9], two distinct neural network models were developed within the scope of DeepLog: one focusing on log key-based anomaly detection and the other on parameter value anomaly detection. Both models functioned to classify loglines as either normal or anomalous. However, for the purpose of this thesis, only the log key version has been utilized.

Training the DeepLog model involves several preparatory steps. This chapter aims to detail the process, including the collection and preprocessing of log data, as well as the model's training and evaluation stages.

6.1 Data Collection and Enhancement

6.1.1 Challenges of the Current Data Collection Strategy

The current strategy involves running a simple script to gather loglines from online devices. This method, albeit functional, confronts significant challenges in achieving optimal data collection and analysis.

A primary limitation is that this strategy only considers online devices, disregarding offline devices. This assumption of consistent device usage patterns can lead to inaccurate anomaly detection, especially for devices with irregular schedules.

Additionally, the existing method lacks an efficient redundancy management system, resulting in the collection of duplicate loglines. This leads to unnecessary storage and bandwidth consumption, with measurements suggesting that duplicates account for 40-50% of the collected loglines.

Finally, the issue of rapid logline generation by certain devices often results in some loglines being missed during the collection interval due to older logs' deletion, indicating a potential overproduction of unnecessary loglines.

6.1.2 Implementing an Agent Program

To address these issues, an agent program was developed and installed on the device during this thesis work. This agent is programmed to transmit logs to the server in a compressed format, either daily or at higher frequencies, thereby minimizing logline duplication and adjusting its sending interval as necessary.

This agent operates directly on the device and is better equipped to handle devices with non-standard usage patterns. Despite the devices' limited processing power, the agent could potentially be used to detect and report some anomalies directly thus, reducing latency in anomaly detection.

6.1.3 Improving Logline Storage

The present storage method involves saving loglines in single compressed files for each device, with filenames indicating the collection date and device number for quick referencing. However, this approach leads to an extensive volume of files and doesn't effectively handle duplicate logfiles within a single compressed file.

Improvements could include ensuring that each file accurately reflects the loglines from the day of its creation and creating an index file with paths for all loglines as they accumulate, to facilitate more efficient data retrieval. Implementing compression techniques is essential because according to measurements compression results in ten-fond reduction in logfile size, significantly reducing storage requirements.

The development and implementation of the agent program during this thesis work represent a substantial stride towards enhancing the efficiency and accuracy of the logline collection and analysis process.

6.2 Preprocessing

6.2.1 Log Data Structure

The system logs can exist in one of three forms:

1. **Raw Text Format:** The original form of the logs as retrieved from the device. An example of a raw logline is as follows:

   ```
   "2022-04-02T16:45:49.515058+03:00 pt10ex_90762 bussi
       -2.2.90.1: trying to do something"
   ```

   The logline starts with a timestamp, followed by the device type and device number. Subsequently, the program (and occasionally its version) that generated the logline is mentioned. In some instances, system services that are part of the operating system generate these loglines. The logline ends with a free text portion that lacks a specific structure. These raw loglines can be transformed into a more structured format using Regular Expressions (RegExr).

2. **Preprocessed Format:** In this format, loglines have been structured using RegExr, yielding the following arrangement:

   This format is a significant improvement over the raw text as it allows for easy filtering or sorting by columns. While rudimentary anomaly detection based

Table 6.1: Metadata parsed from raw log data.

Parameter	**Value**
Date	2022-04-02T16:45:49
Device type	pt10ex
Device number	90762
Program	bussi
Version	2.2.90.1
Logtext	trying to do something

on logline frequency is possible in this format, our primary interest lies in the specific events created by a program.

3. **Tokenized format:** This format is numerical representation of logline and is used as input for the deep learning model. Next section will go through this more closely.

6.2.2 Tokenization

Tokenization, a process of converting text into numbers for machine learning models, can be accomplished via various methods. Discussed in Chapter 3, these methods are as follows:

1. **Source Code Analysis:** This approach involves analyzing the source code to identify original logging statements and using this information to convert raw log data into numerical form. However, parsing this information can be difficult due to the variety of programming languages used and the involvement of various external libraries. Also, full access to the source code may not always be feasible, particularly with logs from Linux devices running multiple programs in parallel.

2. **Online Parsing:** Methods such as spell [1] and drain [2] provide alternatives. They use the longest common subsequence or parse tree to deduce the original logging statement, subsequently creating a log template for each distinct logging statement. These methods pioneered the strategy of online parsing, which allows for loglines to be processed in a streaming fashion as they arrive, rather than in batches.

3. **WordPiece with LanoBERT:** The WordPiece tokenizer, in conjunction with the LanoBERT architecture [4], represents a more recent and robust approach. Unlike Spell and Drain, which convert loglines into single numbers, limiting their ability to capture complex relationships between loglines, WordPiece can extract finer-grained information.

   For instance, consider the logline "sleeping for 50s." WordPiece tokenizes it as "['sleeping', 'for', '50', '##s']," demonstrating its capability to recognize even the units of numbers, a crucial aspect for effective anomaly detection. After tokenization, these tokens can be converted into numerical representations, resulting in a vector such as "[5777, 2005, 2753, 2015]."

   However, it's important to note that the use of WordPiece tokenizer may lead to larger models and require additional training resources due to the vector representation as opposed to a single number.

In this thesis, the Drain parser is the method of choice. It was preferred due to its efficiency over Spell while maintaining a similar level of accuracy [2] also since similar method was used in DeepLog as tokenizer.

Implementation

The drain parser was implemented using the Rust programming language for this thesis. Rust's memory safety feature enables the creation of reliable programs that perform efficiently. Similar projects, such as [20], have utilized Rust for the implementation of tokenizers used in large transformer models.

The parser's parse tree was constructed following the methodology delineated in [2], incorporating three layers. The initial layer is a hash map that organizes log sequences into varied-length groups based on their token count post-tokenization. The second layer is another hash map categorizing the first token of each log sequence. A number or a token that matches a predetermined processing rule is represented by an asterisk, preventing the tree from excessive expansion. The final layer of the parse tree, the log cluster, performs the last stage of processing to convert the logline into the log template. Each log cluster may contain numerous log templates, which are assessed to identify the highest-scoring one. If a template's score reaches or exceeds a certain threshold, the parser selects and modifies it as necessary. These modifications could involve updating the pre-established template to align with the structure of the newly parsed log message. For instance, the parser might introduce new placeholders or adjust existing ones to create more precise log templates that capture more of the variation in system log messages. If no suitable match is identified, a new log template without variables is generated.

Performance

The inaugural version of the drain parser exhibited suboptimal performance characteristics. As shown in the chart 6.1, the parsing speed drastically declined from over 300,000 loglines per second to fewer than 10,000. This decrease significantly hinders the parser's practicality, as it needs to parse roughly 50,000 loglines per second to keep pace with the live system's log production. The loglines underwent no preprocessing before being parsed by the drain parser, which resulted in the creation of excessive, unnecessary log templates, thereby slowing down the parser. When functioning correctly, the parser is anticipated to create a finite number of different

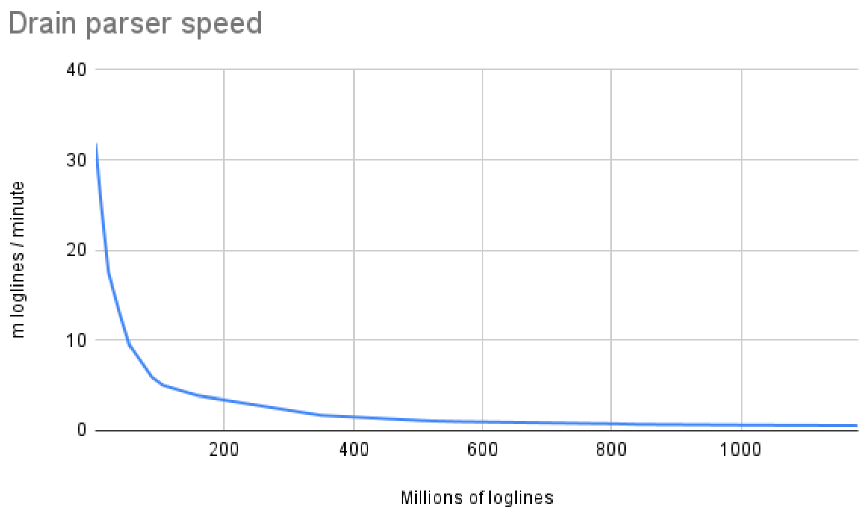

Figure 6.1: Parsing speed of the log parser without preprocessing as the number of parsed loglines increases.

loglines, given the finite number of logging statements in the code. Preprocessing to remove elements like IP addresses, as was done in the original drain paper [2], could alleviate this issue.

Upon deeper analysis, it was apparent that the parser occasionally generated new templates incorrectly. For instance, one issue occurred when a logline payload began with a timestamp in the format "2023-10-5", causing the parser to create a new subtree for every logline with a timestamp. Instead, these loglines should be converted into variables, symbolized by an asterisk, as they form part of the log printing statement. Additional problems surfaced where improper tokenization led to unnecessary log template creation. A logline like

```
"src/ntp.c:decode_msg() offset=0.060462 delay=0.193699"
```

would be parsed inaccurately, with "offset=0.060462" becoming part of the log template. This improper parsing led to the creation of numerous unnecessary log templates that impeded the parser's performance. By adding "=" to the parser's delimiter list, this issue can be rectified, allowing the parser to correctly interpret the

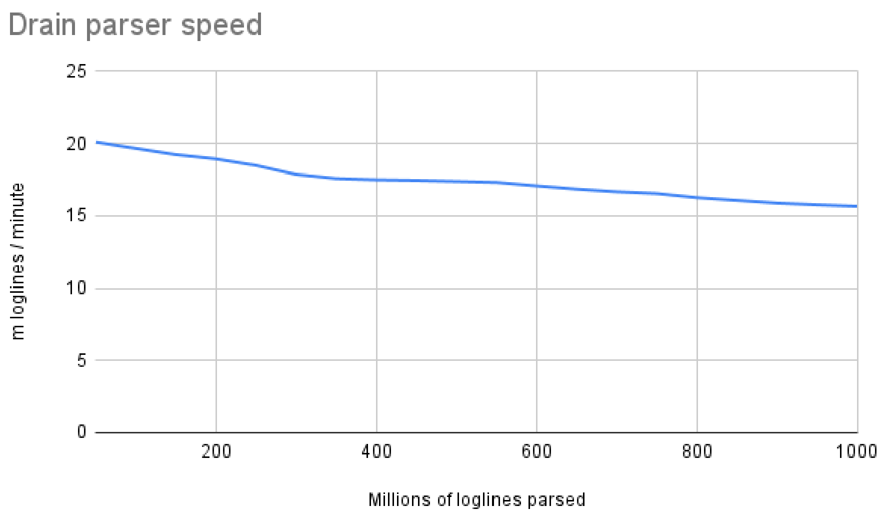

Figure 6.2: Parsing speed of the log parser after preprocessing as the number of parsed loglines increases.

previous example as:

```
"src/ntp.c:decode_msg() offset=* delay=*"
```

After implementing several filtering rules and modifications, the parser's speed significantly increased, as evidenced by the chart in 6.2.

The parser's speed could potentially be enhanced further by introducing parallelism. Given the parser's reliance on parse trees and the fact that leaf nodes are a performance bottleneck, parsing logs in parallel at the leaf node level might accelerate processing. However, parallelism would also increase the parser's complexity, which might negate any performance benefits gained.

6.2.3 Dataset

To train the model, the generated data needs to be transformed into a format that facilitates training and evaluation. Table 6.1, as previously described, details the type of metadata extracted from the raw log data. This metadata includes elements like timestamp, device number, and program, which play a critical role in the formulation of the dataset. The primary objective of the DeepLog model is to learn authentic logline sequences from the log data.

In the context of this thesis, loglines are generated by 1000-2000 devices, each running a custom Linux operating system, and all of the log data is consolidated into a single file on the device. Each device operates numerous programs concurrently. On the server side, all logs are aggregated into a single stream. Parsing log sequences from this combined stream would result in fragmented, less realistic log sequences, diminishing the predictive performance of the model. Therefore, it is vital to sort logs based on date and group them by device number and program. This method preserves more accurate logline sequences.

Each program may also have parallel-executing components that independently generate loglines. This could pose a problem, but grouping at this level would require more domain or language-specific knowledge, which is beyond the scope of this thesis.

Upon constructing log groups, they are used to formulate log sequences, as shown in the format below:

Table 6.2: Dataset structure for encoded loglines

Label	**Encoded Loglines (Features)**					**Real Next Logline**
1	3946	3946	3946	3946	3946	4411
1	3946	3946	3946	3946	4411	4411
0	3946	3946	3946	4411	4411	374
...	...	...	...	...	...	...

The example presented in table 6.2 begins with a logline label: 1 indicates an anomaly, while 0 signifies normalcy. The remaining numbers represent the logline sequence, with each number corresponding to a log template number produced by the drain parser. In this case, the window size is 5, indicating that the first five numbers describe the logline sequence. The final number, which the model is trained to predict, is used to evaluate the model by assessing its accuracy in prediction. If the model incorrectly predicts the next logline, then the log sequence is considered anomalous. The following section provides a more comprehensive explanation of this training and evaluation process.

6.3 Model Training and Evaluation

DeepLog model was implemented for this thesis with PyTorch library using python programming language.

6.3.1 Training Objective and Procedure

The dataset, as formulated in the previous section, was employed as the training source for the DeepLog model. The structure of each line in the dataset, representing a single log sequence, its anomaly status, and the actual next logline, is suited to the goal of the DeepLog model as described in Chapter 3 – predicting the next logline in a given log sequence. The anomaly status of each logline in the dataset was determined by searching for keywords whose presence would suggest that the current logline might be anomalous.

The DeepLog model was architected as follows:

- An input layer to receive the encoded logline sequences.
- Two LSTM layers, each with 64 hidden units, constructed to capture the temporal dependencies within the data.
- A fully connected (dense) layer that maps the LSTM outputs to the final prediction.

Training of the model utilized the cross-entropy loss function, a commonly adopted choice for classification tasks where the model is tasked with predicting one outcome from a set of possibilities. The Adam optimizer was employed for training, known for its effectiveness in various sequence prediction tasks. [21]

Over a span of 6 epochs, the model was trained with a batch size of 1000. To foster good generalization and to mitigate overfitting, the training data was subjected to random shuffling before each epoch. The loss for each epoch was computed as the average loss per instance in the training dataset.

Evaluation of the model's performance focused on the accuracy in predicting wether logline is considered to be anomalous or not. Further discussion on additional metrics and validation strategy will be provided in the ensuing sections.

6.3.2 Performance Evaluation

The F-score is a common measure for evaluating the performance of binary classification models. It provides a more comprehensive perspective on real-world applicability. Although DeepLog is a logline prediction model rather than a binary classification model, it can be utilized for anomaly detection. By predicting what the subsequent logline might be and considering any deviations from this prediction as anomalies, the F-score calculation can be adapted.

Model Evaluation Metrics

DeepLog is an anomaly detection system for log sequences. Given a vocabulary size denoted as W, it determines the probability for each logline $I \in W$ in a log sequence S. Higher numbers imply a higher likelihood, suggesting that the specific logline could be the next one in the sequence. The output of this process is a set of probabilities $L = \{P(I)|I \in W\}$, where $P(I)$ represents the predicted probability for each logline I. By arranging this set L in descending order, the most probable candidates are at the top, with the least probable at the bottom. This output can be utilized for anomaly detection in two distinct ways.

Method 1: Using Position Index The first approach relies on the loglines' position index within L. As a sequence of loglines can be succeeded by a number of valid loglines, the top 'n' candidates from L are selected, where 'n' is a configurable parameter. The determination of whether the next real logline belongs to this group will decide if it's anomalous.

- **True Positives** (tp): Loglines marked as anomalous that do not fall within the top 'n' candidates.
- **False Positives** (fp): Instances where the actual logline isn't among the predictions yet isn't marked as anomalous.
- **False Negatives** (fn): Loglines tagged as anomalous but are part of the top 'n' candidates.

Method 2: Using Probability Values The second method involves using the probability values generated by the DeepLog model directly. These values are scaled to a 0-1 range and a threshold value denoted by 't' is used to determine if a logline is anomalous. The value of 't' can be adjusted based on the balance needed between different types of errors.

- **True Positives**: Instances labeled anomalous but with a prediction probability below the threshold 't'.
- **False Positives**: Loglines not labeled as anomalous, but with prediction probabilities lower than 't'.
- **False Negatives**: Loglines labeled as anomalous but with prediction probabilities higher than 't'.

Once the tp, fp, and fn values have been accumulated, they can be used to compute the F1-score with the formula $F_1 = \frac{2tp}{2tp+fp+fn}$. The F1-score is a common measurement of a model's performance, but it has limitations, such as the implicit assumption that false positives and false negatives carry the same cost.

Challenges

The primary challenge in this domain is the scarcity of well-labeled log data. Often, only a handful of loglines are known to be anomalous, which limits the efficiency of the anomaly detection process. Two strategies can be employed to address this issue.

Leveraging Known Anomalous Loglines The first strategy involves seeking out known anomalous loglines within the logs. By identifying these loglines, it's possible to extract sequences that contain anomalies. This approach, while limited by the availability of known anomalous loglines, can provide valuable insights for the model.

Artificial Anomaly Injection The second strategy proposes the artificial injection of anomalies into normal sequences. By labeling these injected instances as anomalous, the model can gain additional data for learning and prediction. While this approach deviates from the pure log data, it can enhance the model's capability to recognize and predict anomalies, making the testing more representative of real-world scenarios. This approach must be applied cautiously to avoid overfitting on artificially injected anomalies and underperformance on real-world anomalies.

Dataset

For evaluation, we utilize a dataset consisting of 1,400,000 loglines, formatted as described in 6.2. Out of these, 1,300,000 loglines are categorized as normal, while

100,000 are labeled as anomalous. The anomalous classification is based on specific keyword matches within the log text. For clarity, anomalous loglines are identified by direct comparison with a predefined list of keywords:

- "Left overs"
- "channel closed"
- "Resource temporarily unavailable"
- "Could not import device key"
- "No connection to cEMV device"
- "Device is changed; this should not happen"
- "Error while validating request: request/body/0/location/latitude must match format 'float'"

For instance, any sequence containing the log text "channel closed" is marked as anomalous.

Test Configurations

This dataset is applied for training and testing through four different configurations:

1. Anomalies are removed from the entire dataset. Then, 5% of normal loglines in the *test dataset* alone are transformed into anomalies by assigning a random value to the real label and then marking it as anomalous. This test is expected to produce the best score but also represents the most unrealistic scenario, given that the anomalies are artificially generated and excluded from the training.

2. Anomalies are removed from the dataset just as in the first case. However, this time, anomalies are artificially injected into the *entire dataset*, making them part of both the training and the testing data. Although this situation may yield poorer results compared to the first case, it provides a somewhat more realistic depiction because the anomalies become part of the training set.

3. In the third case, anomalies are only removed from the training set and not the test set. There are no additional anomalies injected into the test set. The test set comprises 1,300,000 normal loglines and 100,000 anomalous loglines. This represents a more realistic test, as real anomalies are used.

4. In the fourth scenario, no modifications are made to the dataset prior to training or testing. This is the most realistic approach, but also likely to yield the worst results.

By specifying the context of anomaly injection in each case, we can better understand the differences in their setup and consequently, their outcomes.

Baseline

In the field of machine learning, it's common to compare the performance of our models with certain basic strategies or simple models, often referred to as baselines. A baseline provides a point of reference to understand whether the model is giving meaningful outputs. In the context of anomaly detection, where the goal is to identify data points that are significantly different from the norm, our baselines are methods which use simple strategies to predict whether a data point is an anomaly.

Random Guess The Random Guess method is used as a foundational baseline for anomaly detection. Instead of having patterns or information within the data analyzed, a prediction on whether a data point is an anomaly is made purely based on randomness. Its main function is to offer a basic benchmark against which more sophisticated models can be measured.

In this method, a guess threshold is established. For every data point in the dataset, a random number is generated. If this number falls below the threshold, the data point is labeled as an anomaly.

During the evaluation, true positives, false positives, and false negatives are accumulated as data points are processed. Precision, recall, and F1 score are then computed using these results. Due to the method's inherent randomness, a lower performance in its metrics is expected. Consequently, it is anticipated that any comprehensive anomaly detection model would exceed this basic baseline in terms of performance.

6.4 Results

The performance of DeepLog was assessed across various test cases to evaluate its efficacy in different scenarios. These evaluations can be found in Table 6.3.

From the table, it's evident that DeepLog achieves the highest F-scores in Cases 1 and 2. However, in Case 3, where real anomalies are introduced into the test set, there is a noticeable decline in performance when compared to Cases 1 and 2. For Case 4, the performance shows a minor improvement over the random baseline. The transition from Case 1 to Case 2 exhibiting only a marginal decline in performance, while a more pronounced drop is observed from Case 3 to Case 4, suggests distinct variations in the datasets or conditions between these cases. This could happen because certain anomalous sequences are overrepresented in the dataset which means the model will overfit to those anomalies. From Case 4 results it can be seen that anomalies have lower average position than normal loglines which also supports overfitting idea.

	Case 1	**Case 2**	**Case 3**	**Case 4**
Probability F-score	**0.9439**	**0.9314**	**0.7321**	**0.1761**
True positives	63705	64359	57247	59956
False positives	**6468**	**9029**	**217**	522665
False negatives	1103	449	41690	38501
Threshold	0.6000	0.6000	0.2000	0.7000
Average anomaly probability	0.5164	0.4816	0.2885	0.7128
Average normal probability	0.8189	0.8205	0.6801	0.7214
Position F-score	0.8387	0.8564	0.6722	0.0557
True positives	**64605**	**64627**	60797	6421
False positives	24652	21491	21147	**125801**
False negatives	**203**	**181**	38140	92036
Top-k	15.0000	15.0000	15.0000	1.0000
Average anomaly position	2667.1010	2644.2716	2492.6879	0.6081
Average normal position	5.1945	4.9815	2.7450	2.4921
Median anomaly position	2672.0000	2631.5000	4018.0000	0.0000
Median normal position	0.0000	0.0000	0.0000	0.0000
Random F-score	0.0945	0.0942	0.1304	0.1295
True positives	53161	53017	**80885**	**80261**
False positives	1007154	1007884	1060306	1060385
False negatives	11647	11791	**18052**	**18196**

Table 6.3: Results of the experiments

Further insights into the performance of DeepLog can be gathered from Figures 6.3 and 6.4. In these figures, the results of threshold-based and position-based anomaly detection, across different k and threshold values, are depicted. In Cases 1, 2, and 3, superior performance by DeepLog is consistently displayed. However, in Case 4, only a marginal surpassing of the random baseline is observed.

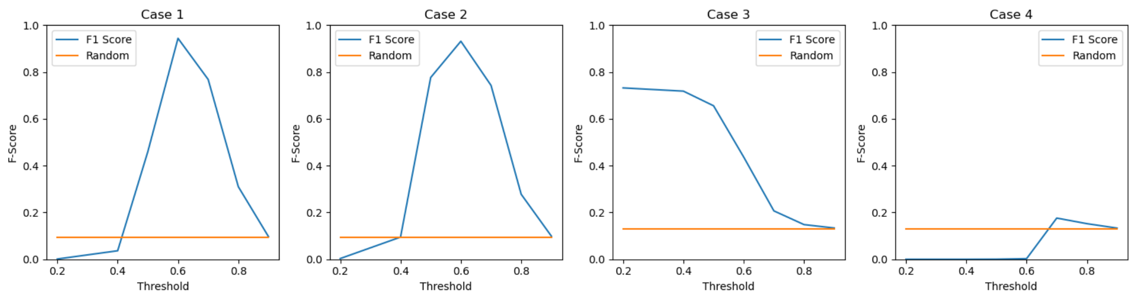

Figure 6.3: Threshold-based F-score calculation.

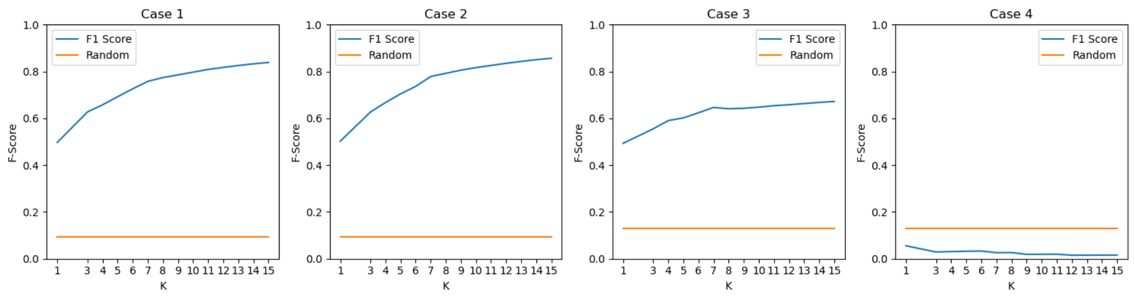

Figure 6.4: Position-based F-score calculation.

In Figure 6.5, the prediction probabilities for subsequent loglines, classified as normal or anomalous, are shown. A distinct separation in predictions is observed in Cases 1 and 2. Some separation in Case 3 is noted, while in Case 4, a blending of normal and anomalous predictions is evident.

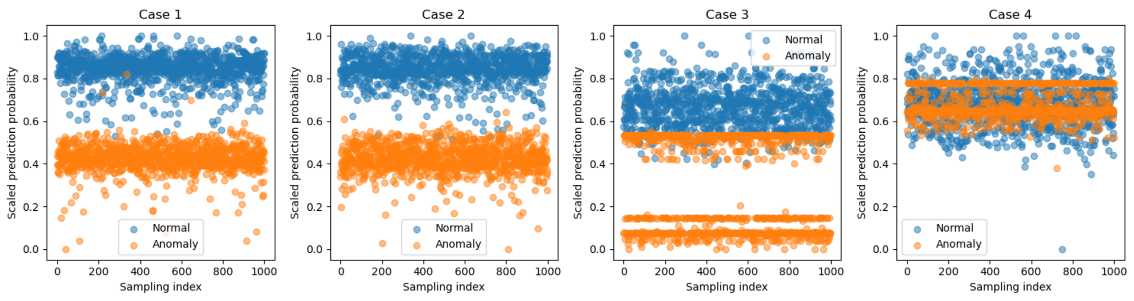

Figure 6.5: Scatterplot of prediction probabilities for normal and anomalous loglines.

In Figure 6.6, the sorted prediction positions of a sample of 1000 data points are showcased. Here, anomalous loglines are color-coded in orange, while the normal ones are in blue. A position of 0 implies that the model recognizes the logline as the most likely subsequent entry. As the positions increase, the model's confidence in that prediction diminishes.

From the chart, the distinction in the model's confidence between normal and anomalous loglines in Cases 1 and 2 is quite pronounced. However, in Case 3, there's a mix, though there are certain anomalous predictions that the model is less sure about. In contrast, Case 4 reveals a muddled distinction, with normal loglines even exhibiting reduced prediction confidence. Upon closer examination, it appears that only normal loglines receive a higher prediction position, as opposed to those close to 0. This observation further reinforces the earlier hypothesis regarding overfitting.

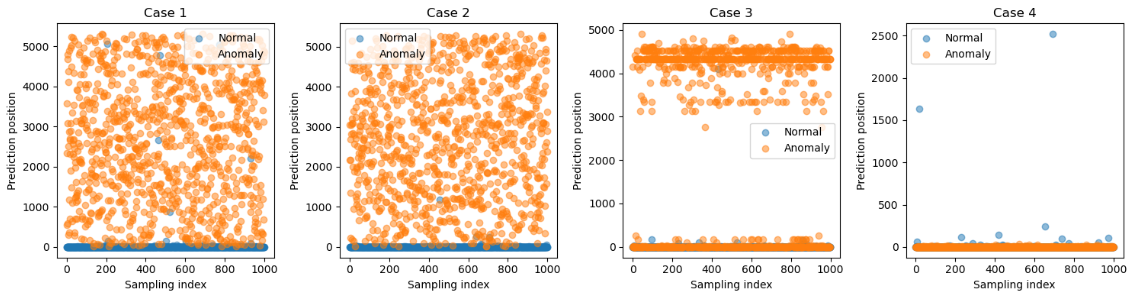

Figure 6.6: Scatterplot of relative positions for normal and anomalous loglines.

7 Analysis

This chapter aims to present and discuss the results obtained in the previous chapter. Firstly, we will provide a detailed overview of the data analysis and findings. Subsequently, we will compare our results with other studies from the literature, highlighting similarities and differences. Finally, we will identify potential limitations of the current method and suggest some possible ways to improve it in the future.

7.1 Results Breakdown

7.1.1 Parsing and Preprocessing Analysis

This study utilized the DeepLog model for log key anomaly detection and the Drain parser. The initial usage of Drain without any modifications led to excessive log templates and poor parsing performance with larger log datasets. However, with the implementation of several filtering rules, the parser's performance notably improved.

A significant challenge faced was maintaining robustness amidst changing logging statements. Two proposed solutions in the literature included using fixed-size semantic vectors that describe loglines or bypassing the parser altogether and applying minimal preprocessing with regular expressions followed by tokenization.

7.1.2 Anomaly Detection Analysis

DeepLog's training and evaluation process involved four unique training and testing configurations. In the first scenario, a dataset free from anomalies was used for training while a 5% anomaly rate was introduced into the test dataset. This allowed for the assessment of the model's anomaly detection performance on previously unseen anomalies.

In the second scenario, anomalies were introduced throughout the entire dataset, evaluating the model's performance when trained on data containing anomalies. The third scenario entirely excluded artificial anomalies, with naturally occurring anomalies removed from the training set. This tested the models performance on previously unseen logs.

In the fourth scenario no other modifications were made than standard shuffling and data splitting tehniques to the data. This scenario tested the models performance in situation where no anomalies were removed from the training which could be the starting point where model is being used for the first time.

DeepLog's evaluation involved one baseline method: random guessing. The primary performance metric was the F-score, and scatter plots of model predictions were used to visually assess the separation between normal and anomalous loglines.

7.1.3 Performance Comparisons

The original DeepLog paper achieved F-scores (0.96 on HDFS, 0.98 on OpenStack 1, and 0.97 on OpenStack 2). The implementation in this study achieved F-score of 0.94 and 0.93 in cases 1 and 2 which are pretty close to the orginal paper. However these case were using artificially injected anomalies which they are not completely realistic. Third case gave F-score of 0.73 which while being better than random guessing is significantly less than other tests.

In the third scenario, the model achieved an F-score of 0.73 which while being higher than random guessing is still not considered to be very good result.

In the fourth scenario, DeepLog displayed subpar performance, achieving an F-score of 0.13. DeepLog did not show any clear distinction between normal and anomalous loglines as can be seen from scatterplot 6.5 and it barely can surpass the random guessing baseline.

7.1.4 Interpretation of Results

In the fourth scenario DeepLog did not perform well however it should also be thought as baseline performance when just starting to use the model for anomaly detection in realworld deployment since it is probably feasible to remove known anomalous sequences from training data. This would potentially increase the model performance overtime when more true positive cases are identified and then removed from training data. Also the simplistic anomaly-labeling methods used in this thesis might not effectively pinpoint all present anomalies in the logs.

The log sesequences in the dataset used in the sixth chapter were constructed by gy hrouping based on device number, and program and then sorted by timestamp. This approach aimed to maintain the sequential integrity of the logs. However, this straightforward technique leaves room for enhancement to assure better sequence retention.

Upon delving deeper into the DeepLog predictions, it was observed that they primarily constituted a single anomaly type. The sequences deemed anomalous typically included just one or two types of repeated loglines. Moreover, numerous sequences presented difficulties for human interpretation. Consider the following instance of a logline perceived as normal:

```
GATEINFO set_busy_state for node * value *
STATUSREQUEST_SENDER Node * has never answered to status requests.
STATUSREQUEST_SENDER Node * has never answered to status requests.
STATUSREQUEST_SENDER Node * has never answered to status requests.

STATUSREQUEST_SENDER Node * has never answered to status requests.

STATUSREQUEST_SENDER Node * has never answered to status requests.

STATUSREQUEST_SENDER Node * has never answered to status requests.

STATUSREQUEST_SENDER Node * has never answered to status requests.

STATUSREQUEST_SENDER Node * has never answered to status requests.

STATUSREQUEST_SENDER Node * has never answered to status requests.
```

In this instance, the model's top predictions were:

```
can't open '/dev/ttymxc0' No such file or directory

qml bindNode/Node * Exists true Online * open *

STATUSREQUEST_SENDER Node * has never answered to status requests.

plugins/ofono.c * /gemalto_1 * *

qml pt10 Line * * required false

plugins/ofono.c * /gemalto_3 * *

qml pt10 * *

plugins/ofono.c * /gemalto_2 * *

/source/flexcard-core/src/flexcard-class.cpp * * * *

DOOR_ACCESS_DUMPER Redump access lists after * s
```

The actual subsequent logline was:

```
STATUSREQUEST_SENDER Node * has never answered to status requests.
```

This indicates that the real next logline was the model's third likeliest prediction.

A typical anomalous sequence in the log data appeared as follows:

```
[ERROR] Execute method failed channel closed

[ERROR] Execute method failed channel closed

qml *
qml *

[ERROR] Publishing new state failed channel closed

[ERROR] Execute method failed channel closed

qml *

[ERROR] Publishing new state failed channel closed

[ERROR] Publishing new state failed channel closed

[ERROR] Publishing new state failed channel closed
```

And the model's top predictions were:

```
pdo link message *

qml *

qml * * * * * *

qml * state from PT11eX *

qml * * *

qml * * * * *

qml NO MESSAGE

qml linestop open * ajosarja * linenr * linename * driver *

qml infocontrol *

/data/send_request finished
```

Since the actual next logline didn't exist among the model's predictions, the sequence was deemed anomalous. In this instance, the model had a prediction confidence of 0.35, with the position of the real next logline being 5014 out of 8739. This example derived from case 3, wherein known anomalies were excluded from the training set.

If there is many of this type of anomalies in the training set like in case 4 the model will learn how to predict them. This paradoxically means that model will learn not to predict anomalies since anomaly detection is based on the fact that anomalous sequences should give lower prediction performance. These observations about the data corroborate the hypothesis presented in Chapter 6, suggesting that the training data is predominantly influenced by a single type of anomaly, leading to model overfitting to these anomalous sequences.

7.1.5 Key Takeaways from Results

As previously mentioned, Cases 1 and 2 demonstrated excellent performance. However, since they employ synthetic anomalies, they more closely resemble ideal conditions wherein anomalous sequences signifigantly differ from the normal ones. In Case 3, which incorporated real anomalies, DeepLog exhibited potential. Yet, the performance in Case 4 was very bad. Initially, it appeared that DeepLog failed in the most realistic scenario, Case 4. But, as highlighted earlier, this should be interpreted more as a baseline for model performance with a specific dataset. Anomaly detection in real-world settings would be a continuous process. At the outset, the model might yield numerous false positives and negatives, but over time, as training data is refined, the model's proficiency at predicting normal log sequences improves. This enhancement implies that its capability to predict anomalous sequences should diminish. By excluding new true positive sequences from the training data, model performance will incrementally improve.

With this methodology, the model can achieve very good results on log data that remains consistent over time. Nevertheless, many software systems are updated frequently nowadays. It's thus essential for anomaly detection systems to adapt to evolving log structures. Part of issues with changing log data arises from the use of parsers like Drain [2]. A limitation with Drain is that two distinct log templates lack semantic similarity. Consequently, even a minor alteration to a logging statement would render it entirely unfamiliar to the model. This challenge was also pinpointed in [10] and [3]. Their proposed solution was the incorporation of off-the-shelf word vectors. This approach enables unique log lines to be depicted with word vectors, rendering the representation more resilient to minor changes, especially when the semantic essence of two unique log lines remains consistent. However, utilizing word vectors designed for general text might not adequately capture the nuances of log data. Although log data predominantly uses English, it doesn't always conform to standard language conventions. [4] explored training a BERT model both from scratch and with prior natural language pre-training. Their results showed a modest F-score increase in the BGL dataset and a slight decrease with the HDFS dataset, indicating that depending on the log data structure, models trained with natural language might either excel or underperform.

Despite these challenges, DeepLog could still be applicable in real-world scenarios, provided there's adequate training dataset manipulation and frequent retraining. A potential strategy would be to monitor the creation of new log templates and sequences in the operational system, update the training data accordingly, remove true positives, and subsequently retrain the model. Given DeepLog's relatively simple and light weight architecture, periodic retraining is plausible with modest computational capabilities.

To the best of my understanding, there has been absence of research focusing on production systems with evolving log data. Zhang et al. [10] investigated the robustness of the LogRobust model by introducing anomalies into the system. Their findings indicated only a marginal decline in the F-score when the anomaly injection rate reached 20%. Nonetheless, it's essential to underscore that the use of artificial anomaly injections may not fully capture the intricacies of real-world scenarios. The potential of this research area is evident, but it also presents challenges. For instance, many academic institutions might not have access to production systems, which restricts their ability to gather authentic data. Historically, the majority of literature reviews in this domain have relied on established datasets like HDFS and BGL.

7.2 Performance

The performance of log anomaly detection method is very important when considering their practical applicability. The value of an optimal model decreases significantly if it cannot deliver timely results. This section delves into the performance analysis of the Drain parser and DeepLog and suggests potential improvements.

7.2.1 Performance of the Drain Parser

Initially, the Drain parser's performance fell short of expectations. However, upon incorporating several preprocessing and filtering rules, its performance witnessed a significant boost.

In a setting where all devices cumulatively produce around 10,000 log lines per second, this thesis' Drain parser implementation demonstrates commendable efficiency. It can parse more than 200,000 log lines per second, even after processing an count of 1 billion log lines. Such performance levels showcase the Drain parser's capacity to handle the system's total log line production effectively.

7.2.2 Performance of DeepLog

Using an RTX 2070 GPU and an i7-8700k CPU, DeepLog processes about 50,000 log lines per second, with a batch size of 1,000. This rate far exceeds the 10,000 log lines per second threshold, illustrating that DeepLog can feasibly process each log line generated by the system even with relatively inexpensive hardware.

When tested without GPU acceleration in the same environment, the processing rate dropped to around 3,500 log lines per second. This drop shows the GPU's crucial role in speeding up the inference. Although there is a requirement for more specialized hardware, such as GPUs, the cost of a system with this computational capability is relatively low. If we were to use a state-of-the-art model like LanoBERT

[4], the costs would likely be much higher. Subsequent sections of this chapter will delve deeper into this topic.

7.2.3 Potential Performance Improvements

The nature of deep learning model inference allows for parallel processing, creating opportunities for speed enhancement when necessary. This capability ensures scalability and adaptability to increasing data volumes, emphasizing the practicality and effectiveness of the approach.

Performance can also be 'improved' indirectly by reducing the number of log lines processed. For example, exclusively processing logs from devices utilized for production could reduce the volume of logs the model must handle. There are devices used solely for testing purposes that could be excluded. Another strategy might involve prioritizing certain devices deemed more important based on specific metrics. Improving logging behavior could also reduce amount of logs produced and contributing to overall improved software maintainability. Additionally, there might be new log groups used exclusively for debugging, such as the 'debug' log level. These could be excluded from model training since they are likely used when an anomaly has already been detected in the system.

7.2.4 Performance Insights

While the current pipeline's speed appears sufficient for parsing every new log line the system generates, future models with more complex architectures might be slower. However, this section has proposed methods to reduce the number of log lines requiring parsing and introduced parallelization to enhance performance, offering potential solutions to such anticipated challenges.

7.3 Cost

The trajectory of automatic anomaly detection appears to be shifting towards increasingly complex neural network architectures. An unavoidable consequence of this trend is the escalating computational demands these models place on their infrastructures. Consequently, these escalating demands also come with increased financial cost and an ever-growing requirement for specialized hardware.

In contrast, simpler solutions employing rule-based systems or classical data science methodologies such as Principal Component Analysis (PCA) primarily require a relatively swift CPU and substantial memory. The main function of these requirements is to ensure timely processing of incoming loglines.

An example of a more resource-friendly model is DeepLog, discussed in previous sections of this thesis. This model is capable of processing loglines at an rate of 50,000 loglines per second on a standard gaming PC. The total cost of implementing this system is likely to fall below 2,000€, and could be reduced to even under 1,000€ by opting for second hand parts.

However, it is important to note that while cost-effective, DeepLog also carries its share of limitations as discussed earlier. Adopting a more advanced method could invariably require significantly greater resources, both in terms of computational power and cost. Thus, one must strike a delicate balance between the sophistication of the method and the feasibility of its implementation.

7.3.1 Computational Cost of State-of-the-Art Methods

LanoBERT, as discussed in Chapter 3, is a BERT model trained from scratch, which allows the model to be more generalizable as it can learn the vocabulary of the log data. However, training a BERT model from scratch can be quite expensive. The original BERT paper by Devlin et al. [14] utilized 64 TPU chips for training, with the pretraining process taking 4 days to complete. One TPU v4 chip has a peak compute capacity of 275 teraflops [22], meaning that 64 chips amount to $64 * 275 = 17,600$ teraflops. However, the TPU v2, which was the highest version available at the time, had a peak compute capacity of 45 teraflops per chip. Consequently, the compute used for training the original BERT was $64 * 45 = 2,880$ teraflops over 4 days. According to the GCP pricing calculator [23], this would cost €6,367 to pretrain. If TPUv4 were used, only 10 chips would be needed, but since one TPU card has 4 chips, 12 chips would be utilized. This would cost €3,417, which is nearly half the cost of using v2 TPUs. Additionally, these calculations were made in 2023, so there may be some price adjustments to TPUv2 from 2018. When using NVIDIA A100 GPUs from LambdaLabs [24], one A100 GPU can produce 312 teraflops according to [25], which means that $2,880/312 = 8.97$, or 9 GPUs, would be needed to pretrain the model. This would cost €864, making it over 7 times less expensive to pretrain a BERT model on LambdaLabs compared to using v2 TPU from Google Cloud.

Researchers have also explored ways to reduce the cost of training BERT from scratch. In [26], the authors aimed to find a way to pretrain BERT models within an academic budget. They simulated an academic budget by limiting the training time to 24 hours and the hardware to a single low-end deep learning server, which they considered equivalent to 1 day with 4 RTX 3090 GPUs. Even in 2021, when the paper was published, the cost of such hardware would be over €10,000. An alternative would be to rent GPU computation from a cloud service. LambdaLabs offers a GPU virtual machine with an NVIDIA A100 40GB VRAM, 30 vCPUs, 200 GB RAM, and 512 GB storage for $1.10 per hour [24]. In [26], the authors stated that training a BERT model with similar performance to the original would take about 2.4 days using one A100 GPU. This means that pretraining a BERT model from scratch with this level of computation would cost approximately €63, which is quite affordable. To match the training speed of 4 RTX 3090 GPUs, two A100 GPUs would be needed, doubling the price to €126. To reach the price of a €10,000 machine, one could conduct nearly 80 pretraining runs. This means that a BERT model could be pretrained every week for 1.5 years before owning the hardware becomes more cost-effective. This calculation does not take into account electricity consumption and other maintenance needs of the machine. Of course, owning the hardware also offers the benefit of using it for inference tasks, which, in this case, means anomaly detection and fine-tuning the model. However, fine-tuning and inference are less computationally expensive than pretraining. Therefore, the key factors to consider when deciding between renting and purchasing hardware are the frequency of model pretraining and the amount of log data produced for anomaly detection.

While the previous analysis provides a cost overview, it's essential to consider the model's performance and viability. To my knowledge, no pretrained models have been specifically designed for log anomaly detection. A study by Lee et al. [4] found that the performance of a model pretrained using normal log data varies between log anomaly detection datasets. In some cases, more complex datasets showed slight performance improvements, while in others, performance was slightly reduced.

7.3.2 Cost of Implementation

In a previous subsection, calculations indicated that using lightweight methods like DeepLog can make the costs of a Deep Learning system relatively affordable. Even when employing state-of-the-art models and rented hardware, the expenses remain manageable. However, it's crucial to also factor in the implementation and maintenance costs when assessing the total financial implications of such systems.

Constructing a system from scratch often means that the human effort involved outweighs the actual hardware expenses. There are off-the-shelf solutions available, such as Google Cloud Stackdriver, but they might not possess advanced deep learning capabilities. While these existing systems provide fundamental functionalities like log collection, search features, and visualizations, they're most suitable for software operating in standard cloud environments. When the software is running in a custom setup, as in the context of this thesis, leveraging existing systems effectively can be challenging. Moreover, the costs associated with platforms like Google Stackdriver can escalate quickly, especially when managing vast amounts of logs.

8 Conclusions

Chapter 3 examined the evolution of detecting anomalies from log data. It did dive deep into various techniques, noting a shift towards methods similar to those used in language technology. This thesis also addresses practical considerations: the type of computer systems needed, cost-effective methods suitable for small businesses, ways to display detected anomalies, and an evaluation of whether these automated tools truly save time in system maintenance. This chapter aims to provide answers to the key questions posed at the beginning of the thesis.

8.1 Answers to Research Questions

RQ1: What kind of methods are currently being used for log anomaly detection.

Chapter 2 presented a comprehensive examination of the historical approaches to log anomaly detection. The content was organized chronologically, offering a clear understanding of the field's evolution. Initial efforts in log anomaly detection relied on rule-based approaches, which demanded significant human effort for maintenance. Subsequently, researchers sought methods to automate log parsing into a format that could facilitate the identification of temporal associations between log lines by algorithms, even for errors unknown to operators beforehand.

Unsupervised log data parsing techniques, such as Drain [1] and Spell [2], were introduced to eliminate the need for additional domain knowledge. However, Drain necessitates some basic preprocessing rules for optimal performance, as discussed in Chapter 3. Early deep learning methods like DeepLog [9] demonstrated high prediction accuracy when logs were stable but failed with evolving log statements. Consequently, subsequent methods incorporated NLP concepts to address this issue.

By employing fixed-size semantic vectors generated from standard word vectors in large NLP models, [3] these approaches aimed to make anomaly detection models more resilient to changing log statements. The fixed-size semantic vectors were designed to capture the semantic meaning of log lines without relying on their specific structure. However, the unique vocabulary of log data may impact model accuracy.

Cutting-edge methods like LanoBERT [4] train the large language model BERT from scratch or initialize it with natural language then finetuning it with logdata, eliminating the need for a separate log parser to preprocess log data. As LanoBERT learns the log data vocabulary, it exhibits greater robustness to changes.

This thorough investigation revealed the various methods used in log anomaly detection, showcasing state-of-the-art techniques and their limitations. This information is essential when choosing the appropriate method for a specific use case.

RQ2: What kind of infrastructure does machine learning methods need to be used for anomaly detection.

Chapter 5 of the thesis discusses the infrastructure needed to implement automatic anomaly detection methods. It explains that to use the methods described in Chapter 3, a pipeline must be built. This pipeline takes into account the collection of logs from devices, processing of log data, and using it for anomaly detection. The reported anomalies are then presented to operators, who can react to them.

To ensure that the system is accurate, it is important to incorporate proper feedback loops into the pipeline. This allows for continuous monitoring and improvement of the system's detection accuracy over time.

Chapter 5 provides a comprehensive guide on the requirements of the pipeline, including the necessary hardware and software components, data processing techniques, and the importance of data quality and management. Additionally, it outlines the roles and responsibilities of stakeholders involved in the development and maintenance of the pipeline, including operators, developers, and data analysts.

Overall, Chapter 5 emphasizes the importance of building reliable infrastructure for automatic anomaly detection.

RQ3: What kind of methods are economically viable to use in a small company.

Rule-based systems generally do not require extensive computational resources or specialized hardware. They do, however, necessitate sufficient CPU processing power to handle incoming logs. The required processing power is directly proportional to the volume of logs being processed. Chapter 6 demonstrated that the Drain parser implemented for this thesis can process over 200,000 log lines per second, after processing more than 1 billion log lines. This performance was achieved on a workflow that is mostly single-threaded, which indicates potential for further improvements. The results were obtained using an AMD Ryzen 9 5900X 12-Core Processor, indicating that the overall cost of a system capable of processing logs at this scale is relatively low. DeepLog models used achieved speed of 50 000 loglines per seconds far exceeds worst case scenario of 10 000 loglines per second. This was achieved with relatively inexpensive hardware.

Even state-of-the art Deep Learning models like LanoBERT [4] can be trained from scrach with relatively low costs around €800 and with some optimisations even closer to €100 which is quite afortable. However these models will likely need to be retrained sometime and inference requires some resources too.

In conclusion, deep learning techniques are viable for smaller companies, but they do require some understanding of deep learning technologies and their application. Although modern deep learning libraries are relatively user-friendly, developers may not be familiar with them and may need time to learn. Additionally, using these methods effectively requires building a log analysis and anomaly detection pipeline, like the one described in Chapter 5. The developer effort needed to construct these systems might be more costly than the methods themselves.

RQ4: How to visualise anomalies found from the system so that the human maintainer can investigate the anomaly efficiently.

Chapter 5's section on user interface provides valuable insights into enhancing operator efficiency when investigating anomalies through visualization. This section offers a range of illustrative examples, showcasing the potential visualizations for effectively deciphering anomalous logs. The chapter presents three distinct types of plots, each serving a unique purpose in visualizing these logs.

The first plot employs a simplistic representation to visualize the occurrence of error loglines over time. By mapping specific points in time along with the corresponding count of detected error loglines, this plot offers a straightforward overview of temporal anomalies.

Moving forward, the second plot adopts a scatter plot approach to differentiate between normal and outlier points based on their spatial distribution. This plot effectively portrays the deviation of outlier points from the norm by measuring the distance between them. An outlier, in this context, is defined as a data point significantly distant from the clustering of other loglines. Consequently, this scatter plot aids in pinpointing loglines that substantially deviate from expected behavior.

The final visualization within this section provides an intricate insight into the model's prediction confidence for individual loglines. By measuring the likelihood of a specific logline appearing within a sequence, this visualization becomes particularly invaluable in scenarios where operators are already aware of anomalies within the system. The visual representation of prediction confidence serves as a guide, directing operators to sections within logfiles that hold the potential to uncover anomalies.

RQ5: Does automatic log anomaly detection decrease the overall worktime required for system maintenance.

Automatic log anomaly detection can significantly decrease the overall worktime required for system maintenance. One way to achieve this is by using regex to identify known anomalous loglines, such as those indicating invalid code execution paths or segmentation faults. This method relies on past experiences and language features like panics to detect anomalies.

A more advanced approach involves counting specific types of loglines detected through regex and using these counts to determine if an anomaly is present. For example, monitoring the number of failed HTTP requests within a certain period can help identify anomalous system behavior. Cloud providers like GCP support these types of methods, which can detect anomalies based on single occurrences of specific loglines.

For more complex anomaly detection, tools like logsurfer can be used, which offer a sophisticated rule system that allows for dynamic rule addition and deletion, as well as context support. However, this method requires system administrators to define every possible anomaly they want to detect, which can be time-consuming and may not cover unknown anomalies.

Machine learning-based approaches can address this limitation by detecting anomalies without prior knowledge. Although these methods require more preprocessing and pipeline setup, they generally require less human effort for rule creation.

In summary, automatic anomaly detection can reduce the worktime required for system maintenance, but it does require some form of pipeline, including log collection, parsing, and analysis, to be effective. By implementing these automated methods, system administrators can focus on more critical tasks and improve overall system quality.

8.2 Key Findings

In this study, the performance of DeepLog was evaluated using log data sourced from real-world systems, particularly Linux-based payment terminals. An in-depth investigation was conducted into the essential factors for effective anomaly detection. It was found that even basic Deep Learning models could yield satisfactory results when provided with high-quality data. Nonetheless, attention must be given to ensuring the model does not overfit to specific anomalous sequences prevalent in the training data. A proposed solution to this challenge is to remove any new anomaly from the training data as it's identified, which would prevent model overfitting. This strategy could potentially enhance the system's performance over time. However, this study did not assess the efficacy of this method with changing log data, highlighting a potential area for subsequent research.

Another notable finding from this study is the affordability of training even the most advanced anomaly detection models.

References

[1] M. Du and F. Li, “Spell: Streaming parsing of system event logs”, in *2016 IEEE 16th International Conference on Data Mining (ICDM)*, IEEE, 2016, pp. 859–864.

[2] P. He, J. Zhu, Z. Zheng, and M. R. Lyu, “Drain: An online log parsing approach with fixed depth tree”, in *2017 IEEE international conference on web services (ICWS)*, IEEE, 2017, pp. 33–40.

[3] S. Huang, Y. Liu, C. Fung, *et al.*, “Hitanomaly: Hierarchical transformers for anomaly detection in system log”, *IEEE Transactions on Network and Service Management*, vol. 17, no. 4, pp. 2064–2076, 2020.

[4] Y. Lee, J. Kim, and P. Kang, “Lanobert: System log anomaly detection based on bert masked language model”, *arXiv preprint arXiv:2111.09564*, 2021.

[5] S. E. Hansen and E. T. Atkins, “Automated system monitoring and notification with swatch.”, in *LISA*, Monterey, CA, vol. 93, 1993, pp. 145–152.

[6] J. E. Prewett, “Analyzing cluster log files using logsurfer”, in *Proceedings of the 4th Annual Conference on Linux Clusters*, Citeseer, 2003.

[7] R. Vaarandi, “A data clustering algorithm for mining patterns from event logs”, in *Proceedings of the 3rd IEEE Workshop on IP Operations & Management (IPOM 2003)(IEEE Cat. No. 03EX764)*, Ieee, 2003, pp. 119–126.

[8] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs", in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 117–132.

[9] M. Du, F. Li, G. Zheng, and V. 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*, 2017, pp. 1285–1298.

[10] X. Zhang, Y. Xu, Q. Lin, *et al.*, "Robust log-based anomaly detection on unstable log data", in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 807–817.

[11] S. Kabinna, C.-P. Bezemer, W. Shang, M. D. Syer, and A. E. Hassan, "Examining the stability of logging statements", *Empirical Software Engineering*, vol. 23, pp. 290–333, 2018.

[12] A. Joulin, E. Grave, P. Bojanowski, M. Douze, H. Jégou, and T. Mikolov, "Fasttext. zip: Compressing text classification models", *arXiv preprint arXiv:1612.03651*, 2016.

[13] A. Vaswani, N. Shazeer, N. Parmar, *et al.*, "Attention is all you need", *Advances in neural information processing systems*, vol. 30, 2017.

[14] J. D. M.-W. C. Kenton and L. K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding", in *Proceedings of naacL-HLT*, vol. 1, 2019, p. 2.

[15] P. He, J. Zhu, S. He, J. Li, and M. R. Lyu, "An evaluation study on log parsing and its use in log mining", in *2016 46th annual IEEE/IFIP international*

conference on dependable systems and networks (DSN), IEEE, 2016, pp. 654–661.

[16] Y. Wu, M. Schuster, Z. Chen, *et al.*, "Google's neural machine translation system: Bridging the gap between human and machine translation", *arXiv preprint arXiv:1609.08144*, 2016.

[17] L. Bottou, "Large-scale machine learning with stochastic gradient descent", in *Proceedings of COMPSTAT'2010: 19th International Conference on Computational StatisticsParis France, August 22-27, 2010 Keynote, Invited and Contributed Papers*, Springer, 2010, pp. 177–186.

[18] W. L. Taylor, ""cloze procedure": A new tool for measuring readability", *Journalism quarterly*, vol. 30, no. 4, pp. 415–433, 1953.

[19] S. Hochreiter and J. Schmidhuber, "Long short-term memory", *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[20] H. Face, *Tokenizers*, `https://github.com/huggingface/tokenizers`, Accessed on February 24, 2023, 2022.

[21] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization", *arXiv preprint arXiv:1412.6980*, 2014.

[22] *Tpu system architecture | cloud tpu documentation*, `https://cloud.google.com/tpu/docs/system-architecture-tpu-vm#tpu_v4`, accessed 2023.

[23] *Google cloud pricing calculator*, `https://cloud.google.com/products/calculator#id=acebd692-807c-47bc-bed1-3f8080086229`, accessed 2023.

[24] *Lambda labs gpu cloud*, `https://lambdalabs.com/service/gpu-cloud`, accessed 2023.

[25] *Nvidia a100 datasheet*, `https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet-us-nvidia-1758950-r4-web.pdf`, accessed 2023.

[26] P. Izsak, M. Berchansky, and O. Levy, "How to train bert with an academic budget", *arXiv preprint arXiv:2104.07705*, 2021.