CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Real World Implementation of LLM-based Log Anomaly Detection

Exploring the feasibility of training-free approaches

Master's thesis in Computer Science and Engineering

DANTE SHINTARO COMETTI

DIOGO LOPES DIONISIO

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2024

Master's thesis 2024

Real World Implementation of LLM-based Log Anomaly Detection

Exploring the feasibility of training-free approaches

DANTE SHINTARO COMETTI
DIOGO LOPES DIONISIO

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2024

Real World Implementation of LLM-based Log Anomaly Detection
Exploring the feasibility of training-free approaches
DANTE SHINTARO COMETTI
DIOGO LOPES DIONISIO

Supervisor: Emilio Jorge, CSE
Advisor: Carl Svensson, Ericsson
Examiner: Ashkan Panahi, CSE

Master's Thesis 2024
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Gothenburg, Sweden 2024

Real World Implementation of LLM-based Log Anomaly Detection
Exploring the feasibility of training-free approaches
DANTE SHINTARO COMETTI
DIOGO LOPES DIONISIO
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

The complexity of systems have escalated to the point where automated techniques leveraging machine learning methods have become indispensable for log anomaly detection. In this project, carried out in collaboration with Ericsson, the feasibility of employing training-free approaches was explored. We implemented the RAPID method for log anomaly detection, which uses a small dataset of "normal" logs and a pre-trained DistilBERT model to classify unseen log lines by measuring distances between their representations, requiring no training or fine-tuning. The implementation was then modified to accommodate a dataset of logs provided by Ericsson, achieving an F1 score of 0.94 and correctly classifying 49991 out of 49993 anomalies. Additionally, we attempted fine-tuning the pre-trained DistilBERT model on a separate dataset comprised of normal log lines; however, this failed to yield significant improvements. The performance of the RAPID method was also compared to a baseline implementation, which utilizes bag-of-words representations. While the baseline method performed extremely well on both the Ericsson and BlueGene/L (BGL) datasets, it fell slightly short on the Ericsson dataset experiencing a drastic loss of performance in detecting anomalies. The results obtained from these experiments, coupled with the research conducted in the log anomaly detection space, highlight the importance of result replication in this field, the limitations of the F1 metric, challenges and trade-offs of fine-tuning models, the effectiveness of simple statistical methods versus LLMs, and the environmental and ethical concerns of using large models in machine learning.

Keywords: Logs, Anomaly Detection, BERT, Representations, Fine-tuning

Acknowledgements

We want to thank our supervisors, Emilio, Carl and Alex, for the continued support and guidance throughout our thesis project.

Dante and Diogo, Gothenburg, 2024-06-10

Contents

List of Figures

List of Tables

1

Introduction

Log systems are critical system components, ensuring operational stability and security across various industries. They collect, store, and analyze data about the operations of computer systems, applications, and network devices, providing a detailed record of events that occur over time. Telecommunication companies, such as Ericsson, produce a substantial amount of logs, generated by the monitoring systems of various hardware components. A significant portion of system failures are detected through the analysis of logs, making this information vital for troubleshooting issues and understanding system performance. Thus, precise anomaly detection is crucial for maintaining the integrity and reliability of IT infrastructures.

Log anomaly detection can be performed manually or by employing simple automations, such as rule matching [1]. However, due to the increasingly complex nature of computer systems, there is a need for faster and more precise methods [2]. Multiple machine learning (ML) approaches have achieved impressive results on publicly available log dataset [3]–[5].

In this project, we will implement and experiment with the RAPID method [6], or "Training-free Retrieval-based Log Anomaly Detection with PLM (Pre-Trained Language Models) considering Token-level information". The RAPID framework treats logs as natural language, extracts representations using a pre-trained language model and then implements a retrieval technique to compare test logs with the most similar normal logs [6].

The main research areas that are going to be explored in the following chapters are:

- **Adapting the RAPID method to a log dataset provided by Ericsson.**
- **Implementing a baseline method.**
- **Exploring the value of model fine-tuning.**
- **Developing and comparing multiple approaches.**

1.1 Thesis Outline

In this section, we will provide an overview of the thesis by going through its structure and briefly summarizing each chapter.

- **Related Work:** In this chapter, we outline the most common challenges encountered in log anomaly detection and provide a description of the best performing and relevant ML-based log anomaly detection approaches
- **Background:** The necessary theoretical knowledge needed to understand this thesis is included in this chapter. We begin by going into the structure of logs and log anomaly detection, to then detailing the machine learning algorithms and concepts which are mentioned throughout this report.
- **Data Description and Analysis:** In the Data Description and Analysis chapter we provide an in-depth analysis of the datasets employed to train and test the various detection approaches.
- **Methods:** In this chapter, we showcase the crucial frameworks and pre-processing techniques which were fundamental for the successful completion of the project. In addition, we go through the fine-tuning techniques and the experimental setup which yielded the best results.
- **Results:** The most relevant results gathered throughout the project are presented in this chapter. Covering the results yielded by the fine-tuned models, a comprehensive comparison of every approach and some additional experiments designed to provide some insights on the obtained results.
- **Discussion:** We present some remarks and considerations that stem from the results, by exploring the importance of result replication and the need for comprehensive classification results. Moreover, we assess the value of model fine-tuning as well as discussing environmental and ethical concerns.
- **Conclusion:** In this final chapter we include a brief summary of the thesis project and some final remarks.

2

Background

This chapter includes the theoretical concepts necessary to understand the methods and techniques utilized throughout the project. It starts by briefly explaining what anomaly detection entails and continues by introducing several statistical and machine learning concepts, providing some crucial background to the methods outlined in the following chapters.

2.1 Log Anomaly Detection

Anomaly detection is a long-standing problem that has been widely researched in the computer science literature. It aims to measure normal or acceptable behavior and identifies any deviations from this norm as potentially intrusive [7].

2.1.1 Logs

Logs are crucial records of activities within a computer system, capturing details such as user access, file operations, and system changes. They are essential for troubleshooting, security, and auditing, providing evidence of malicious activity and alerting administrators to potential issues. Logs are used to monitor system performance, diagnose errors, and generate security and compliance reports. Typically stored in system directories or application-specific folders, logs help identify abnormal patterns and suspicious behavior, ensuring the reliability and security of digital systems [8]. An example log from the IBM [9] documentation is included in listing 2.1.

```
03/22 08:51:01 INFO   :...locate_configFile: Specified
  ↪ configuration file: /u/user10/rsvpd1.conf
```

Listing 2.1: Example Log entry for configuration file location

2.1.2 Anomaly Detection

The necessity for effective log anomaly detection has grown alongside the increasing reliance on digital systems and networks. In the past, manual inspection of logs was a feasible approach to identifying anomalies. However, the exponential growth in data volume and complexity has rendered manual methods impractical. The sheer

amount of data generated by modern systems makes it impossible for human experts to keep up with and accurately analyze logs, leading to errors and missed anomalies [2]. To address these challenges, automated methods based on computer algorithms have been developed. These methods leverage Machine Learning (ML) and Artificial Intelligence (AI) to analyze large volumes of log data efficiently [2].

2.2 Artificial Intelligence and Machine Learning

AI is a broad field of computer science that aims to create systems capable of performing tasks that would typically require human intelligence. These tasks include understanding natural language, recognizing patterns, solving problems, and making decisions [10]. Machine Learning (ML) is a subset of AI that focuses on the development of algorithms and statistical models that enable computers to perform tasks without being explicitly programmed for them [11]. In essence, the concept of learning in ML involves iteratively refining a model's predictive accuracy. The process begins with data collection and preprocessing, leading to the selection or extraction of features. Next, the model is trained on the data to enable pattern recognition. Following training, the model is tested on a separate dataset to verify its performance before being deployed for predictions on new unseen data [11]. Furthermore, ML can be divided into distinct categories based on the available data and the specific task at hand, including supervised learning, unsupervised learning, semi-supervised learning, and reinforcement learning [12].

2.2.1 Supervised Machine Learning Methods

In supervised learning, the model is trained on a dataset consisting of input-output pairs, known as labeled data. The goal of supervised learning is to learn the mapping or relationship between the input variables (features) and the output variable (target) from the labeled data. The model uses this learned relationship to make predictions or decisions when new unseen data is presented [12]. In the context of anomaly detection within supervised ML methods, feature extraction methods and supervised learning algorithms are two essential steps in establishing the necessary framework for detection [13].

In the following sections, a description of the most common machine learning concepts utilized in automatic log anomaly detection is presented, with particular attention to the supervised methods employed in the literature and throughout this project.

2.3 Feature Extraction

Feature extraction involves employing natural language processing (NLP) methods such as bag-of-words (BOW), Word2vec, or Term Frequency-Inverse Document Frequency (TF-IDF) to transform unstructured logs into structured data, consequently allowing for the extraction of semantic information from said logs [13]

2.3.1 Bag of Words (BOW)

The Bag of Words model [14] represents text data by counting the frequency of each word in a document, disregarding grammar and word order. Mathematically, given a document D consisting of words $w_1, w_2, \ldots, w_n$, the BOW representation is a vector $V = [v_1, v_2, \ldots, v_n]$ where each element v_i corresponds to the count of word w_i in D [15].

Table 2.1: Example Bag of Words Representation

Word	**cat**	**cute**	**small**	**dog**
small dog	0	0	1	1
cute cute cat	1	2	0	0
cute dog	0	1	0	1

2.3.2 Word2Vec

Word2vec [16] is a model that can convert words in sentences into vector expressions. By training, the processing of log content is simplified to vector operations in K-dimensional vector space, and the similarity in vector space can be used to represent log semantics [13]. Word2Vec uses either Continuous Bag of Words (CBOW) or Skip-Gram model. In CBOW, the model predicts the current word based on the context, and in Skip-Gram, it predicts surrounding words given the current word. Mathematically, given a sequence of training words $w_1, w_2, \ldots, w_T$, the objective of the Skip-Gram model is to maximize the average log probability:

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{-c\leq j\leq c, j\neq 0}\log p(w_{t+j}|w_t)$$

where c is the context window size, and $p(w_{t+j}|w_t)$ is defined using the softmax function:

$$p(w_O|w_I) = \frac{\exp({v'_{w_O}}^{\top} v_{w_I})}{\sum_{w=1}^{W}\exp({v'_w}^{\top} v_{w_I})}$$

Here, v_w and v'_w are the "input" and "output" vector representations of word w, and W is the vocabulary size.

2.3.3 Term Frequency-Inverse Document Frequency

TF-IDF [17], or Term Frequency-Inverse Document Frequency, is a statistical method used to assess the importance of a word for a file set or one of the files in a corpus. The importance of a word increases proportionally with the number of times it appears in one file, but it also decreases inversely with the frequency it appears in the

corpus. In other words, we can calculate the TF-IDF of all the nouns that appear in the document. The larger the TF-IDF, the higher the distinction between the noun and the document, and the more TF-IDF values [13]. The specific Mathematically formulas are as follows.

$$TF(t,d) = \frac{\text{Number of times term } t \text{ appears in document } d}{\text{Total number of terms in document } d}$$

$$IDF(t,D) = \log\left(\frac{\text{Total number of documents in corpus } D}{\text{Number of documents containing term } t}\right)$$

$$TF - IDF = TF(t,d) \times IDF(t,D)$$

Overall, the utilization of methods, such as Bag of Words (BOW), TF-IDF and Word2Vec, allow for natural language logs to be transformed into numerical vectors. Through this embedding process, the semantic content of the logs is encoded into numerical form, allowing for efficient handling and processing of the data in the subsequent training and classification tasks.

2.4 Learning Algorithms

Once feature extraction has been performed, the following step is to employ learning algorithms to perform the detection. These learning algorithms can range from traditional methods such as Naïve Bayes [18] to more sophisticated ones like artificial neural networks (ANNs). In recent decades, there have been advances in ANNs [19], especially deep neural networks (DNNs), that have proven to outperform previously state-of-the-art methods [13].

2.4.1 Naïve Bayes

Naïve Bayes is a classification algorithm based on Bayes' theorem with the assumption of independence between features. Despite its simplicity, Naïve Bayes has shown remarkable performance in various real-world applications [18]. The algorithm works by calculating the probability of a given instance belonging to each class based on the feature values. Mathematically, it can be expressed as follows:

Given a feature vector $\mathbf{x} = (x_1, x_2, ..., x_n)$ representing n features, and a set of classes $\mathcal{C} = C_1, C_2, ..., C_k$, Naïve Bayes calculates the probability of each class given the features using Bayes' theorem:

$$P(C_k|\mathbf{x}) = \frac{P(\mathbf{x}|C_k) \cdot P(C_k)}{P(\mathbf{x})}$$

Where:

- $P(C_k|\mathbf{x})$ is the posterior probability of class C_k given the feature vector $\mathbf{x}$.
- $P(\mathbf{x}|C_k)$ is the likelihood of observing the feature vector $\mathbf{x}$ given class C_k.

- $P(C_k)$ is the prior probability of class C_k.
- $P(\mathbf{x})$ is the probability of observing the feature vector $\mathbf{x}$.

The Naïve Bayes assumption of feature independence simplifies the likelihood calculation:

$$P(\mathbf{x}|C_k) = P(x_1|C_k) \cdot P(x_2|C_k) \cdot ... \cdot P(x_n|C_k)$$

This assumption allows Naïve Bayes to efficiently compute the class probabilities even with a large number of features, making it a valuable tool in classification tasks such as anomaly detection [13].

2.4.2 Feedforward Neural Networks

A feedforward neural network is an Artificial Neural Network (ANNs) inspired by the structure and functioning of the human brain's biological neural networks [20]. ANNs consist of interconnected nodes, called neurons, organized in layers. Each neuron receives input signals, processes them, and produces an output signal. ANNs are capable of learning complex patterns and relationships in data through a process called training, where the network adjusts its parameters to minimize the difference between the predicted and actual outputs. This is done through backpropagation by calculating the gradient of the loss function with respect to the network's weights. This gradient information is then used to update the weights in a direction that minimizes the loss, effectively improving the network's performance over time [21].

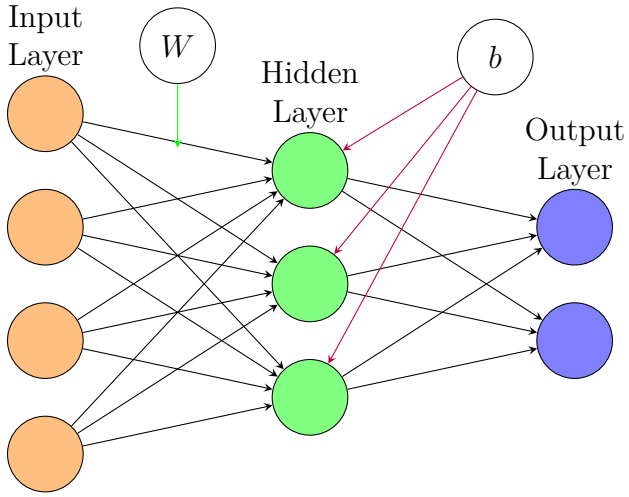


Figure 2.1: Artificial Neural Network (ANN) example

In a feedforward neural network, information flows in one direction, from the input layer through one or more hidden layers to the output layer. The output of each neuron in a layer is determined by applying an activation function to the weighted

sum of its inputs. Activation functions are responsible for introducing non-linearity into the network's computations. They determine whether a neuron should be activated or not based on the weighted sum of its inputs. Essentially, it controls the output of each neuron, enabling neural networks to learn complex patterns and relationships in the data [22]. Mathematically, the output of a neuron j in layer l can be expressed as:

$$a_j^l = \sigma\left(\sum_{i=1}^{n^{l-1}} w_{ji}^l a_i^{l-1} + b_j^l\right)$$

Where:

- a_j^l is the activation of neuron j in layer l,
- w_{ji}^l is the weight of the connection between neuron i in layer $l-1$ and neuron j in layer l,
- a_i^{l-1} is the activation of neuron i in layer $l-1$,
- b_j^l is the bias of neuron j in layer l,
- σ is the activation function.

Commonly used activation functions are the sigmoid, ReLU (Rectified Linear Unit), and tanh (hyperbolic tangent) functions [22]. The sigmoid function, also known as the logistic function, compresses the input values between 0 and 1, making it suitable for binary classification problems where the output needs to be interpreted as probabilities [23].

With the advancement of deep learning, which involves the training of Artificial Neural Networks (ANNs) with numerous hidden layers, deep neural networks (DNNs), such as RNNs, have started to outperform traditional machine learning methods in terms of both performance and scalability [24].

2.4.3 Long Short-Term Memory (LSTM)

An LSTM takes a sequence of data as input (such as time series or text) and outputs a prediction or classification based on the information from the entire sequence. It is a type of recurrent neural network (RNN) architecture, and also classified as a deep neural network (DNN), was specifically designed to address the vanishing gradient problem and capture long-range dependencies in sequential data [25]. Traditional RNNs suffer from the vanishing gradient problem, where gradients diminish exponentially as they propagate back through time during training, making it difficult to learn from distant past information [26]. LSTMs improve on this issue by introducing a specialized memory cell and gating mechanisms.

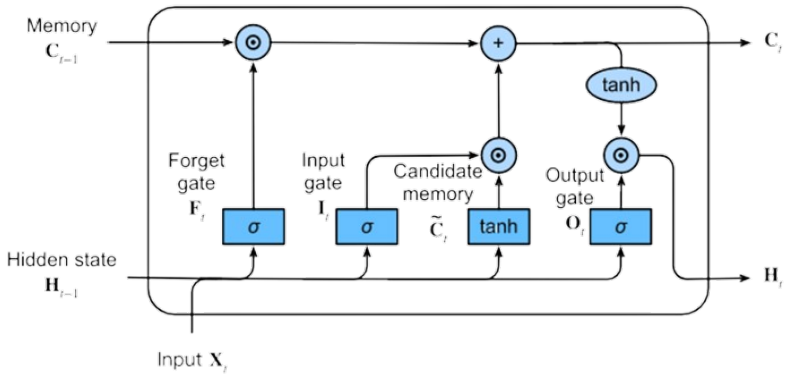


Figure 2.2: LSTM Architecture from the paper by Pascanu et al. [26].

The LSTM cell maintains a cell state, which serves as a long-term memory, and three gates: input gate, forget gate, and output gate. These gates control the flow of information into and out of the cell state, allowing LSTMs to selectively update and forget information over time.

At each time step t, the operations of an LSTM cell can be described as follows:

- Forget gate: $f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$
- Input gate: $i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$
- Candidate cell state: $\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$
- Updated cell state: $C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$
- Output gate: $o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$
- Hidden state: $h_t = o_t * \tanh(C_t)$

Where:

- x_t is the input at time step t,
- h_{t-1} is the previous hidden state,
- f_t, i_t, o_t are the forget gate, input gate, and output gate activations,
- $\tilde{C}_t$ is the candidate cell state,
- C_t is the updated cell state,
- W_f, W_i, W_C, W_o are weight matrices, and b_f, b_i, b_C, b_o are bias vectors,
- σ is the sigmoid activation function,
- $*$ denotes element-wise multiplication.

This architecture has enabled LSTMs to excel in tasks related to natural language processing (NLP) and specifically log anomaly detection [4]. Such tasks require LSTMs to capture the dependencies within sequential data, particularly textual data, in order to process and understand it thoroughly. Their capability of retaining

information over extended periods, inherent in such deep neural networks (DNN), allows them to effectively model the complex structures and nuances of language.

As outlined in the works of Mengying Wang, Lele Xu, and Lili Guo [13], table 2.2 showcases the efficacy of Naïve Bayes and LSTMs when performing anomaly detection on the Thunderbird log dataset. Notably, employing either TF-IDF or Word2vec feature extraction with both Naïve Bayes and LSTM demonstrated exceptional performance, with the deep neural network LSTM surpassing the previously state-of-the-art Naïve Bayes.

Table 2.2: Evaluation metrics of different methods on thunderbird dataset [13].

Feature extraction	**Learning algorithms**	**Accuracy**	**Precision**	**Recall rate**	**F1-score**
TF-IDF	Naïve Bayes	0.93	0.89	0.91	0.90
	LSTM	0.99	0.99	0.99	0.99
Word2vec	Naïve Bayes	0.98	0.98	0.97	0.97
	LSTM	0.99	0.99	0.99	0.99

While the previous models displayed adequate performance in public log datasets for some real-time systems [13], they increasingly started to exhibit limitations. These models heavily rely on labeled data, which is often scarce in real-world scenarios. Additionally, RNNs suffer from prolonged training times and deficiency in capturing long-range dependencies efficiently due to sequential processing [26]. A promising alternative to overcome these challenges, while potentially surpassing the performance of traditional models, is the transformer architecture.

2.5 Transformer-based Architectures

Transformer-based architectures were introduced in the context of sequence-to-sequence learning by Vaswani et al. [27]. The key innovation of the Transformer was the self-attention mechanism, which replaced sequential processing, allowing for parallel processing of input sequences, thus eliminating the need for recurrence. This mechanism computes attention weights across all positions of an input sequence to capture relationships between words. It allows each word in the sequence to attend to all other words simultaneously, enabling better capture of long-range dependencies and facilitating better understanding of the context.

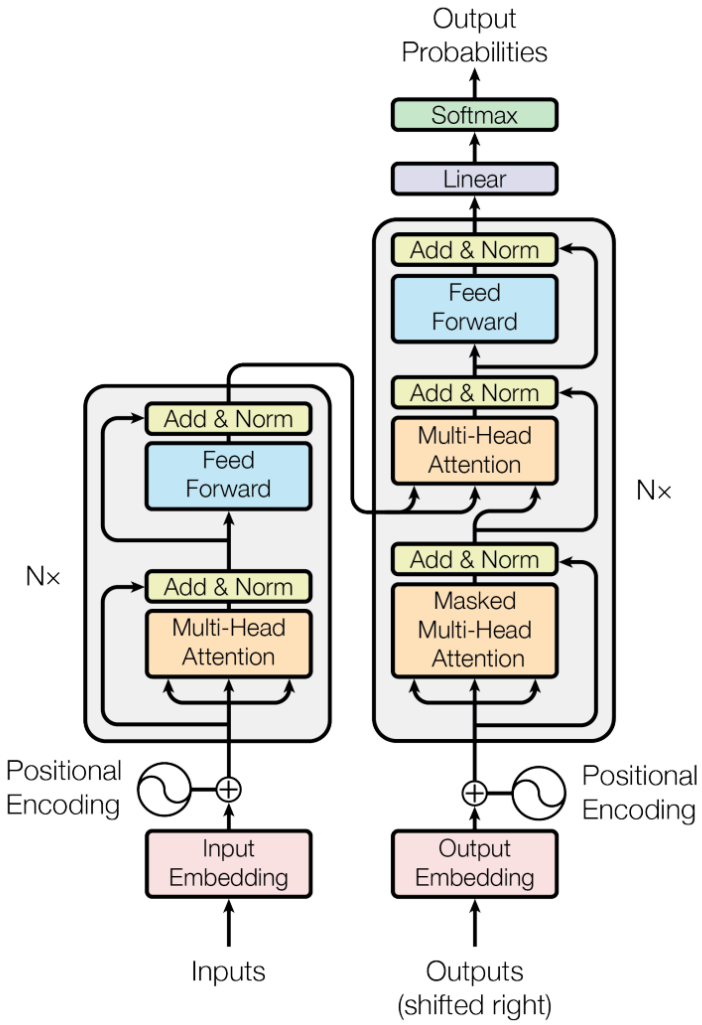


Figure 2.3: The Transformer (left: Encoder, right: Decoder), from model architecture from the paper by Vaswani et al. [27].

The Transformer architecture consists of several key components:

- **Encoder**: The encoder, represented on the left side of figure 2.3, processes the input sequence and generates a sequence of hidden states. Let $x = (x_1, x_2, ..., x_n)$ be the input sequence where x_i is the embedding vector for the i-th token. The encoder is composed of L identical layers. Each layer l has two sub-layers:

    1. **Multi-Head Self-Attention Mechanism**: This mechanism computes attention weights that capture dependencies between different positions of the input sequence. Given queries Q, keys K, and values V, the attention weights A are computed as:

$$A = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)$$

where d_k is the dimensionality of the key vectors. The output of the attention mechanism is obtained by weighting the values V with the attention weights A: $Z = AV$.

2. **Position-wise Feed-Forward Network (FFN)**: After the attention mechanism, a position-wise fully connected feed-forward network is applied independently to each position of the sequence. It consists of two linear transformations with a ReLU activation function in between:

$$\text{FFN}(Z) = \text{ReLU}(ZW_1 + b_1)W_2 + b_2$$

where W_1, W_2 are weight matrices, and b_1, b_2 are bias vectors.

- **Decoder**: The decoder,represented on the right side of figure 2.3, generates the output sequence based on the encoded input sequence. Similar to the encoder, the decoder is composed of L identical layers, each containing three sub-layers:

  1. **Masked Multi-Head Self-Attention Mechanism**: In the decoder, a masked multi-head self-attention mechanism is used to prevent positions from attending to subsequent positions. This ensures that during generation, each token only attends to previous tokens. The masking is achieved by setting the attention weights to $-\infty$ for positions that should not be attended to and applying a softmax function afterward.

  2. **Multi-Head Attention Mechanism**: This sub-layer attends to the encoder's output to incorporate information from the input sequence. It takes the decoder's output as queries and the encoder's output as keys and values.

  3. **Position-wise Feed-Forward Network (FFN)**: Similar to the encoder, a position-wise FFN is applied to each position of the decoder's output sequence.

- **Multi-Head Attention Mechanism**: This mechanism allows the model to focus on different parts of the input sequence independently. Given queries Q, keys K, and values V, the output of the multi-head attention mechanism with h heads is computed as:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_h)W^O$$

  where each head is calculated as:

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

  and W_i^Q, W_i^K, W_i^V, W^O are learnable weight matrices.

- **Positional Encoding**: Since the transformer architecture does not inherently capture the sequential order of the input sequence, positional encodings are added to provide information about the position of each token in the sequence. The positional encoding for position *pos* and dimension i is computed as:

$$PE_{(pos,i)} = \begin{cases} \sin\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right) & \text{if } i \text{ is even} \\ \cos\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right) & \text{if } i \text{ is odd} \end{cases}$$

where d_{model} is the dimensionality of the model.

By leveraging self-attention, Transformers can process input sequences in parallel, making them highly scalable and efficient, especially when dealing with long sequences. This parallelism leads to faster training and inference times compared to sequential models like RNNs. One notable example of the efficiency and effectiveness of the Transformer architecture is the BERT (Bidirectional Encoder Representations from Transformers) model, introduced by Jacob Devlin et al. [28].

2.5.1 BERT

The BERT (Bidirectional Encoder Representations from Transformers) model [28] consists of multiple layers of Transformer encoders. BERT differs from traditional Transformer models in how it processes input sequences. Traditional models process input sequences in a sequential unidirectional manner, meaning that at each position in the sequence, the model attends only to the tokens that precede it. In contrast, BERT utilizes a bidirectional approach. During both pre-training and fine-tuning, it considers information from both the left and right contexts of each token simultaneously. This results in each token's representation encoding information from both directions. In order to learn contextualized representations of words, BERT is trained on large corpora of text data and employs two strategies during pre-training, namely, Masked Language Model (MLM) and Next Sentence Prediction (NSP) [28].

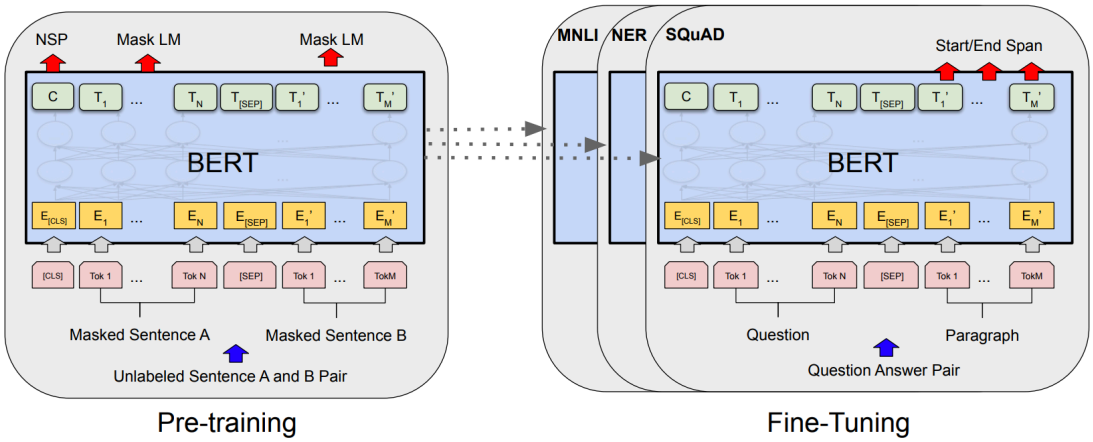


Figure 2.4: Overall pre-training and fine-tuning procedures for BERT from the paper by Devlin et al. [29].

In the MLM task, BERT randomly masks some of the tokens in the input sequence and trains the model to predict the masked tokens based on the surrounding context. This task encourages the model to understand bidirectional dependencies between

words and capture contextual information effectively. For example, considering the sentence: "The cat sat on the mat.", BERT might replace the word "mat" with the [MASK] token and try to predict it based on the context of the surrounding words [28].

Similarly, in the NSP task, BERT predicts whether a given pair of sentences occur consecutively in the original text or not. This task helps BERT to understand the relationships between sentences and capture discourse-level information. For instance, given two sentences: "The cat sat on the mat." and "It was a sunny day.", the NSP task aims to predict whether the second sentence follows the first one or not. By utilizing these pre-training objectives, BERT is able to capture rich contextual representations of words, enabling it to understand the semantics and relationships between words more effectively [28].

Following pre-training on large text corpora, transfer learning approaches, such as fine-tuning, can be used to adapt BERT on specific downstream tasks with relatively small amounts of task-specific data. This enables BERT to be effectively tailored to address domain specific tasks, which is one of the key advantages contributing to its success [30].

2.6 Transfer Learning

The concept of Transfer Learning lies in employing a model that has been pre-trained on a large dataset for a particular task and adapting its learned knowledge to a related, yet distinct, domain [31]. Through this method, we are able to re-purpose the learned features of the pre-trained model as a foundation for the new endeavor, potentially saving significant computational resources and time. Typically, the pre-trained model undergoes modifications, such as the addition of task-specific output layers, to adapt it for the new downstream task [32]. Transfer learning has shown significant advantages [31], especially in scenarios where the dataset for the new task is limited, or training a model from scratch is unfeasible due to resource constraints.

2.6.1 Fine-tuning

Fine-tuning represents a specialized form of transfer learning and is one of the most common strategies among applications in neural networks [33]. It involves taking a pre-trained model, trained on a large dataset, and then further training it on a new, smaller dataset that may have a different distribution or task than the original one. During fine-tuning, not only the weights of the added layers (typically the output layer) are trained, but also some or all of the weights of the pre-trained layers are updated to adapt to the new data. While fine-tuning offers significant advantages over training from scratch, it can still be computationally expensive to fine-tune every layer and billions of parameters. Therefore, techniques like layer freezing can be employed to simplify the process and optimize computational resources.

2.6.2 Layer Freezing

Layer freezing is a technique employed during fine-tuning to selectively restrict the updating of certain layers while allowing others to continue learning [34]. By "freezing" specific layers, typically those closer to the input and containing more generic features, the computational burden is reduced significantly. These frozen layers retain their learned representations from the pre-training phase, acting as fixed feature extractors, while only the parameters in the unfrozen layers are updated to adapt to the new task or dataset [33]. This selective updating not only streamlines the fine-tuning process but also helps prevent overfitting, as it focuses the model's capacity for learning on the most task-relevant layers.

These traditional transfer learning approaches have been successful in adapting pre-trained language models (PLMs) to specific downstream tasks. Recently, as PLMs continue to increase in size, the feasibility of full fine-tuning, which retrains all model parameters, becomes increasingly limited due to computational constraints [35]. This prompted the development of parameter-efficient fine-tuning (PEFT) methods [36] to adapt PLMs in resource-constrained environments.

2.7 Knowledge Distillation

Knowledge distillation [37] [38] is a compression technique that trains a compact model, referred to as the student, to replicate the behavior of a larger model or an ensemble of models, known as the teacher. This process allows the distilled, smaller model to perform similarly to the larger model while being more efficient in terms of computation and memory usage [39].

In supervised learning, classification models are trained to predict the class of an instance by maximizing the estimated probability of the correct labels. This typically involves minimizing the cross-entropy between the predicted distribution of the model and the empirical distribution of the training labels, represented in a one-hot manner. The probabilities assigned by the model to incorrect classes, although near zero, are not all equally small. Some are larger than others, indicating a partial insight into the model's generalization capability and its expected performance on unseen data [39].

The student model is trained using a distillation loss on the soft target probabilities provided by the teacher model. The distillation loss is defined as $L_{\text{ce}} = \sum_i t_i \cdot \log(s_i)$, where t_i and s_i represent the probabilities estimated by the teacher and the student, respectively. This approach leverages the full distribution of the teacher's output to provide a richer training signal to the student.

Following Hinton et al. (2015) [38], the softmax function is modified with a temperature parameter T, defined as $p_i = \frac{\exp(z_i/T)}{\sum_j \exp(z_j/T)}$, where z_i is the score for class i produced by the model. The temperature T controls the smoothness of the output distribution, with higher values producing softer probabilities. Both the student and the teacher use the same temperature T during training, while T is set to 1 during inference to revert to the standard softmax function. The final training objective

is a linear combination of the distillation loss L_{ce} with the supervised training loss [39].

2.7.1 DistilBERT

DistilBERT is a streamlined version of BERT, designed to retain the model's high performance while significantly reducing its size and computational footprint. The student model, DistilBERT, shares the general architecture of BERT, except that it eliminates the token-type embeddings, the pooler and reduces the number of layers by half. This reduction in layers is based on findings that, within the constraints of a fixed parameter budget, the computational efficiency is less affected by variations in the tensor's hidden size dimension than by the number of layers. Therefore, the emphasis is on reducing layer count, as operations in the Transformer architecture, such as the linear layer and layer normalization, are well-optimized in current linear algebra frameworks [39].

A key factor in the training procedure is the proper initialization of the student model to ensure convergence. Given the dimensional compatibility between the teacher and student networks, the student is initialized by selectively using every second layer from the teacher. This method leverages the structural insights from the teacher model, providing a solid foundation for the student's learning process [39].

The distillation follows recent best practices for BERT training, as suggested by Liu et al. (2019) [40]. DistilBERT's training utilizes large batches, facilitated by gradient accumulation, to handle up to 4,000 examples per batch. This process employs dynamic masking and excludes the next sentence prediction objective, aligning with modifications in BERT's training regime that focus on core language understanding capabilities.

With these modifications, DistilBERT is able to achieve comparable performance on downstream tasks while requiring fewer computational resources, by distilling the essential information from BERT into a more lightweight architecture [39].

2.8 Evaluation Metrics

All of the models and approaches detailed in the previous paragraphs require precise metrics to evaluate their performance. In order to evaluate classification models, the confusion matrix is a fundamental tool. It provides a tabular representation of the model's predictions versus the actual values. In a binary classification scenario with two classes, the confusion matrix typically is as follows [11]:

	Predicted Positive	Predicted Negative
Actual Positive	True Positive (TP)	False Negative (FN)
Actual Negative	False Positive (FP)	True Negative (TN)

With the information provided by the confusion matrix, we're then able to calculate

five common metrics, including accuracy, precision, recall, F1 score and ROC [11].

Accuracy measures the proportion of correctly classified instances among the total instances:

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

Precision measures the proportion of true positive predictions among all positive predictions made:

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

Recall, also known as sensitivity or true positive rate (TPR), measures the proportion of true positive predictions among all actual positive instances:

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

F1 score is the harmonic mean of precision and recall, providing a balance between them. It is calculated using the formula:

$$\text{F1 Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

The ROC (Receiver Operating Characteristic) curve is a graphical representation used to evaluate the performance of a binary classifier system as its discrimination threshold is varied. The ROC curve plots the recall against the False Positive Rate (FPR) at various threshold settings. The False Positive Rate (FPR) is given by:

$$\text{FPR} = \frac{\text{FP}}{\text{FP} + \text{TN}}$$

3

Related Work

Traditionally, engineers have relied on keyword searches or rule matching techniques to identify anomalous logs [1]. However, with the exponential growth in the volume, velocity, and variety of logs generated by modern systems [41], manual methods have become increasingly impractical. To address this challenge, numerous studies have explored automated solutions leveraging statistical, traditional machine learning and deep learning algorithms, such as [42] and [4]. While these methods have been showing convincing improvements, they are still limited by some challenges, inherent to both the problem space and the chosen approaches. Some of the main hurdles that need to be overcome are summarized in the following paragraph [2].

3.1 Challenges in Log Anomaly Detection

We first explore the primary hurdles in the field, including data representation, class imbalance, label availability, stream processing, and the evolution of logging statements. These challenges underscore the complexities involved in processing and analyzing log data effectively.

1. **Data Representation**: Representing data for deep learning poses a significant challenge, especially when dealing with log data. Logs often contain a mixture of diverse event types, unstructured messages, and parameters. This complexity makes pre-processing logs quite complicated. Traditional methods rely heavily on manual feature extraction, which is not scalable [43].

2. **Class Imbalance**: Anomalous events in log data occur far less frequently than normal ones. When a dataset contains a number of occurrences of one event/class which significantly outweighs the remaining one, it is called an imbalanced dataset. When dealing with anomaly detection, approaches based on neural network encounter sub-optimal performance, since this imbalance can lead neural networks to prioritize learning the more frequent class, remaining unable to detect the rarer anomalies [44].

3. **Label Availability**: In real-world applications, it is extremely rare to find labeled datasets, especially large enough to successfully train a supervised machine learning model. For this reason, many approaches fall into the semi-supervised or unsupervised categories, which rely on the assumption that anomalies are rare and different from normal data [41].

4. **Stream processing** : Logs are normally produced in a continuous stream, requiring anomaly detection models to have quick inference times and necessitating single-pass data processing [43]. Models need to balance accuracy with computational efficiency to be practical in real-time environments.

5. **Evolution of Logging Statements**: Since developers are constantly modifying the codebase, logging statements can change frequently, forcing anomaly detection techniques to be adaptable. This requires models that can generalize well from past data and quickly adapt to new patterns [5].

Following the discussion on challenges, we present various machine learning approaches that have been proposed to overcome these obstacles. We examine traditional and deep learning models, focusing on LSTM-based approaches like DeepLog [4] and LogRobust [5], as well as more advanced methods leveraging transformers and BERT architectures, such as LogSy [45], LogBERT [42], and LAnoBERT [3]. Each of these methods offers unique solutions to the highlighted challenges, demonstrating the evolution and innovation in the field of log anomaly detection.

3.2 LSTM-based Approaches

In the following sections, two LSTM-based approaches will be presented, namely DeepLOG [4] and LogRobust [5]. Both of these approaches tackle some of the important challenges listed in section 2.1.

3.2.1 DeepLOG

To overcome some of the challenges highlighted in the previous paragraphs, Du et al. [4] proposed a novel log anomaly detection approach called "DeepLog". This model implements a Long Short-Term Memory (LSTM) architecture, a type of recurrent neural network that is well-suited to capture the sequential nature of log entries. DeepLog learns normal execution patterns and flags deviations as anomalies, adapting to new patterns over time. To handle the unstructured nature of system logs, it employs a log parser that converts unstructured, free-text log entries into a structured representation. DeepLog achieved impressive results compared to its predecessors, reaching almost perfect F1 scores on the Hadoop dataset [46]. However, it still heavily relied on extensive parsing and structuring of the logs, which can be resource-intensive and may not generalize well across different log formats and systems.

3.2.2 LogRobust

LogRobust, like DeepLog, employs an LSTM architecture. However, the focus of the work by Zhang et al. [5] is on adapting to log files containing constantly changing content. They investigated log data from Microsoft's online service system and found that unstable log data is prevalent in real-world systems. LogRobust addresses this challenge by not relying solely on log event occurrences. Instead, it transforms each event into a semantic vector, capturing its semantic information. This approach

allows the model to identify and adapt to new but similar log events resulting from evolving logging statements or parsing errors. Additionally, LogRobust utilizes an attention-based Bidirectional Long Short-Term Memory (Bi-LSTM) classification model, which considers the sequence of semantic vectors as input. This model captures contextual information within log sequences and automatically learns the importance of different log events, positioning itself conceptually between an RNN and a transformer approach. Evaluations using the publicly available Hadoop dataset [46] with various ratios of changes injected (ranging from 5% to 20%) resulted in F1 scores from a maximum of 0.96 to a minimum of 0.89, demonstrating LogRobust's capability to maintain high precision and recall even in unstable scenarios.

3.3 Transformer and BERT-based Approaches

In the following sections, we are going to focus on transformer-based approaches by giving an overview of three distinct methods (LogSy [45], LogBERT [42] and LAnoBERT [3]), each tackling some of the important challenges listed in section 2.1.

3.3.1 LogSy

To further improve the generalization capabilities of anomaly detection models and address the need for better representations, Nedelkoski et al. [45] proposed a novel anomaly detection method that focuses on obtaining representative and compact numerical log embeddings. This method involves training a transformer to learn log vector representations by incorporating self-attention mechanisms and a new hyperspherical learning objective. This approach enables the model to achieve better separation between normal and anomalous data. Evaluations on three publicly available datasets (BGL, Thunderbird, and Spirit [46]) showed that LogSy achieved near-perfect F1 scores for the latter two datasets. Despite not achieving satisfactory scores on the BGL dataset, it still outperformed DeepLog's implementation [4], highlighting the effectiveness of transformer-based models in certain scenarios.

3.3.2 LogBERT

Increasing the complexity and size of the model, Guo et al. [42] proposed LogBERT, a BERT-based anomaly detection model. This approach involves training a model using a BERT architecture to recognize normal log data, similar to DeepLog. However, unlike RNN-based models that focus on predicting the next log message based on previous ones, LogBERT leverages BERT's capability to capture patterns within normal log sequences. The publication introduces two self-supervised tasks: masked log key prediction, which aims to predict masked log keys in normal sequences, and volume of hypersphere minimization, which encourages normal log sequences to cluster closely in the embedding space. This approach achieved extremely high F1 scores, reaching 0.9 on the BGL dataset and 0.96 on the Thunderbird dataset [42], [46], demonstrating the potential of BERT-based models for log anomaly detection.

3.3.3 LAnoBERT

While the studies described in the previous paragraphs have shown impressive performance, they still suffer from some limitations. One major constraint is their reliance on log parsers for preprocessing. Methods like DeepLog [4] that rely on parsers necessitate predefined templates and manual refinement by experts. Moreover, some approaches (e.g., [5], [45]) treat log anomaly detection as a binary classification problem, which is impractical due to the huge class imbalances found in log datasets. LAnoBERT aims to tackle these issues by training a BERT-based model solely on normal log data and introducing a novel detection technique. Leveraging the assumption that normal and abnormal logs exhibit distinct contextual patterns, the model classifies logs as anomalous or normal based on the mean confidence in predicting the next token given a specific log. If the model's confidence exceeds a certain threshold, the log is labeled as normal. LAnoBERT's performance is comparable to LogBERT's, achieving an F1 score of 0.87 for the BGL dataset and 0.99 for the Thunderbird dataset [3], [46], indicating its robustness and effectiveness in practical scenarios.

4

Data Description and Analysis

The data selected for this project was obtained from two different sources: the publicly available BGL dataset [47], [48] and proprietary log data from Ericsson. In the following sections these two datasets are analyzed and compared.

4.1 Data Collection and Description

In the sections below, we are going to give a brief description of the two datasets that were utilized throughout this project.

4.1.1 The BlueGene/L Dataset

The BlueGene/L (BGL) dataset was acquired from a publicly available source on GitHub and required no advanced collection steps [46]. The BGL dataset contains logs collected from the BlueGene/L supercomputer system at Lawrence Livermore National Labs (LLNL) in Livermore, California. This dataset comprises a collection of system logs generated by various components of the supercomputer during its operation. These logs encompass a wide range of activities, including job scheduling, resource allocation, and system performance monitoring. Leveraging the BGL dataset allows us to replicate and validate results from prior related research, thereby establishing a baseline for comparison and assessing the efficacy of our proposed solution.

4.1.2 The Ericsson Dataset

The process of collecting Ericsson's log data involved collaborating with Ericsson's internal teams responsible for managing and monitoring network operations. Through this collaboration, we gained access to a representative sample of log data generated by Ericsson's network infrastructure.

The Ericsson log data represents a proprietary dataset obtained from Ericsson's hardware components. This dataset contains log sequences from Ericsson's network infrastructure, presenting distinct characteristics compared to BGL. By using Ericsson's log data in our study, we aim to assess the generalization capabilities and robustness of our solution, thereby validating its effectiveness in real-world scenarios

Within the representative sample of log data, we were granted access to two distinct working directories. One directory contained data files exclusively comprising normal log occurrences, while the other directory contained a mixture of both normal and anomalous log occurrences. For the purpose of this thesis, they will be referred to as the 'train' directory and the 'test' directory, respectively. The train directory was composed of 55 log files, each containing an average of 22,000 normal lines per file. The test directory contained 19 files, with an average of 2,200 normal lines and 2,600 anomalous lines per file.

4.2 Data Processing

In the next two sections, the most crucial data cleaning processes that were applied to the Ericsson dataset as well as the data distributions for both the BGL and Ericsson dataset.

4.2.1 Data Cleaning

Our initial data processing step involved merging the contents of the 'train' and 'test' folders into 'train.log' and 'test.log' files, respectively. The objective was to ensure that all normal lines started with `YYYY-MM-DD HH:MM:SS`, denoting the timestamp, while anomalous lines began with `- YYYY-MM-DD HH:MM:SS`, with the preceding dash indicating an anomalous line.

The data cleaning process for the 'train.log' involved three main steps. Firstly, we focused on removing any XML content that was present throughout the file and eliminated empty lines. Subsequently, certain log lines did not begin with the month of the year, indicating the log had been split across multiple lines. Therefore, we merged these with the preceding lines to maintain continuity and accuracy in the log data. Finally, we leveraged regular expressions to identify and standardize various patterns within the log data. These patterns included date, time, IP addresses, numeric values, and file paths. Each line was processed individually, with recognized patterns replaced by designated placeholders such as "TIME," "IP," "NUM," or "PATH." This standardization of the data format facilitated easier pattern recognition and learning by subsequent models. Following these steps, the 'train.log' file contained 156,803 normal log entries.

The cleaning process continued with the 'test.log' file, where we also needed to remove XML content and empty lines. Here, we didn't apply regex operations since these will be done later on within the anomaly detection pipeline. With these operations, we obtained 190,590 normal log entries and 49,993 anomalous ones.

4.2.2 Data Distribution

The following table summarizes the distribution of data from the Ericsson and BGL datasets. There are 397,386 annotated log messages from Ericsson and 4,747,963 from BGL. Among the annotated Ericsson logs, there are 347,393 instances of normal

data and 49,993 instances of anomalous entries. Regarding the BGL dataset, there are 4,399,503 instances of normal data and 348,460 instances of anomalous data.

Table 4.1: Overview of log data with percentages.

	Ericsson	BGL	% (Ericsson)	% (BGL)
Total (annotated)	397,386	4,747,963	100	100
Normal Data	347,393	4,399,503	87.42	92.66
Anomalous Data	49,993	348,460	12.58	7.34

4.3 Data Analysis

To better understand the nature of the BGL and Ericsson datasets, we performed an analysis on the unigram level. This allowed us to compare the main differences between the two datasets.

4.3.1 Token Frequency in Normal Lines

We started by comparing the top 10 tokens in normal lines in Table 4.2. In the BGL dataset, the most frequent tokens are "ras" and "kernel", appearing in approximately 97.65% and 91.89% of the normal lines, respectively. These tokens reflect a system-level technical vocabulary. On the contrary, terms like "info" "core", "error", "generating" are also present and represent more commonly used words in natural language.

In the Ericsson dataset, a different set of dominant tokens are present, with "info" being the most frequent at 61.15%, followed by "asesdk" and "notice" at 38.79% and 34.82%, respectively. These tokens, together with "uid," "opad," and "rcm.", belong to a more technical context in comparison to the BGL dataset.

Table 4.2: Top 10 tokens in normal log lines.

BGL Tokens	Percentage	Ericsson Tokens	Percentage
ras	97.650986	info	61.150754
kernel	91.890792	asesdk	38.791127
info	84.942056	notice	34.822899
core	40.415001	uid	26.900972
generating	38.853576	in	22.932744
alignment	13.374573	opad	21.012399
exceptions	13.326954	id	20.080075
fatal	11.518910	rcm	19.345517
dear	10.539822	session	14.796100
error	9.903046	thread	14.330341

4.3.2 Token Frequency in Anomalous Lines

In Table 4.3, we observe a similar trend in the technical nature of anomalous logs between BGL and Ericsson. In the BGL dataset, the top technical tokens are "ras," and "kerndtlb," occurring with 99.95% and 42.83% frequency in anomalous lines, respectively. The remaining tokens, such as "fatal," "error," "interrupt," and "data," are common in natural language. In the Ericsson dataset, tokens like "asesdk," "uid," and "lp" dominate the anomalous lines. Furthermore, we see the presence of tokens like "libiforwardingmgmtservice" and "onCreateGatewayReq" which represent event names in the logs.

Therefore, it is evident that there is a disparity between the top tokens in each dataset. The BGL dataset contain tokens that are more frequent in natural language, whereas the Ericsson dataset comprises tokens that are more technical and less used in natural language contexts. This distinction influences how a machine learning model recognizes tokens within its vocabulary if it is not trained on specific log content.

Table 4.3: Top 10 tokens in anomalous log lines.

BGL Tokens	**Percentage**	**Ericsson Tokens**	**Percentage**
ras	99.995695	asesdk	82.047487
fatal	99.982207	uid	66.815354
kernel	81.600184	lp	51.245174
error	62.445905	failed	45.226332
interrupt	62.141996	to	39.659552
data	62.069965	service	32.840598
tlb	43.831430	gifmd	32.218511
kerndtlb	43.831143	libiforwardingmgmtservice	32.116496
on	18.983528	onCreateGatewayReq	29.482127
message	18.580612	err	29.460124

4.3.3 Token Frequency in both Normal and Anomalous Lines

In Table 4.4, we observe contrasting patterns between the BGL dataset and the Ericsson dataset regarding the presence of common tokens found in both normal and anomalous logs.

In the BGL dataset, tokens like "ras" and "kernel" show similar presence in both normal and anomalous logs, indicating they aren't more prevalent in either type of log. Conversely, tokens such as "fatal," "info," and "error" show clear distinctions. "Fatal" appears in 99% of all anomalous logs, but only in 11% of normal logs. "Info" is present in 84% of normal logs, but rarely in anomalous ones. Similarly, "error" occurs in 62% of anomalous logs but only in 9% of normal logs. This suggests that based on the presence of these tokens, it is relatively easy to predict whether a log is normal or anomalous.

In contrast, the Ericsson dataset demonstrates a different behavior. The token "asesdk" has an 82% presence in anomalous lines and 38% in normal lines. This indicates that the presence of "asesdk" alone is not as indicative of whether a log is anomalous or normal. This pattern extends to other common tokens like "uid," "info," "lp," "notice," and "failed," where their presence does not strongly correlate with the classification of logs as normal or anomalous.

Table 4.4: Top Common Tokens between Normal and Anomalous Lines.

BGL Tokens	**% N**	**% A**
ras	97.650986	99.995695
kernel	91.890792	81.600184
fatal	11.518910	99.982207
info	84.942056	0.000574
error	9.903046	62.445905
Ericsson Tokens	**% N**	**% A**
asesdk	38.791127	82.047487
uid	26.900972	66.815354
info	61.150754	16.586322
lp	11.781990	51.245174
notice	34.822899	27.675875
failed	4.552646	45.226332

After observing the results from the tokens common between normal and anomalous lines, we decided to further explore the tokens exclusive to either normal or anomalous lines.

4.3.4 Frequency of Tokens Exclusive to Normal Lines

In Table 4.5, we examine the top tokens found exclusively in normal lines of the BGL dataset. The token "core" appears only in normal lines, occurring with a frequency of 40%. Therefore, whenever "core" is present in a log line, it is guaranteed be normal. This observation also applies to the tokens "generating", "alignment", "exceptions", and "dear", though their frequencies decrease from 38% to 10% in normal logs. On the other hand, the top tokens found exclusively in normal lines of the Ericsson dataset, such as 'cmQueryRequestCtxtId' and 'contextName,' have a much lower frequency of 2.81%. This indicates that their presence in log lines is relatively rare compared to normal tokens in the BGL dataset, which range from 10% to 40% frequency. Similarly, the tokens 'rdbRead', 'username', and 'rdbread' also appear less frequently, around 2.15%. These findings underscore that while exclusive normal tokens in the BGL dataset are more prevalent, in the Ericsson dataset, they are far less common, ranging from 2.8% to 2.1%. This demonstrates that the Ericsson dataset is more complex and cannot be as easily with rule-based methods.

Table 4.5: Top Tokens that are Present in Normal but not in Anomalous Lines.

BGL Tokens	Percentage	Ericsson Tokens	Percentage
core	40.415001	cmQueryRequestCtxtId	2.809079
generating	38.853576	contextName	2.809079
alignment	13.374573	rdbRead	2.157664
exceptions	13.326954	username	2.157664
dear	10.539822	rdbread	2.157664

4.3.5 Frequency of Tokens Exclusive to Anomalous Lines

In Table 4.6, we include the most common tokens found exclusively in anomalous log lines. In the BGL dataset, the token "kerndtlb" stands out, appearing in 43.83% of anomalous lines, indicating that when present the log line can be classified immediately as anomalous. Other top tokens are "kernstor" and "socket", with occurrence rates of 18.22% and 17.28% respectively. The tokens "ciostream" and "prefix" also show a similar frequency at 17.22%, indicating the a anomalous log. In the Ericsson dataset, the token "onCreateGatewayReq" appears in 29.48% of anomalous logs, making it the most frequent token associated with anomalies in this dataset. This is followed by tokens like "getCpuAllocationData" and "RouteMgmtControllerIpos", each occurring in approximately 11.91% to 14.65% of anomalous lines.

This analysis highlights a contrast in the frequency of tokens exclusive to anomalous lines between the two datasets. While the BGL dataset tokens show a higher frequency range (from 17.22% to 43.83%), the Ericsson dataset tokens are generally less frequent, with a maximum of 29.48%. Such differences imply that the Ericsson dataset exhibits a bigger variety of anomalies, making it more challenging to detect anomalies based solely on token frequency compared to the BGL dataset.

Table 4.6: Top tokens that are present in anomalous but not in normal lines.

BGL Tokens	Percentage	Ericsson Tokens	Percentage
kerndtlb	43.831143	onCreateGatewayReq	29.482127
kernstor	18.220456	getCpuAllocationData	14.654052
socket	17.277162	RouteMgmtControllerIpos	11.913668
ciostream	17.217184	creatingGateway	11.913668
prefix	17.217184	libroutemgmtctrl	11.913668

4.3.6 Frequency-based Stochastic Classification

Leveraging these frequencies, we can determine the probability of a log being anomalous when one of the top common tokens between normal and anomalous logs is present. Therefore, the probability of a log being anomalous given that token i is present is denoted as:

$$P(\text{anomalous}|\text{token}_i)$$

Using the complement of the previous probability, we can also calculate the probability of a log being normal given that token i is present as:

$$P(\text{normal}|\text{token}_i) = 1 - P(\text{anomalous}|\text{token}_i)$$

Table 4.7: Probability of BGL Log Anomaly Given Token Presence

Token	**% of Times Present**	$P(\textbf{anomalous}\|\textbf{token})$	$P(\textbf{normal}\|\textbf{token})$
ras	98.3161	0.288525	0.711475
kernel	88.9716	0.260176	0.739824
fatal	36.6141	0.774643	0.225357
info	60.8460	0.000003	0.999997
error	24.8083	0.714057	0.285943

In Table 4.7, related the BGL dataset, the top common token "ras" appears in 98.3161% of the logs, with a 28.8525% probability that a log containing this token is anomalous. Conversely, there is a 71.1475% probability that a log containing "ras" is normal. This suggests that while "ras" is very common, its presence is more strongly correlated with normal logs than with anomalies. In contrast, the token "fatal" appears in only 36.6141% of the logs, but it has a much higher probability (77.4643%) of indicating an anomalous log. The token "info," which appears in 60.8460% of the logs, almost exclusively indicates a normal log, with a probability of being normal at virtually 100%. Tokens like "kernel" and "error" also show interesting patterns. "Kernel" is present in a high percentage of logs (88.9716%) but has only a 26.0176% probability of indicating an anomaly, suggesting it is typically associated with normal logs. "Error" appears less frequently (24.8083% of logs) but has a significant 71.4057% probability of being associated with anomalies.

Table 4.8: Probability of Ericsson Log Anomaly Given Token Presence

Token	**% of Times Present**	$P(\textbf{anomalous}\|\textbf{token})$	$P(\textbf{normal}\|\textbf{token})$
asesdk	51.2282	0.460494	0.539506
uid	38.3771	0.500577	0.499423
info	48.3376	0.098658	0.901342
lp	23.1284	0.637051	0.362949
notice	32.7680	0.242839	0.757161
failed	16.2471	0.800354	0.199646

In the Ericsson dataset, the token "asesdk" appears in 51.2282% of the logs, showing a relatively balanced probability of being associated with either anomalous (46.0494%) or normal logs (53.9506%). The token "uid" is noted in 38.3771% of logs with a near-equal probability of these logs being anomalous (50.0577%) or normal (49.9423%). This implies that "uid" is a common token but does not specifically correlate with

anomalies. On the other hand, "info" is present in 48.3376% of logs and primarily indicates a normal log with a high probability of 90.1342%. The tokens "lp" and "failed" show more pronounced indications of anomalies, while not being frequent. "lp" appears in 23.1284% of logs with a 63.7051% probability of being anomalous. Similarly, "failed" is less frequent (16.2471% presence) but has a high likelihood (80.0354%) of indicating an anomaly. Finally, "notice", found in 32.7680% of logs, is more often associated with normal logs (75.7161%) rather than anomalies (24.2839%).

Overall, when comparing the results between the BGL and Ericsson datasets, we observe that the tokens in the BGL dataset typically indicate whether a log is anomalous or normal. In contrast, in the Ericsson dataset, tokens that occur with high frequency do not strongly indicate specific anomalies by themselves, as their conditional probabilities are more balanced. Consequently, while certain tokens in the BGL dataset have a significant impact on log classification, the Ericsson dataset appears to be more complex, featuring more technical tokens compared to the BGL ones, which often resemble natural language. Therefore, common tokens in the Ericsson dataset exhibit less predictive power regarding log classification, suggesting the need for more sophisticated methods compared to the BGL dataset.

5

Methods

In this chapter, the approaches that were utilized throughout the project will be described. They include the RAPID method, introduced in 2023 by Lee et al. [6], and an alternative approach which entails fine-tuning a model and employing it to extract representations. In the following paragraphs, we will go through the RAPID framework, the adaptions that had to be implemented to make it compatible with the Ericsson log data, and the fine-tuning strategies that were explored.

5.1 The Rapid Framework

The RAPID framework [6] takes advantage of the inherent features of log data to perform anomaly detection without training a model. It treats logs as natural language, extracts representations using a pre-trained language model and then implements a retrieval technique to contrast test logs with the most similar normal logs [6]. The RAPID pipeline can be divided into two main steps: database construction and RAPID processing.

5.1.1 Database construction

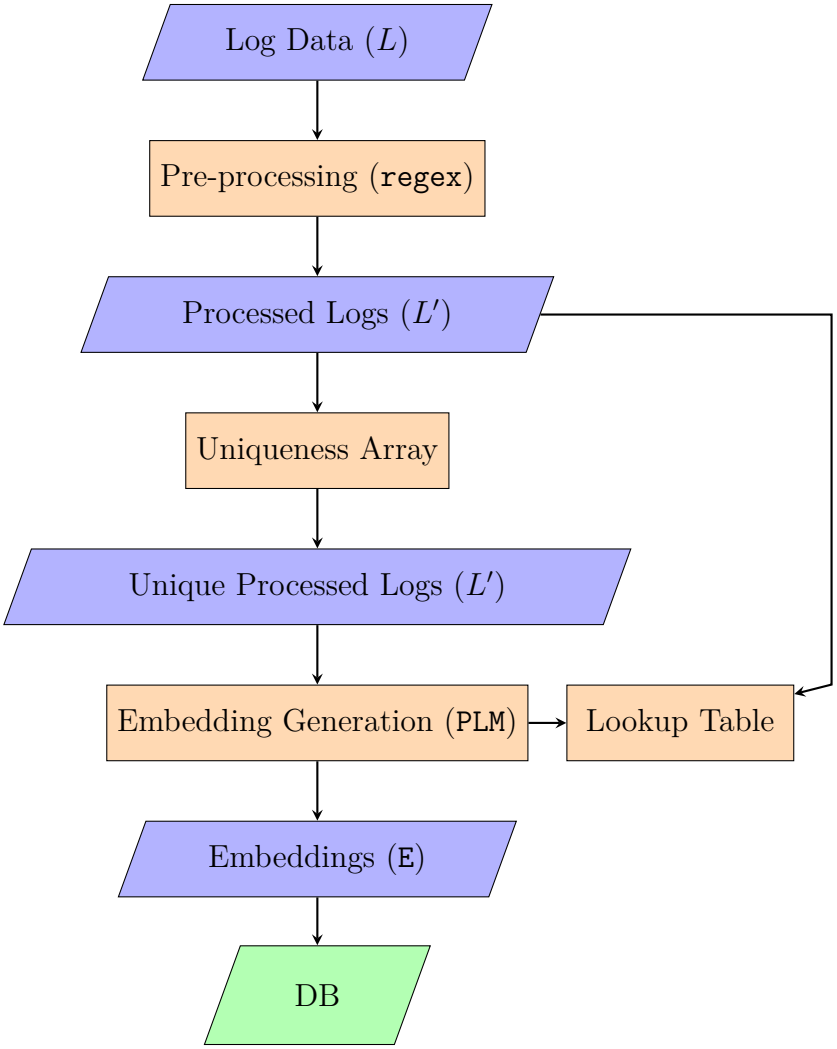
Database construction involves creating a database (DB) from a set of logs (L) by applying a regular expression pre-processing (regex) and then storing unique processed log sequences alongside their embeddings. To achieve this, we iterate through each log in the train and test sets. Each log is pre-processed to obtain a processed sequence (L'), its respective timestamp, and, in the case of test logs, the label associated with the sentence. After pre-processing is completed, we create an array of unique sentences L' to avoid duplicates. These unique sentences L' are then used to create embeddings generated through a pre-trained language model (e.g., BERT). Once the embeddings are ready, we create a unique lookup table for each set to manage redundancy. The list of sentences L contains all sentences, including duplicates, while the unique sentence array holds only unique entries. The unique lookup table maps each sentence in the sentence list to its corresponding index in unique sentences and respective embeddings, facilitating quick reference and reducing storage needs. A graphic showcasing the most crucial steps of the database construction pipeline is included in Figure 5.1.

- **Pre-processing:** Applies a regular expression function (`regex`) to each log

in the dataset to obtain a processed log sequence (L').

- **Uniqueness Array:** Creates an array with the unique processed log sequences (L')
- **Embedding Generation:** Utilizes a pre-trained language model (`PLM`) to generate embeddings for the unique processed log sequences (L'), indicating their numerical representation for further processing or analysis.
- **Lookup Table:** Creates a unique lookup table which maps each sentence in the sentence list (L) to its corresponding index in unique sentences (L')and respective (`E`).
- **Result:** Outputs a `DB` containing unique log sequences, their embeddings, and a `Lookup` table.

Figure 5.1: Database construction pipeline inspired by the paper by Lee et al.[6] and refined for this project.



5.1.2 RAPID processing

The second step in the RAPID pipeline describes the process of the RAPID anomaly detection, where test logs (`Lts`) are attributed unique coreSets of known normal logs (`Lkn`). Using the coreSets, we calculate an "Anomalous score" for each test log based on similarity measures between them and the known normal logs. Each test log is classified as either "Normal" or "Anomalous" based on a predefined anomaly detection threshold (δ). To achieve this, we iterate over each test log representation (Lts) and create a coreSet of train representations using k-nearest neighbors (KNN). Once the coreSet for each test log is defined, two methods are used to calculate the abnormal score. Both methods start by calculating the cosine similarity between each test representation and every train representation of their respective coreSet. The first method uses the maximum cosine similarity between the test log and the train coreSet to obtain a single score for that data point. The second method uses the mean of the cosine similarity scores in the coreSet to obtain the score. With these similarity scores for each test log, we calculate the optimal threshold based on the ROC (Receiver Operating Characteristic) curve, computing the false positive rate (FPR), true positive rate (TPR), and threshold. The best threshold (δ) is then returned and used to classify every test log as either normal or anomalous. This process is represented in figure 5.2.

5.1.3 CoreSet creation

The first step of the anomaly detection process is to create a coreSet for each test point. A coreSet is a group of points from the known normal logs that are most similar to the test point in question based on their embeddings. To determine the size of the coreSet, which is the same for every single test point, we use a predefined parameter. This parameter is a percentage, ranging from 0 to 1. For example, if the coreSet parameter is 0.005, this means that the size of each coreSet will be $\mathbf{0.005 \times len(train\_representations) = 45}$. To calculate the exact coreSet for each test point, we use the **k-nearest-neighbors (KNN)** method where $\boldsymbol{k = 45}$. This way, each test log has its own coreSet which contains the 45 normal logs that are closest to their respective embedding.

5.1.4 Similarity Measures

Once the coreSets have been created, the next step is to calculate the similarity scores between each test data point and their respective coreSets. This is done using two methods. Both start with the calculation of the cosine similarity between the test point in question and each of the elements of the respective coreSet. Once all the similarity scores for the test data point are calculated, the first method selects the score with the maximum similarity. The second method selects the score by computing the mean of all the similarities within the coreSet. The result is an array of the size of the test points, where each entry has an associated anomalous score.

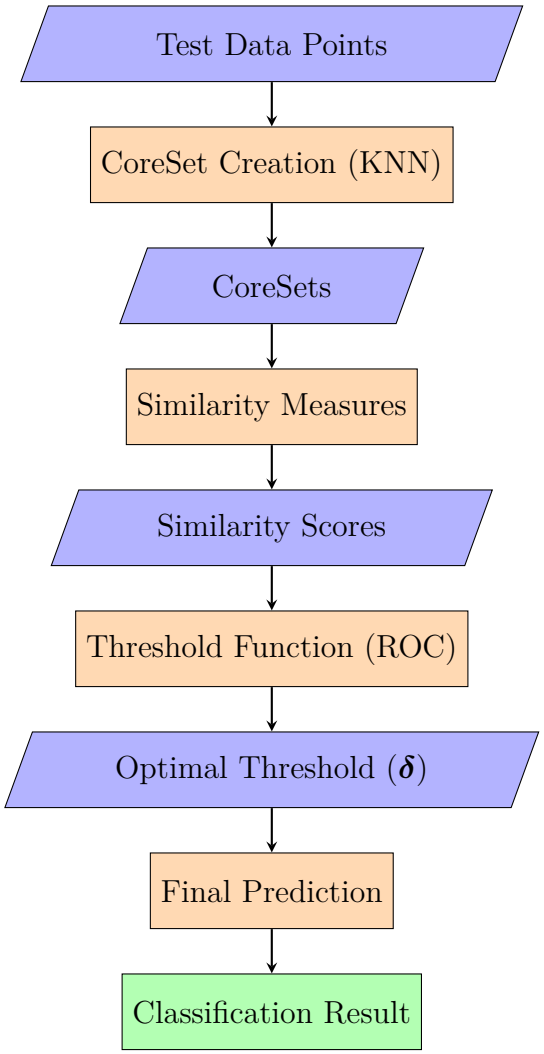
5.1.5 Threshold Function

Given the similarity scores obtained for each test log, we proceed to determine the optimal threshold using the Receiver Operating Characteristic (ROC) curve. This involves calculating the false positive rate (FPR), true positive rate (TPR), and corresponding thresholds. We then generate the ROC curve by plotting TPR against FPR at various threshold values. The point on the curve that maximizes the Youden's J statistic, defined as TPR minus FPR, signifies the optimal threshold ($\boldsymbol{\delta}$) used for classification.

5.1.6 Final Prediction

The final prediction is done using the optimal threshold ($\boldsymbol{\delta}$) and the anomaly scores previously calculated. For each test point, if their anomaly score is greater or equal then $\boldsymbol{\delta}$, we classify it as an anomaly. Otherwise, the log is classified as normal.

Figure 5.2: Anomaly detection pipeline inspired by the paper by Lee et al.[6] and refined for this project.



5.2 Adaptation to the Ericsson Dataset

In this section, we go through the pre-processing steps that were taken in order to adapt the Ericsson dataset to the RAPID method [6]. In addition, we also describe the fine-tuning method which yielded the most significant results as well as a brief mention of the methods which proved to be unsuccessful.

5.2.1 Pre-processing

In order to make this method as general as possible, pre-processing steps are being kept to a minimum. However, there are still some necessary steps that need to be taken, with the aim of avoiding any leakage of sensitive data as well as greatly reducing the computational load of the method. The following are the most important pre-processing steps applied to every log line in the data-set. These mostly include regex substitution operations and normalization of strings.

- **Date and Time Extraction and Substitution:** Utilizes two regular expressions, `date_time_regex` and `date_regex`, to identify date and time patterns in log strings, substituting these with a placeholder `TIME` for anonymity.
- **IP Address Anonymization:** Employs `ip_regex` to detect and replace IP address patterns with `IP`, ensuring network information is anonymized.
- **Numerical Data Generalization:** Uses `num_regex` to identify all numerical values, replacing them with `NUM` to abstract away specific numbers that could be sensitive and reducing the total number of unique token and strings.
- **Path Information Anonymization:** Applies `path_regex` to find and substitute path-like strings with `PATH`, anonymizing file or URL paths.
- **String Normalization:** The function `normalizeString` transforms the input string to lowercase, removes non-ASCII characters and any character not a letter or `<>`, and consolidates whitespaces, simplifying the data format for processing.

These pre-processing steps significantly contribute to the dual goals of enhancing data privacy and optimizing the dataset for the anomaly detection task at hand. By anonymizing sensitive information and reducing the variety of tokens through regex substitution, the data is made safer and the subsequent training and inference processes becomes more efficient. The normalization of strings further standardizes the data, making it easier to process and analyze. An example which uses a fictitious log line to show how the pre-processing steps affect the original log lines are included below.

```
"Error reported on Apr 7 15:20:30 by server at
   ↪ 192.168.1.100: User accessed /var/log/messages at
   ↪ 10234 bytes"
```

Listing 5.1: Original Log Line

```
"Error reported on  TIME  by server at  IP : User
  ↪ accessed PATH  at  NUM  bytes"
```

Listing 5.2: Modified Log Line after ericsson_regex

```
"error reported on time by server at ip user accessed
  ↪ path at num bytes"
```

Listing 5.3: Normalized Log Line after normalizeString

5.2.2 MLM Fine-tuning

To better adapt the model to the Ericsson dataset, we decided to explore the feasibility of finetuning by adapting distilBERT utilizing Masked Language Modeling (MLM), given that BERT employs MLM during pre-training [28]. In MLM, a percentage of tokens in the input text were randomly masked (replaced with the [MASK] token). The model's objective is to predict these masked tokens based on the surrounding context, encouraging the model to learn representations that capture the semantics and syntactic structures necessary for the specific task. Within each epoch, the model is trained on batches of data, with the masked input sequences and attention masks as inputs, and the original unmasked input sequences as labels. This setup allows the model to predict the masked tokens and compute the loss based on the predictions compared to the actual labels using the MLM objective.

5.2.3 Other Fine-tuning Approaches

We also attempted to fine-tune the model using a Sequence Classification approach and a Spherical Loss approach. The results were subpar, and therefore, we will focus on Masked Language Modeling (MLM) throughout the thesis.

5.3 Baseline

In order to establish a comparative foundation for our thesis, we decided to create a baseline approach for detecting anomalous logs using a classic Naive Bayes classification model. The embeddings in this approach are created by transforming the log data into a numerical format. This is achieved through a Bag-of-Words (BoW) model, which converts the logs into a sparse matrix representation where each entry represents the frequency of a word within a log. Once the data is vectorized, our Multinomial Naive Bayes model is trained on the training set, which contains both normal and anomalous logs. This classifier assumes word independence between tokens and is able handle varying word frequencies effectively. The training process involves learning the probabilistic relationships between the presence of words and the log categories (anomalous or normal).

5.4 Experimental setup

In this section, we will discuss the experimental setup, encompassing details on fine-tuning and the detection pipeline (RAPID parameters).

5.4.1 Fine-tuning

The approach described in this section pertains to the Masked Language Modeling method. Fine-tuning of the DistilBERT model was executed on an NVIDIA A2 with 15GB of memory, utilizing CUDA version 12.2. Throughout training, all layers were frozen except for the last transformer layer, leveraging pre-trained weights. To determine hyperparameters, several experiments were conducted to select the optimal ones. We settled on a learning rate of 2e-6, the Adam optimizer, and Cross Entropy for the loss function. Additionally, we employed a learning rate scheduler with a step size of 1 and a gamma of 0.95 to ensure gradual adjustment of the learning rate throughout training, promoting stable convergence and preventing abrupt fluctuations in optimization. The training process spans 22 epochs, during which forward and backward passes are performed for each batch, model parameters are updated, and metrics for training loss and accuracy are calculated. Following training, we compute validation loss and accuracy for each batch. The pseudocode for this process is described bellow in Algorithm 1.

Algorithm 1 Fine-tuning DistilBERT for Masked Language Modeling

Input: Training and validation data, pre-trained model, tokenizer
Output: Fine-tuned model, training, and validation logs
Load data tensors and create dataloaders
Initialize DistilBERT model and tokenizer
Freeze all layers except the last transformer layer
Set training parameters:
    Learning rate: 2e-6
    Optimizer: Adam
    Loss function: Cross Entropy
    Scheduler: step_size=1, gamma=0.95
for each epoch (0 to 22) **do**
    Training phase:
    for each batch **do**
        Forward pass and compute loss
        Backward pass and update parameters
    end for
    Validation phase:
    for each batch **do**
        Forward pass and compute loss
    end for
end for
Return: Fine-tuned model

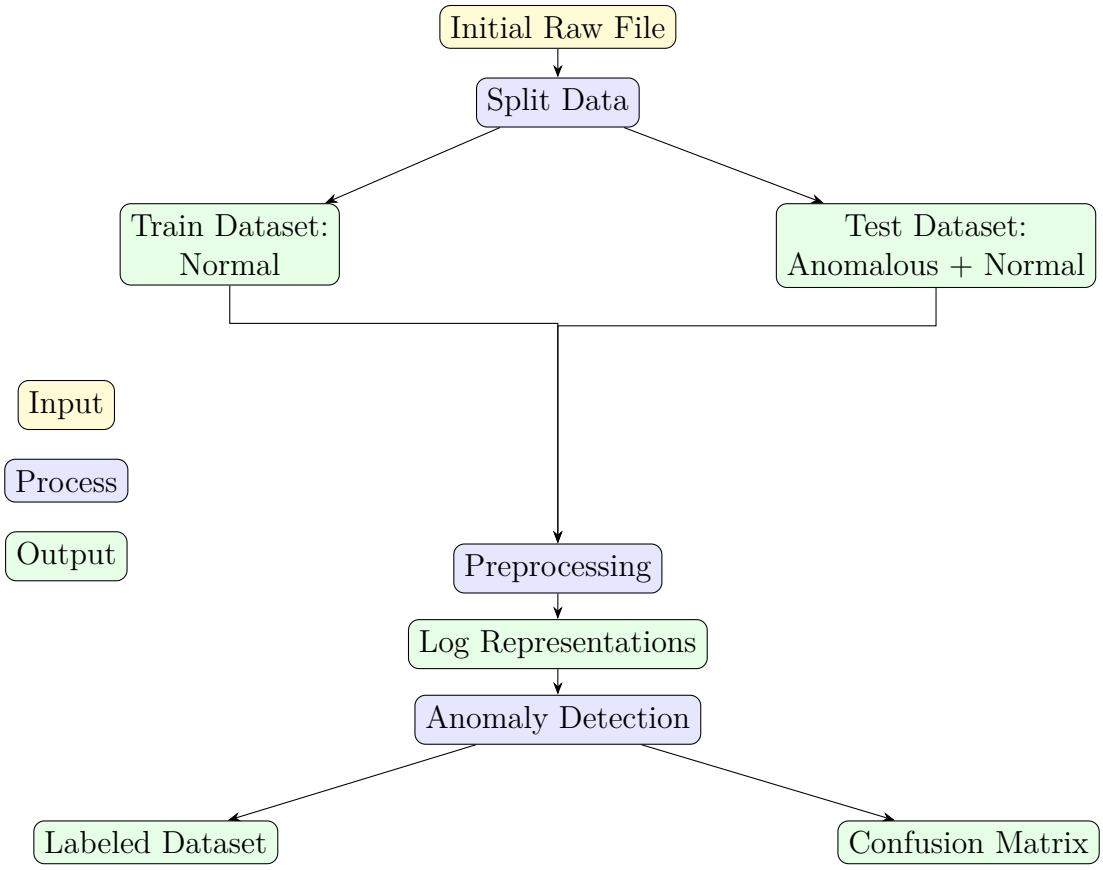
5.4.2 Anomaly detection

Several experiments were conducted to produce the results detailed in the following chapter. These experiments primarily involved parameter tuning and the utilization of various fine-tuned models to generate representations. The setup employed for anomaly detection, utilizing the fine-tuned DistilBERT model, along with a process chart, is presented below in Figure 5.3.

Table 5.1: Example of configuration parameters employing the MLM fine-tuned model

Process	**Parameter**	**Value**
split_data	split	0.65
preprocessing	model	"MLM_fineTuned"
anomaly_detection	core_set	0.005
anomaly_detection	threshold	ROC

Figure 5.3: Processes and outputs flow for the anomaly detection pipeline



6

Results

In this chapter, the most relevant results gathered throughout the project will be presented. The first section will cover the results yielded by various fine-tuning techniques. The second section will present and compare the anomaly detection results, along with a brief analysis of the mislabeled logs. Finally, we will discuss some additional experiments that may shed light on the performances included in the previous sections.

6.1 Fine-tuning DistilBERT

While various fine-tuning configurations were tried, this chapter will include results for the fine-tuning done on the DistilBERT model optimized for Masked Language Modeling (MLM), as this solution yielded the most meaningful results. The other methods included fine-tuning on a one-class sentence classification task and using a spherical objective function. The first method entailed training a DistilBERT model with an extra layer for sentence classification. It was fine-tuned to output a class, either 0 or 1, however, the extracted embeddings from this last layer turned out to be significantly less representative than the embeddings from the pre-trained DistilBERT model. A spherical objective function was also defined, and utilized to fine-tune the standard DistilBERT model with the hope of "pooling" the embeddings of normal lines closer together, thus widening the divide between normal and anomalous lines. However, when these approaches were tested, they yielded results that were either too poor or not instrumental for log anomaly detection and were therefore not included in this chapter.

6.1.1 Fine-tuning on the Masked Language Modeling Task

This section focuses on the fine-tuning of the DistilBERT model for masked language modeling. Every layer of the model was frozen except for the last one, which was trained on a dataset containing normal log lines. The goal was for the model to gain a better understanding of normal log lines, leading to increased effectiveness in performing log anomaly detection. Figures 6.1 and 6.2 showcase the training dynamics of the model. The first plot 6.1 depicts the progressive decrease in both training and validation loss over 22 epochs, indicating effective learning and adaptation of the model to the task. Meanwhile, the second plot illustrates a very slight increase in training and validation accuracy.

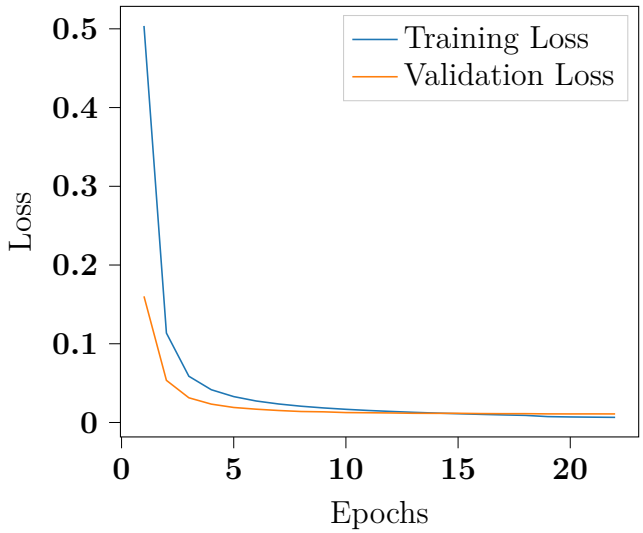


Figure 6.1: Comparison of Training and Validation Loss over 22 epochs. The plot depicts the decrease in loss during the fine-tuning of the DistilBERT model for masked language modelling. The convergence of both training and validation suggests effective learning

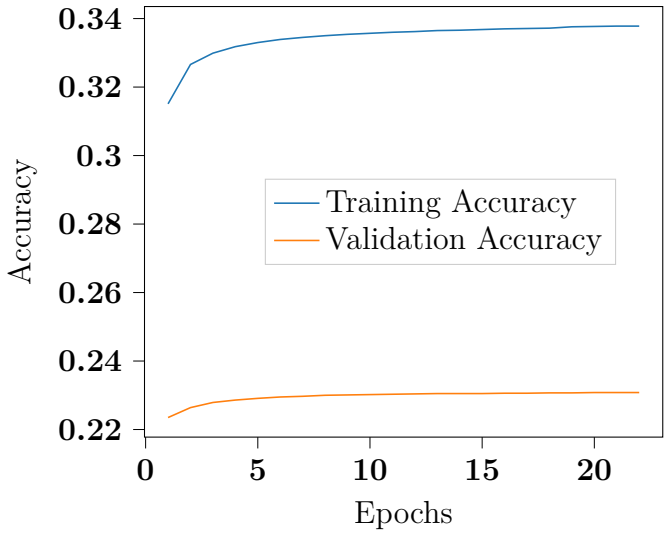


Figure 6.2: Comparison of Training and Validation accuracy over 22 epochs. The plot illustrates a slight increase in accuracy during training, with both training and validation accuracies approaching a plateau over successive epochs.

6.2 Anomaly Detection Results

In the following sections, the anomaly detection results will be presented. These will cover the confusion matrices and F1 scores for every method utilized to generate representations, as well as a brief analysis of the mislabeled log lines. To facilitate understanding of the results, the labels of the confusion matrices and their respective

meanings will be included in this section. Since this is a binary classification problem, normal log lines were defined as **"0"** and anomalous lines were assigned **"1"**, leading to the following equivalences. Throughout the results section, the focus will be on the incorrectly classified log lines.

- **True Anomalous (True Positives):** These represent the anomalous log lines which were correctly classified as anomalous, and will be referred to as **TA**.
- **True Normal (True Negatives):** These represent the normal log lines which were correctly classified as normal, and will be referred to as **TN**
- **False Anomalous (False Positives):** These represent the log lines which were incorrectly classified as anomalous, and will be referred to as **FA**
- **False Normal (False Negatives):** These represent the log lines which were incorrectly classified as normal, and will be referred to as **FN**

6.2.1 BERT vs DistilBERT

In Table 6.1, we compare the log anomaly detection performance of BERT and DistilBERT representations. Given that their results are identical, we opted to use DistilBERT throughout this project. This model is preferable due to its smaller size, resulting in faster and more efficient generation of representations.

Table 6.1: Comparison of BERT and DistilBERT when performing anomaly detection

Model	Precision	Recall	F1-score	FN	FA
BERT	0.95	0.94	0.94	2	10435
DistilBERT	0.95	0.94	0.94	2	10435

6.2.2 Detection Results when Utilizing the Pre-trained DistilBert Model

Tables 6.2 and 6.3 showcase the results obtained by the anomaly detection pipeline employing the pre-trained DistilBERT to generate the log representations with a core set of 0.005. The confusion matrix in Table 6.2 shows that the RAPID method rarely misclassifies an anomalous log line as a normal one. This is an important result since it means that very few anomalies are missed, with the method labeling two anomalous lines as normal out of 49,993 anomalous lines. However, the table also highlights that around 10,000 log lines are labeled as anomalous when they are in fact normal lines (high false anomalies).

In real-world scenarios, correctly detecting anomalies is more crucial than misclassifying normal lines as anomalous since missing an actual anomaly could lead to severe consequences such as security breaches, or system failures, whereas a false positive might simply result in additional checks or minor inconvenience. However

while it is indeed preferable to misclassify normal lines rather than the opposite, there is still room for improvement.

Table 6.2: Confusion matrix using a core set of 0.005 (45 unique log lines) and ROC as a threshold.

	Predicted Normal	**Predicted Anomalous**
Actual Normal	113449	10435
Actual Anomalous	2	49991

Table 6.3: In depth results of the best anomaly detection result on the Ericsson dataset.

	Precision	**Recall**	**F1-score**
Normal	1.00	0.92	0.96
Anomalous	0.83	1.00	0.91
Weighted avg	0.95	0.94	0.94

6.2.3 Comparing Every Approach

In Table 6.4, we present the results from various methods. The same dataset and split were used for all DistilBERT-based approaches, and a very similar split was utilized for the baseline model to ensure a fair comparison. The split could not be identical due to the baseline method being supervised and requiring both normal and anomalous data for training. As shown in the table, the F1 scores for the DistilBERT-based techniques are consistently 0.94, while the baseline model performs slightly worse with an F1 score of 0.90. However, the F1 score does not fully reflect actual performance, especially in real-world scenarios. We observe significant variation in the number of false normal, anomalous lines incorrectly classified as normal, ranging from 2 to 1413 for the MLM fine-tuning technique. This table makes it clear that the best method remains the one described in [6], which utilizes representations from the last layer of a pre-trained model.
In the following sections, we will investigate the reasons behind the lack of improvement from fine-tuning by analyzing the results of the fine-tuned models using a different method for calculating the similarity between logs.

Table 6.4: Comparison of classification results

Model	**TN**	**FN**	**TA**	**FA**	**F1**
PLM	**113449**	**2**	**49991**	10435	**0.94**
MLM fine-tuning	113926	1413	48580	**9955**	**0.94**
Baseline	113621	8035	41958	10263	0.90

6.3 Understanding the Results

In this section, we perform experiments and analyze the incorrectly labeled log lines to determine if there is room for improvement and to understand the effects of the fine-tuning processes.

6.3.1 Analysing the Composition of Misclassified Log Lines

We have analyzed misclassified log lines to better understand the reasons behind the results obtained by the best-performing model. The hope was that the model was incorrectly classifying a small subset of unique log lines. However, as shown in Table 6.5, the misclassified log lines are mostly unique, with 1368 of these appearing more than once and only 41 lines appearing more than four times.

Table 6.5: Composition of misclassified log lines when performing log anomaly detection with the pre-trained distilbert model and a coreset of 0.005

Statistic	**Count**
Misclassified lines	10086
Unique lines	8164
Multiple appearances	1368
Multiple appearances > 4	41

6.3.2 Removing Tokens

For this first set of experiments, we removed tokens from the start and end of each line to see which parts are more crucial for effective detection. In Figure 6.3, we removed 3, 6, 9, and 12 tokens from the start of the log lines, while in Figure 6.4, we removed them from the end. Comparing these figures, we notice that removing tokens from the end has a much more drastic effect than removing tokens from the start, even when removing only three tokens. This result aligns with our expectations since the beginning of each log line is usually very similar between lines.

Figure 6.3: Comparison of false anomalies (lines that are actually normal but were classified as anomalous) and false negatives when removing n tokens from the beginning of each log line and performing detection.

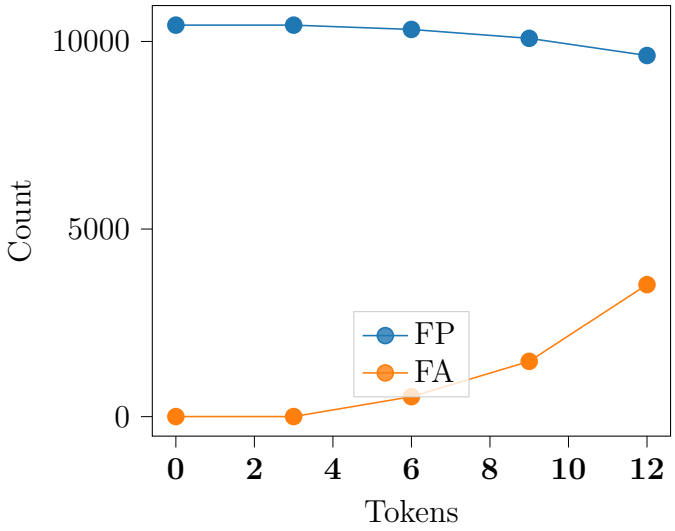
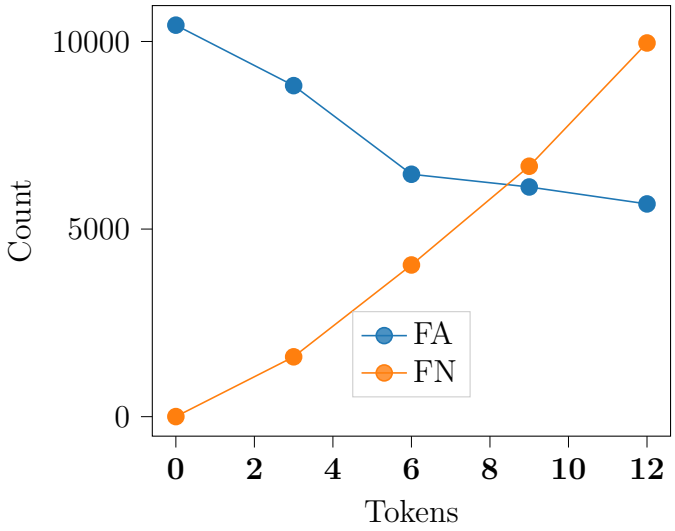


Figure 6.4: Comparison of false anomalies (lines that are actually normal but were classified as anomalous) and false negatives when removing n tokens from the end of each log line and performing detection.



6.3.3 Detection Results Employing Mean core_set Similarity

This section showcases the results of log anomaly detection employing the mean coreset distance. These results were yielded by both the PLM and the MLM fine-tuning. The increase in performance for the representations generated by the fine-tuned model might indicate that, while this type of fine-tuning does not improve performance when simply taking the closest cosine similarity, it could be useful for different approaches.

Figure 6.5: F1 score improvement as core_set size decreases employing the PLM DistilBERT model

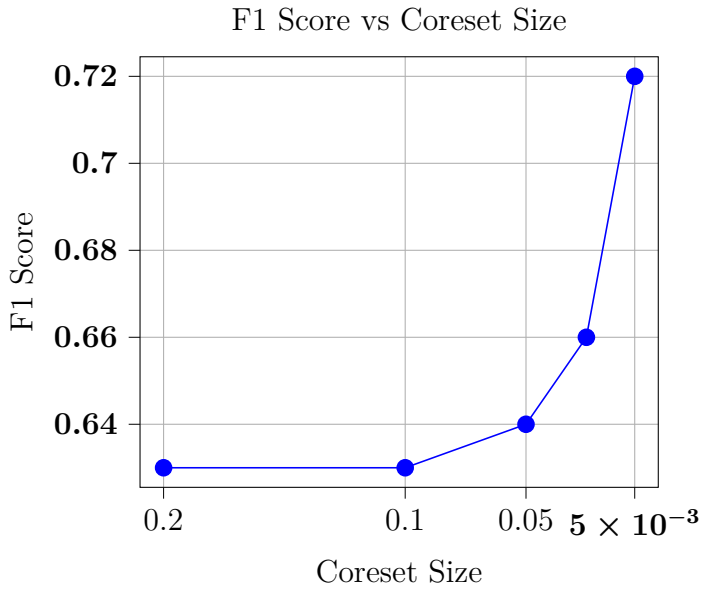
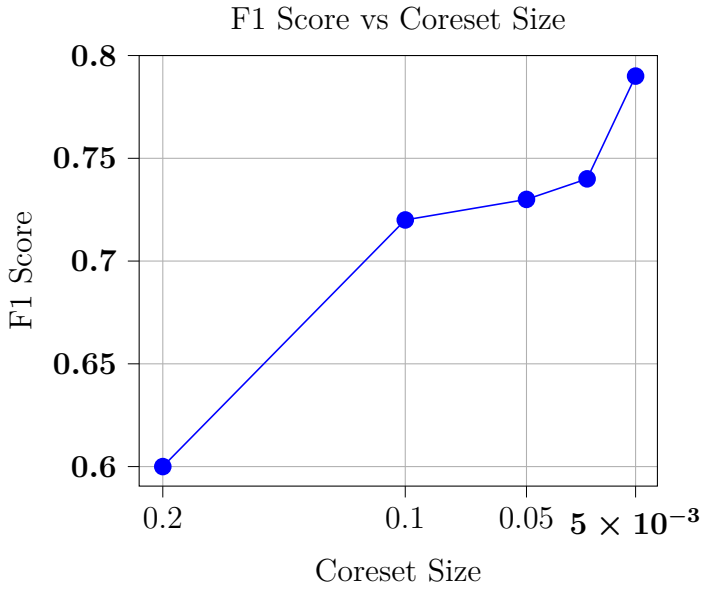


Figure 6.6: F1 score improvement as core_set size decreases employing the MLM finetuned DistilBERT model



6.3.4 Human Performance

For this experiment, we selected 21 random log lines from the testing dataset and asked experts at Ericsson to label them as anomalous or not. The log lines were pre-processed to hide sensitive information. Table 6.6 compares the performance of two experts (Expert1 and Expert2) and a the best performing variation of the RAPID method (PLM). We notice how, although with a very limited dataset, the

model completely outperforms manual classification. However, it is important to highlight that, while both "Experts" are familiar with and work with Ericsson log data, their daily work does not directly involve troubleshooting log lines. Thus, troubleshooters would likely achieve better results, and comparing their performance with the RAPID method's performance would definitely produce more meaningful results.

Table 6.6: Comparison of TA, FA, FN, TN, and F1 Score for Experts and Model

	TN	**FN**	**TA**	**FA**	**F1 Score**
Expert1	11	6	0	4	0.0
Expert2	12	3	2	4	0.36
PLM	14	0	6	1	0.92

7

Discussion

In this chapter, several considerations that stem from the results are presented. We stress the importance of replicating results and the necessity for rigor when presenting them. The value of fine-tuning and utilizing language models for anomaly detection is explored as well as some remarks on the environmental and ethical concerns that come with machine learning models.

7.1 Importance of Result Replication in Comparative Analysis of Novel Techniques

Our initial experiment aimed to reproduce the findings outlined in the original RAPID paper [6] on the BGL dataset. We successfully replicated the reported F1 score of 0.9999. While achieving this result might appear straightforward, its significance should not be underestimated. Throughout our research process, we observed several inconsistencies in the reported results of various papers on log anomaly detection, even when applying the same approach to identical datasets. For instance, in table 2 section 4.2 of the paper presenting LogBERT [42] we observe an F1 score of 0.7408 for the LogAnomaly approach on the BGL dataset, however in figure 6 section 4.2 of the LogAnomaly paper [49], the same F1 score is reported to be 0.96. While some variance is expected when employing machine learning models, discrepancies of this magnitude, on a classification task on large datasets, could potentially represent the misclassification of millions of lines. Moreover, more recent papers such as RAPID [6], do not directly replicate the results, but simply report the scores from older publications such as LogBERT [42], thus perpetuating any possible mistakes.

7.2 Assessing Log Anomaly Classification Results

The ability to successfully replicate the results should definitely be the standard, however it may not be sufficient for log anomaly detection papers such as [5], [8], [45] which heavily focus on the F1 metric when determining the performance of their approach. From the results included in chapter 6 table 6.4, it becomes evident that the F1 metric, with a precision of two decimal values, lacks the necessary precision to accurately distinguish differences in performance between approaches. For instance, in table 6.4 both PLM and MLM yield an F1 score of 0.94, upon closer examination of the number of false negatives, we notice a discrepancy of 1411 log lines. This

entails that over one thousand anomalies were classified as normal log lines while maintaining the same F1 score. In addition, this issue becomes even more evident when dealing with extremely large datasets with millions of lines such as the BGL dataset. Therefore, it is necessary to also include a confusion matrix, together with the standard evaluation metrics for classification tasks, in order to faithfully portray any results.

7.3 Exploring the Value of Model Fine-Tuning

Throughout this project we explored several fine-tuning techniques, different hyperparameter configurations and a number of DistilBERT variations. None of our experiments yielded an improvement in performance when compared to the pre-trained DistilBERT model included in the Huggingface library [39]. There could be two main reasons behind these results, either the appropriate techniques were not explored, or perhaps fine-tuning is not justified, at least in the context of this particular Ericsson dataset.

7.3.1 The performance/time trade-off

Fine-tuning the DistilBERT model to generate more faithful representations of log data has objective trade-offs. Developing a training pipeline adds complexity to the entire log anomaly detection process and increases, potentially by a large factor, its overall duration. In addition, it might make the method less generalizable [50], as the content in log datasets can vary greatly. Therefore, it should only be implemented if a substantial performance improvement is noticed.

The performance of the RAPID method and the quality of the representations generated by all the fine-tuning variations that were tried did not improve when using the cosine similarity as a metric, as shown in table 6.4. However, when employing the mean cosine similarity together with the fine-tuned model, we saw a clear improvement in the F1 score when compared to the pre-trained model fig. 6.6, fig. 6.5. Despite this approach still yielding poorer results than the standard RAPID configuration [6] and requiring longer times, its results may indicate that fine-tuning can indeed have a positive effect on the detection pipeline with a suitable structure.

7.3.2 Fine-tuning on new knowledge

There could be a fine-tuning technique or different method configurations (employing a different model, fine-tuning on a larger dataset) which substantially improve the results in the context of training-free approaches. However, recent publications, such as [51], point out that fine-tuning LLMs on new knowledge might lead to an increase in "hallucinations", these refer to a large language model's tendency to make up facts. While this might seem outside the scope of our project, hallucinations are closely tied to the model's representation space [52]. According to the paper by Gekhman et al. [51] LLMs have difficulty acquiring new factual knowledge through fine-tuning because they learn examples that introduce new knowledge much more

slowly than examples that align with what they already know. Although this research utilized a different model than BERT, namely PaLM 2-M [53], it highlighted an issue that is corroborated through our results. Since log data significantly differs from the training data utilized for BERT [29], the decrease in performance yielded by our experiments could be the "natural" consequence of fine-tuning. If this was the case, fine-tuning natural language models like DistilBERT on log data may not be appropriate.

7.4 Comparing the Effectiveness of Simple Statistical Approaches and LLMs

Machine learning approaches have been shown to perform extremely well on log anomaly detection tasks [4], [5], [45], however, it is not clear if the trade-off between time and performance is justifiable. In table 6.4, we show that a simple method involving BoW and naïve bayes has a comparable performance to the LLM-based RAPID method when tested on the Ericsson dataset. The same approach reaches almost perfect results A.1 when tested on much less diverse and simpler datasets such as the BGL dataset. These results, and the ones included in the literature [4], [42], [45], suggest that a completely general approach has not been found yet. Therefore, it may be suitable to choose an appropriate method of log anomaly detection based on various factors such as a comprehensive data analysis on the dataset, the presence of a labeled dataset and eventual time constraints. Based on these crucial aspects it could be possible to discern whether simpler statistical models could be sufficient or if more complex solutions are necessary.

7.5 Environmental and ethical concerns

Throughout this report, we show that machine learning solutions are being implemented to solve a variety of issues, including log anomaly detection. While these approaches can perform extremely well, it is wise to always take the performance/-time trade-off into consideration. However, it is also crucial to reflect on the environmental impact of solutions that utilize large models. In a recent interview, the CEO of Meta, the Artificial Intelligence behemoth and maintainer of the popular machine learning library PyTorch [54], emphasized that the biggest bottleneck in AI development is energy constraints [55]. Thisx is extremely concerning, as the World Economic Forum report that the energy required to train and utilize AI solutions is accelerating with a growth rate between 26% and 35% [56].

We estimate that, among various iterations and testing pipelines, we have utilized around 60 hours of GPU time. With an average wattage of 50W [57] we calculate a usage of 3kWh. While this is not an enormous amount of energy, it is still several order of magnitude more than utilizing the baseline method, which was trained and tested multiple times on a cpu, taking overall less than 5 minutes. Since Ericsson data centres are located in Sweden, one of the most sustainable countries on earth, our total greenhouse gas emissions amount to 21g, based on the Swedish greenhouse

gas emission intensity (g CO_2e/kWh) of 7g CO_2e/kWh reported in 2022 by the European Environment Agency [58]. However, based on the same report, if a data center in Germany was utilized, the total emissions of this project would amount to around 1098g, 50 times more than using a data center in Sweden. Our project was minuscule compared to some AI products that are being released every day, therefore, it is crucial to not only focus on the between performance and time or cost but also take energy consumption and carbon emissions into consideration.

8

Conclusion

The need for automated methods using ML and AI has become apparent since manual methods have become impractical, due to the exponential growth in log data volume and complexity [2]. In this thesis, we implement the RAPID log anomaly detection method [6] and adapt it to an Ericsson log dataset, by implementing various pre-processing and fine-tuning approaches. Therefore, our primary goals were to develop a fast and accurate anomaly detection pipeline and to examine the impact of fine-tuning on LLM embeddings for domain-specific log data.

The RAPID framework utilizes pre-trained language models to extract log representations and employs a retrieval technique to compare test logs with normal logs. This approach avoids the need for model training, making it both efficient and effective for anomaly detection.

Our experiments successfully replicated the results of the original RAPID paper, achieving an F1 score of 0.9999 on the BGL dataset. However, inconsistencies in reported results across various papers highlight the need for rigorous result replication and the inclusion of confusion matrices alongside standard evaluation metrics to accurately portray performance. Additionally, we successfully obtained an F1 score of 0.94 on the Ericsson dataset using the pre-trained DistilBERT model.

Despite our efforts, fine-tuning techniques, such as fine-tuning DistilBERT on a Masked Language Modelling task, did not yield performance improvements when compared to the pre-trained model. The performance/time trade-off and potential decrease in model generalizability suggest that fine-tuning may not be justified for our specific dataset [50]. Additionally, recent research [51] indicates that fine-tuning LLMs on new knowledge may introduce "hallucinations" and reduce model effectiveness. During our research we noticed that while the F1 metric is commonly used in log anomaly detection papers, it may not be sufficient to accurately distinguish differences in performance. For example, both PLM and MLM approaches yielded an F1 score of 0.94, but a closer look revealed a discrepancy of 1,411 false negatives. This issue is particularly significant with large datasets like BGL. Therefore, it is essential to include a confusion matrix alongside standard evaluation metrics to more faithfully portray results.

In conclusion, while the RAPID framework and LLM-based methods offer promising solutions for log anomaly detection, the trade-offs between performance, time, and environmental impact must be carefully considered. Simple statistical methods,

such as Bag of Words (BoW) combined with naïve Bayes, can perform comparably to LLM-based methods like RAPID, especially on simpler datasets. This finding suggests that the choice of log anomaly detection method should be based on comprehensive data analysis, the presence of labeled data, and time constraints. Future work should continue to explore these trade-offs and seek more sustainable and efficient methods for log anomaly detection.

Bibliography

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

[2] M. Landauer, S. Onder, F. Skopik, and M. Wurzenberger, "Deep learning for anomaly detection in log data: A survey," *Machine Learning with Applications*, vol. 12, p. 100 470, 2023.

[3] Y. Lee, J. Kim, and P. Kang, "Lanobert: System log anomaly detection based on bert masked language model," *Applied Soft Computing*, vol. 146, p. 110 689, 2023.

[4] 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*, Oct. 2017, pp. 1285–1298.

[5] X. Zhang, Y. Xu, Q. Lin, *et al.*, "Robust log-based anomaly detection on unstable log data," ser. ESEC/FSE 2019, Tallinn, Estonia: Association for Computing Machinery, 2019, pp. 807–817, ISBN: 9781450355728. DOI: 10.1145/3338906.3338931. [Online]. Available: https://doi.org/10.1145/3338906.3338931.

[6] G. No, Y. Lee, H. Kang, and P. Kang, "Rapid: Training-free retrieval-based log anomaly detection with plm considering token-level information," *arXiv preprint arXiv:2311.05160*, 2023.

[7] S. Kumar, "Classification and detection of computer intrusions," Ph.D. dissertation, Purdue University, 1995.

[8] [Online]. Available: https://www.lenovo.com/us/en/glossary/log/#:~:text=A%5C%20log%5C%20is%5C%5C%20a%5C%20file,keeping%5C%20track%5C%20of%5C%20user%5C%20activity..

[9] [Online]. Available: https://www.ibm.com/docs/en/zos/2.4.0?topic=problems-example-log-file.

[10] H. Sheikh, C. Prins, and E. Schrijvers, "Artificial intelligence: Definition and background," in *Mission AI: The New System Technology*, Springer, 2023, pp. 15–41.

[11] Z.-H. Zhou, *Machine learning*. Springer Nature, 2021.

[12] T. O. Ayodele, "Types of machine learning algorithms," *New advances in machine learning*, vol. 3, no. 19-48, pp. 5–1, 2010.

[13] M. Wang, L. Xu, and L. Guo, "Anomaly detection of system logs based on natural language processing and deep learning," in *2018 4th International*

Conference on Frontiers of Signal Processing (ICFSP), IEEE, 2018, pp. 140–144.

[14] Z. S. Harris, "Distributional structure," *Word*, vol. 10, no. 2-3, pp. 146–162, 1954.

[15] Y. Zhang, R. Jin, and Z.-H. Zhou, "Understanding bag-of-words model: A statistical framework," *International journal of machine learning and cybernetics*, vol. 1, pp. 43–52, 2010.

[16] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[17] K. Sparck Jones, "A statistical interpretation of term specificity and its application in retrieval," *Journal of documentation*, vol. 28, no. 1, pp. 11–21, 1972.

[18] D. D. Lewis, "Naive (bayes) at forty: The independence assumption in information retrieval," in *European conference on machine learning*, Springer, 1998, pp. 4–15.

[19] X. Wang, Y. Zhao, and F. Pourpanah, "Recent advances in deep learning," *International Journal of Machine Learning and Cybernetics*, vol. 11, pp. 747–750, 2020.

[20] S.-H. Han, K. W. Kim, S. Kim, and Y. C. Youn, "Artificial neural network: Understanding the basic concepts without mathematics," *Dementia and neurocognitive disorders*, vol. 17, no. 3, p. 83, 2018.

[21] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation, parallel distributed processing, explorations in the microstructure of cognition, ed. de rumelhart and j. mcclelland. vol. 1. 1986," *Biometrika*, vol. 71, pp. 599–607, 1986.

[22] S. Sharma, S. Sharma, and A. Athaiya, "Activation functions in neural networks," *Towards Data Sci*, vol. 6, no. 12, pp. 310–316, 2017.

[23] S. Narayan, "The generalized sigmoid activation function: Competitive supervised learning," *Information sciences*, vol. 99, no. 1-2, pp. 69–82, 1997.

[24] I. H. Sarker, "Deep learning: A comprehensive overview on techniques, taxonomy, applications and research directions," *SN Computer Science*, vol. 2, no. 6, p. 420, 2021.

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

[26] R. Pascanu, T. Mikolov, and Y. Bengio, "On the difficulty of training recurrent neural networks," in *International conference on machine learning*, Pmlr, 2013, pp. 1310–1318.

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

[28] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[29] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[30] C. Sun, X. Qiu, Y. Xu, and X. Huang, "How to fine-tune bert for text classification?" In *Chinese Computational Linguistics*, M. Sun, X. Huang, H. Ji, Z. Liu, and Y. Liu, Eds., Cham: Springer International Publishing, 2019, pp. 194–206.

[31] S. J. Pan and Q. Yang, "A survey on transfer learning," *IEEE Transactions on knowledge and data engineering*, vol. 22, no. 10, pp. 1345–1359, 2009.

[32] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *science*, vol. 313, no. 5786, pp. 504–507, 2006.

[33] G. Vrbani and V. Podgorelec, "Transfer learning with adaptive fine-tuning," *IEEE Access*, vol. 8, pp. 196 197–196 211, 2020.

[34] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson, "How transferable are features in deep neural networks?" *Advances in neural information processing systems*, vol. 27, 2014.

[35] E. J. Hu, Y. Shen, P. Wallis, *et al.*, "Lora: Low-rank adaptation of large language models," *arXiv preprint arXiv:2106.09685*, 2021.

[36] L. Xu, H. Xie, S.-Z. J. Qin, X. Tao, and F. L. Wang, "Parameter-efficient fine-tuning methods for pretrained language models: A critical review and assessment," *arXiv preprint arXiv:2312.12148*, 2023.

[37] C. Bucilu, R. Caruana, and A. Niculescu-Mizil, "Model compression," in *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2006, pp. 535–541.

[38] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," *arXiv preprint arXiv:1503.02531*, 2015.

[39] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, "Distilbert, a distilled version of bert: Smaller, faster, cheaper and lighter," *arXiv preprint arXiv:1910.01108*, 2019.

[40] Y. Liu, M. Ott, N. Goyal, *et al.*, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.

[41] Z. Chen, J. Liu, W. Gu, Y. Su, and M. R. Lyu, "Experience report: Deep learning-based system log analysis for anomaly detection," *arXiv preprint arXiv:2107.05908*, 2021.

[42] H. Guo, S. Yuan, and X. Wu, "Logbert: Log anomaly detection via bert," in *2021 International Joint Conference on Neural Networks (IJCNN)*, IEEE, Jul. 2021, pp. 1–8.

[43] R. B. Yadav, P. S. Kumar, and S. V. Dhavale, "A survey on log anomaly detection using deep learning," in *2020 8th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions)(ICRITO)*, IEEE, 2020, pp. 1215–1220.

[44] R. Chalapathy and S. Chawla, "Deep learning for anomaly detection: A survey," *arXiv preprint arXiv:1901.03407*, 2019.

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

[46] S. He, J. Zhu, P. He, and M. R. Lyu, "Loghub: A large collection of system log datasets towards automated log analytics," *arXiv preprint arXiv:2008.06448*, 2020.

[47] A. Oliner and J. Stearley, "What supercomputers say: A study of five system logs," in *37th annual IEEE/IFIP international conference on dependable systems and networks (DSN'07)*, IEEE, 2007, pp. 575–584.

[48] J. Zhu, S. He, P. He, J. Liu, and M. R. Lyu, "Loghub: A large collection of system log datasets for ai-driven log analytics," in *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*, IEEE, 2023, pp. 355–366.

[49] W. Meng, Y. Liu, Y. Zhu, *et al.*, "Loganomaly: Unsupervised detection of sequential and quantitative anomalies in unstructured logs.," in *IJCAI*, vol. 19, 2019, pp. 4739–4745.

[50] A. Aghajanyan, A. Shrivastava, A. Gupta, N. Goyal, L. Zettlemoyer, and S. Gupta, "Better fine-tuning by reducing representational collapse," *arXiv preprint arXiv:2008.03156*, 2020.

[51] Z. Gekhman, G. Yona, R. Aharoni, *et al.*, "Does fine-tuning llms on new knowledge encourage hallucinations?" *arXiv preprint arXiv:2405.05904*, 2024.

[52] Z. Xu, S. Jain, and M. Kankanhalli, "Hallucination is inevitable: An innate limitation of large language models," *arXiv preprint arXiv:2401.11817*, 2024.

[53] R. Anil, A. M. Dai, O. Firat, *et al.*, "Palm 2 technical report," *arXiv preprint arXiv:2305.10403*, 2023.

[54] A. Paszke, S. Gross, F. Massa, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[55] S. Moss, *Metas mark zuckerberg says energy constraints are holding back ai data center buildout*, Apr. 2024. [Online]. Available: https://www.datacenterdynamics.com/en/news/metas-mark-zuckerberg-says-energy-constraints-are-holding-back-ai-data-center-buildout/.

[56] Apr. 2024. [Online]. Available: https://www.weforum.org/agenda/2024/04/how-to-manage-ais-energy-demand-today-tomorrow-and-in-the-future/#:~:text=AI%5C%20and%5C%20energy%5C%20demand&text=The%5C%20energy%5C%20required%5C%20to%5C%20run,of%5C%20Iceland%5C%20used%5C%20in%5C%202021..

[57] [Online]. Available: https://www.nvidia.com/en-us/data-center/products/a2/#:~:text=A2%20delivers%20a%20low%2Dprofile,it%20ideal%20for%20any%20server..

[58] Oct. 2023. [Online]. Available: https://www.eea.europa.eu/data-and-maps/daviz/co2-emission-intensity-14/#tab-googlechartid_chart_41.

A

Appendix 1

Table A.1: Comparison of Baseline Classification Results

Model	TN	FN	TA	FA	F1
Baseline BGL	4344001	725	344278	11480	1.00
Baseline Ericsson	113621	8035	41958	10263	0.90