AI-driven Log Analysis for Intrusion Detection

Johan Sundin

Linus Särud

LUND UNIVERSITY

Department of Automatic Control

MSc Thesis
TFRT-6232
ISSN 0280-5316

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

Abstract

Today's security systems generate system logs that contain information about important events such as intrusion attempts and hardware failures. However, the large volume of data makes manual analysis impractical. Instead, this thesis proposes a method of using AI for classification.

Building on previous research, a transformer model has been integrated with a hyperspherical loss function and a Large Language Model (LLM). This combination handles the context of new logs and enhances the detection of anomalies. In collaboration with Advenica, the work contributes to the cybersecurity field by integrating a transformer model with a previously proposed embedding approach to create a model with better accuracy than previous approaches.

The model demonstrated an overall improvement in performance on both benchmark datasets (HDFS and BGL) when concept drift was not considered, with F1-scores of 0.931 compared to 0.766 for HDFS, and 0.952 compared to 0.694 for BGL. When concept drift was taken into account, the F1-scores were 0.831 compared to 0.807 for HDFS, and 0.871 compared to 0.721 for BGL.

Keywords: system logs, AI, transformer models, anomaly detection, cybersecurity, AIOps

Acknowledgements

We would like to thank Advenica, the company which this thesis was in collaboration with, and especially our supervisors at Advenica, Ola Angelsmark and Fanny Söderlund for guidance and support during the work.

We would also like to thank Johan Eker, our supervisor at LTH for providing us support and good feedback during the process.

Finally, we are grateful for the resources provided by WARA-Ops, a part of the WASP (Wallenberg AI, Autonomous Systems and Software Program). Their online Jupyter Notebook environment and provided computer power enabled us to evaluate the model using GPUs.

Contents

1
Introduction

System logs are generated by software applications, operating systems, and other devices to document events occurring within a system. These logs serve as a record of system activities, capturing everything from user interactions to security events and application behaviors. System logs are generated continuously while the system is running. The primary purpose of these logs is to provide a detailed description of the system's activities, enabling troubleshooting, performance monitoring, and security analysis.

Given that numerous activities on a system are recorded in these logs, monitoring them for anomalies can help detect issues ranging from intrusion attempts to hardware failures. However, in large, continuously operating environments, the volume of generated logs is substantial. Expecting any human worker to monitor all these outputs is unrealistic [29].

Consequently, log monitoring must be automated. Security tools such as Security Information and Event Management (SIEM) systems and Intrusion Detection Systems (IDS) play a crucial role in this process. SIEM systems collect and analyze logs from various sources within a network.

However, there are some complicating factors in this automation. Applications may vary how they log information between versions, due to developers fixing misspelled error messages, adding additional log statements, or introducing new features. This results in concept drift, where the characteristics of the log change over time. Ideally, a model used for anomaly detection would handle such minor changes without misclassifying them as anomalies.

1.1 Thesis Purpose

This thesis aims to build on and improve a previous research project by Murphy and Larsson [21], which initiated a pipeline between LTH and Advenica to create a log anomaly detection machine learning model. Their work primarily focused on the embedding space and was evaluated using simple models. We will enhance this by using the same embedding space in conjunction with a Transformer model.

The model's purpose is to detect and classify anomalies in computer networks and systems. Several previous works [2, 22] have demonstrated significant improvements when using an unsupervised Transformer for log anomaly detection compared to previously commonly used unsupervised models.

Our proposal is to use a Transformer-based model inspired by Logsy [22], integrated with the embedding approach demonstrated by Murphy and Larsson [21]. Their solution showed great resilience towards concept drift compared to their baseline models. We aim to create an even more robust and accurate full model.

Successfully creating and optimizing a model that can accurately and robustly identify anomalies from streams of log messages, along with other research, can help advance safer and more danger-aware systems. As previously mentioned, any modern digital system needs to protect itself from severe threats, and identifying these through automated log surveillance can be applied to any system with large-scale streams of logs.

This thesis aims to contribute to this field by developing and evaluating an unsupervised anomaly detection model with high accuracy and robustness, further enhancing the current solutions in this area. By utilizing a Transformer-based model, we aim to build upon the foundational work of previous research.

This thesis is supported by Advenica, a company based in Malmö, Sweden, that offers cybersecurity solutions to various companies and authorities to protect and enhance their information security. Detecting anomalies in system logs is highly relevant to Advenica, aligning with the company's focus on employing state-of-the-art technology for cybersecurity.

1.2 Related Work

NuLog [23] is a parsing technique that utilizes the transformer architecture [28] to generate embeddings from log entries. This approach is similar to the use of the ['CLS'] token, as described in Section 3.3.

Murphy and Larsson published a paper [21] in which they constructed an embedding space for system logs primarily by applying Lexicon-based Word Embedding Tuning (LWET) [14] to Word2Vec [18]. They also used Large Language Models (LLMs) to handle out-of-vocabulary cases. This embedding space was then combined with different models for anomaly detection. In this thesis, we adopt the same approach for generating the embedding space but employ a different model for anomaly detection.

Du et al. introduced a method called DeepLog [6], which uses a Long Short-Term Memory (LSTM) model to detect anomalies in system logs. This method predicts the next log entry and classifies it as an anomaly if it deviates from the prediction.

In 2020, Nedelkoski et al. presented their model Logsy [22], which leverages a transformer with a hyperspherical loss function to determine whether a window of data logs is normal or anomalous. This concept is the basis of the model used in our thesis as well.

An advancement of the Logsy model is Adlilog [2]. To address the issue of requiring a large amount of labeled data, they trained a model based on log instructions from over 1000 GitHub projects. This creates a general model that can then be fine-tuned for the specific target system.

Deep-loglizer [3] is a project by Chen et al. that implements various anomaly detection methods for comparison, including a Logsy-inspired transformer model. This project provided the initial code we used to develop our own model. The adapted transformer model by deep-loglizer forms the coding architecture base for the model used in this work.

1.3 Individual Contributions

While both authors have contributed equally to this thesis, some parts were naturally divided between the two. For example, Johan wrote the majority of the transformer background, while Linus wrote most of the preprocessing and data-handling code. Johan, however, wrote the code for out-of-vocabulary handling.

Before we had access to WARA-Ops, Linus was responsible for using Google Colaboratory and getting the model to work with rented GPUs. After gaining access to WARA-Ops, Johan experimented with the model and optimized the hyperparameters.

Although not all of them, most of the figures and illustrations were created by Johan.

2

Background

2.1 Log Messages

System log messages, often referred to as logs, are crucial components in the field of computing and information technology. They serve as detailed records of events and activities that occur within a computer system or network, providing a stream of information that can be used for monitoring, troubleshooting, and analyzing system behavior [11].

Logs are generated by various software components, and hardware devices within a computing environment. They carry a wide range of information, including system events, application activities, user actions, security incidents and hardware operations. Logs can function as warnings or error messages when issues arise, or in severe cases, they can indicate intrusions or unauthorized access attempts.

The frequency and volume of log messages can be substantial, reflecting the complexity and activity of the systems in which they are created. Today, the volume of logs being generated is increasing drastically due to the expansion of computer systems [11]. Hence, automating the detection and monitoring of log streams is becoming increasingly important.

An example of a sequence of 11 raw log messages from the Hadoop Distributed File System (HDFS) [4], explained in Section 2.3, can be seen in Figure 2.1.

The eighth log from the sequence in this particular example read as following:

```
081109 203741 181 INFO dfs.DataNode$PacketResponder:
Received block blk_-816514945136691 2526 of size 67108864 from
/10.251.90.239
```

This log message is constructed of six different parts seen in Listing 2.1.

The parts containing the most useful information of the log message for our case are parts 5 - 7, i.e. "Level", "Component" and "Content", with the last being the most important. Here, `blk_-8165149451366912526` is the block ID, `67108864` is the size and `/10.251.90.239` is the source ID. By removing the block ID, size and source ID tokens represented by numbers and replacing these with special

```
081109 203740 35 INFO dfs.FSNamesystem: BLOCK*
       ↪ NameSystem.addStoredBlock: blockMap updated:
       ↪ 10.250.5.237:50010 is added to blk_
       ↪ -6588618799865695917 size 67108864
081109 203740 35 INFO dfs.FSNamesystem: BLOCK*
       ↪ NameSystem.addStoredBlock: blockMap updated:
       ↪ 10.251.38.197:50010 is added to
       ↪ blk_1403510496212632306 size 67108864
081109 203740 35 INFO dfs.FSNamesystem: BLOCK*
       ↪ NameSystem.allocateBlock: /user/root/rand/
       ↪ _temporary/_task_200811092030_0001_m_000289_0/
       ↪ part-00289. blk_3517967462343156558
081109 203740 35 INFO dfs.FSNamesystem: BLOCK*
       ↪ NameSystem.allocateBlock: /user/root/rand/
       ↪ _temporary/_task_200811092030_0001_m_000368_0/
       ↪ part-00368. blk_7907271105957366348
081109 203740 35 INFO dfs.FSNamesystem: BLOCK*
       ↪ NameSystem.allocateBlock: /user/root/rand/
       ↪ _temporary/_task_200811092030_0001_m_000391_0/
       ↪ part-00391. blk_8909073982308380058
081109 203741 13 INFO dfs.DataBlockScanner:
       ↪ Verification succeeded for blk_
       ↪ -3443869707407490925
081109 203741 181 INFO dfs.DataNode$PacketResponder:
       ↪ PacketResponder 0 for block blk_
       ↪ -8165149451366912526 terminating
081109 203741 181 INFO dfs.DataNode$PacketResponder:
       ↪ Received block blk_-8165149451366912526 of size
       ↪ 67108864 from /10.251.90.239
081109 203741 182 INFO dfs.DataNode$PacketResponder:
       ↪ PacketResponder 1 for block blk_
       ↪ -8165149451366912526 terminating
081109 203741 182 INFO dfs.DataNode$PacketResponder:
       ↪ Received block blk_-8165149451366912526 of size
       ↪ 67108864 from /10.251.193.175
```

Figure 2.1 Example of 10 rows of log messages from the HDFS log dataset.

placeholder tokens, $< * >$, one acquires the following line:

```
Received block <*> of size <*> from <*>
```

This is the main template structure to be used when automatizing the parsing and log analysis. This is because the words and their meanings are the tokens containing the most semantic information in the log message. The placeholder token $< * >$ is

```
{
  "Date": "081109",
  "Time": "203741",
  "Pid": "181",
  "Level": "INFO",
  "Component": "dfs.DataNode$PacketResponder",
  "Content": "Received block blk_-8165149451366912526 of size
      ↪ 67108864 from /10.251.90.239"
}
```

Listing 2.1 Log Entry from the HDFS dataset

a way to symbolize varying parameters that are not included in the analysis of the log.

In contrast, a log message with another semantic meaning in this context could be the next log, that is the ninth log in Figure 2.1:

```
081109 203741 182 INFO dfs.DataNode$PacketResponder:
PacketResponder 1 for block blk_-8165149451366912526 terminating
```

After the masking, this log has the following corresponding version of the "Content" part:

```
PacketResponder <*> for block <*> terminating
```

A human can easily distinguish between two log messages based on their different contextual meanings. However, merely differentiating between log messages is insufficient to determine whether any of them is an anomaly. Identifying anomalies requires additional information and context. For instance, understanding whether a log message indicates a terminated process or an error could depend on the surrounding logs, which provide contextual information about the log's meaning. Transformer-based models have demonstrated significant potential in this area due to their ability to capture and interpret the semantic meaning of logs within their context, largely due to their attention mechanism, as explained in Section 2.5.

Anomalies

There exists different kind of anomalies, for example:

- **Point anomaly:** When the log entry itself is an anomaly just by itself.
- **Conditional (contextual) anomalies:** The log entry in itself is not an anomaly, but given the context it occurs in. This could be factors such as the time of day or if something else happening at the same time as the system is consuming large amounts of memory.

- **Collective anomalies:** A log entry is not an anomaly itself, but the collection of multiple log entries is.

In the scope of this thesis, all anomalies are handled the same in the sense that the proposed model is not able to signal what type of anomaly it has detected, just that it has found *an* anomaly.

2.2 Concept Drift

Concept drift is the phenomenon where the characteristics of the logged data change over time. This could be due to developers modifying log messages when developing the application, adding more logging to features, or a popular service getting more users, resulting in a higher frequency of certain messages. One of the desired properties of the model developed in this thesis is to warn about anomalies but not trigger an alert each time a new word appears in a log. Therefore, it is crucial to differentiate between what signifies an anomaly and what is normal concept drift.

In the work by Murphy and Larsson [21], concept drift was emulated by replacing certain words in the dataset of logs, simulating developers changing words in existing logging functions. While this scenario could occur in reality, it is only one of many ways concept drift can be introduced.

Concept drift generally refers to the target vocabulary changing over time. In the context of log messages, this could mean that an administrator has either added new lines of logs to the system or changed the structure or individual words of already existing messages. Therefore, it is important for an anomaly detection model to recognize this drift in concept in order to still understand that these possibly changed lines are not truly anomalies but modifications of the expected "normal" logs. Murphy and Larsson [21] emulated concept drift by substituting some predetermined words from the target dataset with synonyms. For example, "reset" would become "restarted", and "src" would become "source". While this approach catches the the concept of some individual words changing between the training and the evaluation data, it does not capture the other types of concept drift that could occur in a real-life setting. For instance, it does not emulate new log messages being added but only assumes that some words have been randomly substituted while keeping everything else the same. A more general approach and case to test would be to change the overall structure of a log message. For example

```
Received block blk_-8165149451366912526 of size 67108864 from
/10.251.90.239
```

could be restructured to

```
Block blk_-8165149451366912526 (magnitude 67108864)
transferred from source /10.251.90.239.
```

Simulating changes in the evaluation data in more ways would further test and showcase how well the model's resilience against concept drift really is.

An interesting situation and example that could be viewed as either concept drift or anomalous is if the number of different users suddenly starts to gradually increase. If the gradual increase is slow, a model with great resilience against concept drift might not catch this as anomalous because the change is not drastic enough to trigger any alarms. However, if the increase in the number of users and activity happens rapidly, the model could view this as anomalous because it deviates significantly from what it has learned to be normal.

The problem here is that both of these cases could be either anomalies caused by an external force attacking the system or simply more users being more active than before for some reasonable reason. This example highlights that distinguishing between concept drift and anomalies can be a gray area. It can sometimes be difficult to differentiate between the two. Therefore, training a model to handle concept drift as effectively as possible may not always be an advantage since it could obscure some potentially anomalous activities.

2.3 Datasets

The HDFS (Hadoop Distributed File System) [4] and BGL (Blue Gene/L Supercomputer) [24] datasets are commonly used in anomaly detection research. These datasets are labeled, making them advantageous for building and evaluating anomaly detection models using standard metrics.

- **HDFS**: Used for storing large volumes of data, the logs include file reads, writes, deletions, and other related activities as well as error messages.
- **BGL**: A supercomputer, the logs contain system events including normal operations as well as errors.

These datasets are frequently used in similar research, making them good candidates for benchmarking. For example, they are the same datasets that Murphy and Larsson [21] used in their work, allowing for direct comparison with their results.

The datasets are split into training, validation, and test sets according to Table 2.1. The anomaly contamination rates shown in the training data are for the supervised setting. When training the model in an unsupervised setting, without labels in the training data from the target dataset, the anomalous logs from the training data are not included. This is explained further in Section 3.3.

Advenica Dataset

In addition to the standard datasets used for benchmarking, an additional dataset of log messages provided by Advenica is also available. One of Advenica's products is a data diode, a physical device that sits between two servers and ensures that

Table 2.1 Number of log messages per dataset and split.

Set	HDFS	BGL
Training (Normal)	8,672,288 (97%)	1,815,454 (49%)
Training (Anomaly)	268,215 (3%)	1,889,554 (51%)
Validation & Test (Normal)	2,168,072 (97%)	351,976 (38%)
Validation & Test (Anomaly)	67,054 (3%)	574,276 (62%)

network traffic can only travel in one direction, allowing for network segmentation. On either side of the data diode, there is a device running proxy software: the device on the side allowed to send traffic through the data diode is called an upstream proxy, and the device on the receiving end is called a downstream proxy.

These devices run different services. When generating this dataset, the upstream server was configured to monitor a data directory. Whenever a new file appeared, it was sent over UDP through the data diode to the downstream proxy, where another service received and saved it. The setup is illustrated in Figure 2.2. The files were created with a script generating the training data. Anomalies were then added in the form of extra files being transferred over the data diode, potentially indicating data exfiltration.

The logs generated by the service running on the upstream server are also streamed through the data diode to the downstream service. The system logs are collected on the downstream server and therefore contain logs from both services as well as the system logs of the downstream server.

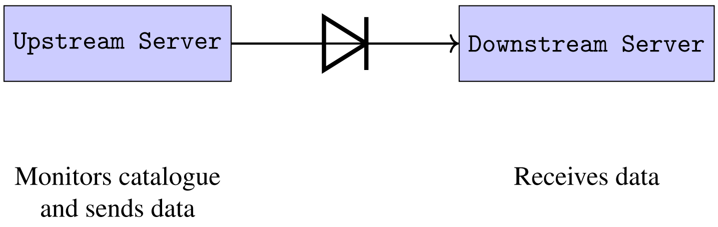


Figure 2.2 Schematic of the server setup with a data diode used to generate the Advenica data.

A hacker might use security tools to run a security scan against a server, such as a port scanning tool, to identify vulnerabilities or potential entry points in a target system. This process involves mapping out the attack surface by identifying running services and their versions, which the attacker can exploit later.

Applications or services that these security scans attempt to connect to might log activities to syslog. These logs may not necessarily indicate a hacker attempt directly but might show patterns such as repeated connections to different ports in quick succession or multiple login attempts.

Analyzing logs to detect these anomalies compared to normal behavior is crucial. One of the datasets provided by Advenica is therefore recorded while such a security scan was happening, allowing to evaluate if a model can detect these hacking attempts as anomalies in the system logs.

There are multiple ways to preprocess log messages. One approach is to filter the logs to use only those from the service in question and not from other applications. This method emulates a setup similar to the HDFS and BGL datasets, which individually contain data from a single service. However, the downside is that we lose information about the behavior of other applications on the same server, which might affect the model's anomaly prediction.

Another preprocessing method is to keep all information regardless of the source. The downside of this approach is that if information from other applications is less critical for prediction, it might introduce too much noise, leading to worse predictions. Given the same window size, such windows would contain less relevant information about the actual application.

2.4 Preprocessing & Embeddings

Drain

There are several common approaches to handling raw log messages. One popular method is *Drain* [8], which uses a fixed-depth parse tree to group different types of log messages by creating and assigning event templates. Drain works by first parsing the raw logs using custom rules to create log templates. For example, it transforms:

```
2023-05-17 12:00:00 INFO User 123 logged in from IP 192.168.1.1
```

into:

```
INFO User * logged in from IP *
```

These log templates are then grouped together using a decision tree that first branches based on the length of the log entry. The tree then splits again, this time based on the first token of the log entry. This process is repeated for the second token, and so forth, depending on the depth configured for Drain, as illustrated in Figure 2.3.

When the log entry reaches a leaf node, it must be determined which log group within this leaf node the log entry belongs to. This is done by calculating a similarity score between the log entry and each log group.

As each log group is given an ID and it is known how many log IDs are part of each group, it is possible, for example, to create a window of log entries and then convert that into an event matrix. This matrix indicates how many times a certain log group ID appears within the window, which can be used in an anomaly detection model.

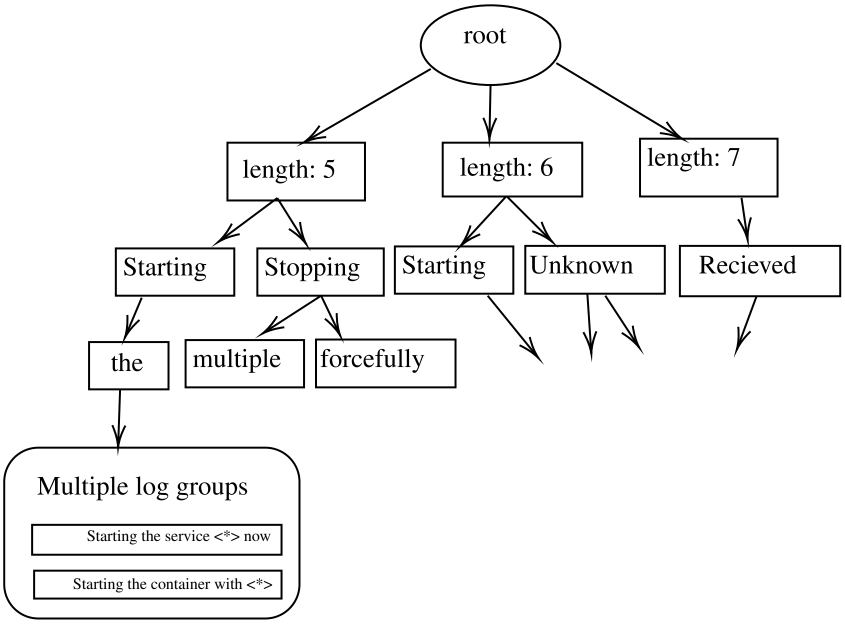


Figure 2.3 Illustration on how Drain [8] parse a log, with depth set to 4.

Embeddings

Another common approach is to give each token a unique integer vector representation of a predetermined length or dimension, i.e., to create *embeddings* or to *embed* each word. By embedding words, one can work with vectors of integers, which are generally more manageable in a coding environment. Together, the embeddings that make up the entire source vocabulary of a specific problem create an *embedding space*, denoted as **Q** in this thesis, which is a high-dimensional vector space with the same dimension as the predetermined embedding length.

Assigning a unique embedding to each word can be done in various ways depending on the task. Generally, one aims to place embedding vectors representing words that are alike close to one another in the embedding space. The definition of alike is task-dependent; for example, two words may be considered similar if they are spelled similarly, have the same length, or are synonyms.

When embeddings are used for anomaly detection in log messages, semantics are usually the most important aspect. This is because distinguishing "bad" keywords from "normal" ones is crucial for determining the full nature of the log.

Word2Vec [18] consists of a group of models built on shallow neural networks that create vector embeddings for a given corpus of words. It is built on the idea that words can have multiple degrees of similarity, i.e., words can be similar both syntactically and semantically. For example, syntactically similar words might include nouns with different word endings, such as "chair" and "chairs". Semantically similar words could be completely different in a syntactical sense but share similar meanings, such as "funny" and "humorous", or words that have a close relationship, such as "tree" and "leaf".

Murphy and Larsson [21] apply a transformation first proposed by Liu et al. [14] to the embedding space after the initial training and fitting using Word2Vec. The method is known as *Lexicon-based Word Embedding Tuning* (*LWET*), and it is based on lexical contrasts on word embeddings. The main purpose of applying LWET in this context is to more easily separate antonymous words in log messages while still preserving their relationships with other words in the embedding space. LWET minimizes a cost function by positioning synonyms closer together and antonyms further apart in the embedding space.

To apply LWET to the embeddings, one needs to provide synonymous and antonymous relations between the words in the vocabulary. Murphy and Larsson achieve this by merging two lexicons containing these relations. The first lexicon is gathered from *WordNet* [20], a widely used lexical database for the English language that organizes words into sets of synonyms called *synsets*. The second lexicon is created by utilizing a *Large Language Model* (*LLM*), mainly Meta's *LLaMa 2* model. The LLM lexicon is used to extract additional lexical relations between words that are not provided by WordNet, especially for more domain-specific words.

Out-of-vocabulary Words

Sometimes log messages contain words or tokens that are either misspelled, have alternative spellings, or are not real words, such as abbreviations or merges of other words. Since an embedding space is most often based on a large dataset of real-word vocabularies, these types of words do not have pre-made representations. These words are commonly called *out-of-vocabulary* words, or *OOV* words for short, and need to be considered separately. There are several approaches to handling OOV words. For example, Chen et al. [3] handle them by introducing a special ['OOV'] token with its own embedding, which is assigned to all OOV words. Another common approach is to use the *Levenshtein distance* proposed by Levenshtein et al. [12], which is a function that takes an OOV word and a vocabulary as input. It finds the word in the vocabulary closest to the OOV word, where "closest" means the smallest Levenshtein distance. The closest word is the one that requires the least amount of insertions, deletions, and substitutions to transform it into the OOV word.

Murphy and Larsson [21] propose another approach to handling OOV words. Their idea is that rather than only using the Levenshtein distance or giving each OOV word the same embedding, they have implemented a method that instead utilizes synonyms of the word. If an OOV word is found, they use a LLM to provide synonyms for the word. If synonyms are provided, it searches the vocabulary for these words and returns the corresponding embeddings if found. If that is the case, they create a new embedding for the OOV word as the mean vector of the respective embedding vectors for the synonymous words in the embedding space. If no synonyms are provided or found in the vocabulary, however, the OOV word instead gets its embedding vector from the word in the embedding space with the shortest Levenshtein distance from it.

Murphy and Larsson [21] show most improvement in comparison to their baseline by how resilient their approach is against concept drift. Concept drift refers to when the vocabulary and overall content of incoming log messages change over time, explained more in Section 2.2. For example, the word "terminating" might become "stopping". The problem lies in the fact that the model could be trained on a particular set of words or vocabulary, which will lose its performance when the semantics change over time, and the newer words are not included in or recognized by the trained model. By handling these new unrecognizable words due to concept drift in the same way that OOV words were described to be handled before, their results maintained high accuracy while the performance of other models worsened significantly.

2.5 Anomaly Detection Models

Automation of log anomaly detection has gathered attention due to the vast amount of log data generated by systems and applications, which makes manual analysis impractical. Several machine learning-based models have been developed and applied to effectively automate the detection of anomalies within log data. Among these, Autoencoders [7], Isolation Forests [13] and Logistic Regression [9] are important techniques. One of the newest and most relevant models of today is the Transformer model [28], which is the model that this work is mainly focused on. This thesis is handling every type of anomaly in the same way and therefore want to classify everything as either *normal* or *anomalous*, i.e. it is a *binary classification problem*, see Section 2.7.

Autoencoders

Autoencoders [7] are a type of unsupervised feed-forward neural network consisting of an input layer, a hidden layer and an output layer, each potentially containing multiple sub-layers. Figure 2.4 illustrates an Autoencoder with one input layer, one hidden layer and one output layer. The input to an Autoencoder is first processed through the input layer, where it is transformed into a latent space representation. The hidden layers then work to transform the latent representation back to the same dimensions as the input, aiming to reconstruct the original input data. The training phase utilizes a *loss function*, as explained in Section 2.7, to adjust model weights by minimizing the reconstruction error between the input and the output. During the testing phase, logs that significantly deviate from the normal pattern exhibit higher reconstruction errors, thereby signaling potential anomalies.

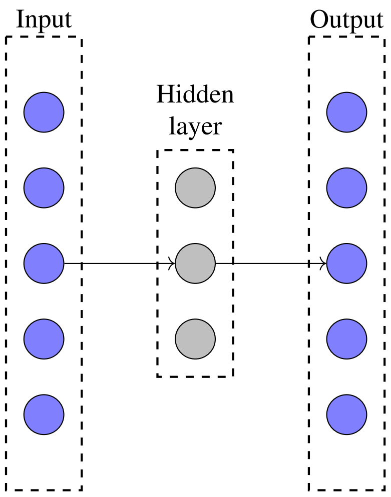


Figure 2.4 A simple depiction of an Autoencoder with one input layer, one hidden layer, and one output layer.

Isolation Forests

An Isolation Forest [13] is another unsupervised machine learning model that isolates different observations from the input data to a certain leaf. It operates on the principle that anomalies are few and different, and hence can be isolated more easily than normal instances. The model constructs each tree by isolating an instance by randomly selecting a feature and then randomly selecting a split value between the maximum and minimum values of the selected feature. Whether an instance is an anomaly or not is determined by the number of splits required to isolate it. Instances requiring fewer splits to isolate are considered anomalies, as illustrated in Figure 2.5.

Logistic Regression

Logistic Regression [9] is a supervised statistical method commonly used for binary classification tasks. This model can be adapted to identify anomalies by modeling the likelihood that a given log entry deviates from the normal behavior. The probability of an instance, i, being classified as an anomaly is calculated using a logistic function, as shown in Equation 2.1.

$$p_i = \frac{e^{\mathbf{x_i}\beta}}{1+e^{\mathbf{x_i}\beta}} \tag{2.1}$$

Here, $\mathbf{x_i}$ is the model input for instance i, β represents the trained model coefficients, and p_i denotes the probability of $\mathbf{x_i}$ being an anomaly. The model labels a testing instance as an anomaly if $p > 0.5$; otherwise it labels the instance as normal.

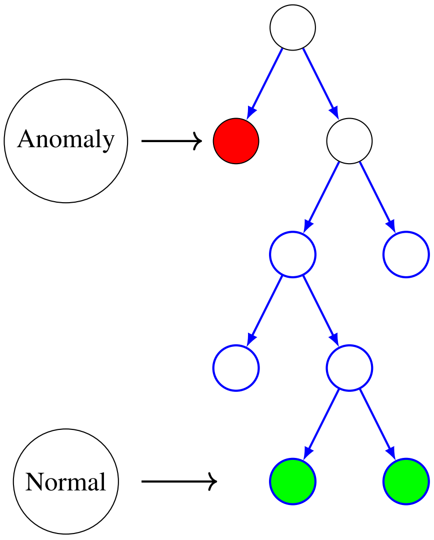


Figure 2.5 Visualization of an Isolation Tree of depth four. The red point, isolated with one split, is classified as anomalous and the green points, isolated with more splits, are classified as normal.

Transformers

Transformers are a machine learning architecture introduced in 2017 that has since revolutionized the field [28]. The architecture utilizes a self-attention mechanism and is highly effective in processing sequential data. Its ability to handle long-term dependencies in data makes it an ideal choice for log data analysis, as it allows for a long log window while effectively managing many features of each event. The full structure of the Transformer model is shown in Figure 2.6.

When the input embeddings reach the model, they are first processed by the *positional encoding* layer [28], which adds the incoming embedding vectors to vectors of the same size, d, that contains information about the position of each token in each sentence or row of logs. The method proposed by Vaswani et al. to represent this positional information in integer vector format involves applying sine and cosine functions in alternation, as shown in Equations 2.2 and 2.3.

$$PE_{(pos,2i)} = \sin(\frac{pos}{10000^{2i/d}}) \tag{2.2}$$

$$PE_{(pos,2i+1)} = \cos(\frac{pos}{10000^{2i/d}}) \tag{2.3}$$

Here, i represents the index in the embedding and pos is the position of the token in the sequence. This means that each dimension of the positional embedding vectors corresponds to a sinusoid. The indexing of the embedding dimension i ensures that

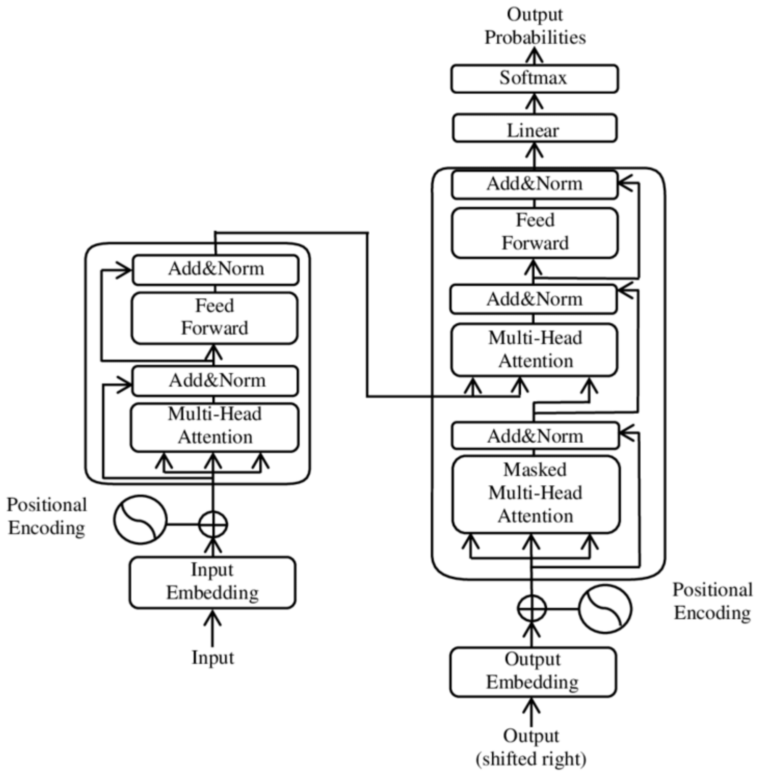


Figure 2.6 Overview of the Transformer model by Jia [10]. Licensed under CC BY-SA 3.0.

every other index is affected by the sine function and the alternate indices by the cosine function.

The purpose of the positional encoding function is to preserve a token's relative position to other adjacent or nearby tokens. Transformers process all tokens in parallel, which means they need additional information to understand the position of each token in the input sequence. Positional encoding helps the model to differentiate between the different positions in the sequence. The positional encoding method does not consider the embedding values of its input, only the internal positions, embedding indices, and total length of the input, as shown in Equations 2.2 and 2.3. Therefore, in an environment where each data instance is either padded or truncated to the same length, the additional contribution from the positional encoding algorithm will be exactly the same regardless of the contents of any instance.

The encoder part (left in Figure 2.6) consists of *N* parallel encoder layers, each containing two sub-layers: the *multi-head self-attention* sub-layer followed by a position-wise fully connected feed-forward neural network. Both of the sub-layers are followed by a layer normalization [1] of the sum of the sub-layer output, x^O, and the sub-layer input, x^I. Thus, the input to the layer normalization, x, is defined as $x = x^O + x^I$. The layer normalization function is shown in Equation 2.4.

$$\text{LayerNorm}(x) = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x]}} * \gamma + \beta \tag{2.4}$$

Here, γ and β are parameters learned during training.

Similarly, the decoder stack (right in Figure 2.6) also consists of sub-layers, including one position-wise fully connected feed-forward neural network [28]. The main difference from the encoder is that the decoder has two layers performing multi-head self-attention: one masked multi-head self-attention layer, which prevents positions from accessing subsequent positions, and a second layer applying normal multi-head self-attention to the output from the encoder stack. All three sub-layers are also followed by layer normalization as before.

Multi-head Self-attention

One of the most unique and revolutionary parts of the Transformer is the multi-head self-attention layer [28]. The primary purpose of attention is to identify context and relate tokens of the input log message to one another. For example, attention mechanisms can determine which verb belongs to which noun in a sentence. In the context of log messages, it relates entire log entries semantically to other log messages in their proximity.

The output from an attention mechanism — specifically, the *scaled dot-product attention* in this case — is the scaled and weighted sum of the values from key-value pairs. The input consists of query, key, and value variables as vectors with dimensions d_k (for queries and keys) and d_v (for values). By stacking the queries, keys, and values of a token batch as rows to create matrices Q, K, and V, respectively one can calculate the attention output for each token in a message simultaneously.

The attention is calculated by first multiplying the rows of Q with the rows of K, then dividing the results by $\sqrt{d_k}$ and applying the *softmax* function, see Equation 2.5, function to each row. Finally, this is multiplied by V to obtain the attention output. The attention function is shown in Equation 2.6.

$$\text{softmax}(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{d_k} e^{z_j}} \quad \text{for } i = 1, \ldots, d_k \tag{2.5}$$

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V \tag{2.6}$$

In the most general setting, the roles of the three matrices Q, K, and V are defined and sourced differently. The query matrix Q consists of columns each corresponding to a specific token in the attention scope. Each token's corresponding vector then usually has the role of asking questions about which words in its surroundings directly affects its meaning. For example, a noun might query, "Is there any adjective describing my attributes?".

The key matrix K also consist of token-specific vectors, similar to the query matrix. However, here the vectors corresponding to a token usually answer the questions stated by the query vectors of other tokens. Continuing with the same example, a key vector belonging to an adjective would inform the neighboring noun that it refers to it.

Lastly, the value matrix V is responsible for the final adjustment in embedding value for each token. By applying it to the results obtained from the query, key, and softmax operations, it transforms each embedding vector to a new position in the embedding space, ideally providing a better representation of the semantic and contextual meaning of every token.

As an example, consider a scenario where a token represents a full word. The word "train" would have an original embedding vector attempting to represent both the noun and the verb. When applying self-attention to the sentences "Traveling by train is my favorite form of transport" and "I train a lot for my upcoming football game", the embedding vector for the word "train" should be transformed in two different ways in the embedding space since its semantic meaning differs significantly in the two sentences. Similarly, a log message stating that is has received a package should be interpreted differently depending on whether it follows a message indicating a package request or not, assuming it is expected to follow such a request.

Since the matrices Q, K, and V are the vectors for each individual token of a sentence packed together, Q and K will have the dimensions $D \times d_k$, and V will be of size $D \times d_v$ [28]. The output from the attention function will then be another vector of dimension d_v. Here, D denotes the token length of each log message. In the context of log anomaly detection with embeddings, as discussed earlier, the most natural choice for the dimensions d_k and d_v is to have them all equal to d, because the output vectors are intended to be added to the embedding vectors in the same way as the positional encoding was before.

Vaswani et al. [28] then proposes that instead of using a single attention head, one can calculate the attention in parallel h times by projecting the queries, keys, and values onto the dimensions d_k and d_v, respectively, before calculating the attention. This process yields h outputs, each with dimensions d_v/h. After concatenating the outputs from each *head*, the result is an output of the full dimension. This method is referred to as multi-head self-attention, where h represents the number of heads.

One of the major advantages of using multiple heads instead of a single head is the ability to parallelize the mechanism. Using parallel layers allows the model to save a lot of computational resources and efficiency. It also provides a significant computational advantage by allowing the process to run in parallel on a GPU, thereby saving time.

The new function for the multi-head self-attention is defined in Equation 2.7, where the matrices W are the learnable weighted projection matrices. Specifically, W^O are the output projection weights, W_i^Q are the query projection weights, W_i^K are the key projection weights, and W_i^V are the value projection weights for each head $i = 1,...,h$.

$$\begin{aligned} \text{MultiHead}(Q,K,V) &= \text{Concat}(\text{head}_1,\dots,\text{head}_h)W^O \\ \text{where head}_i &= \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \end{aligned} \tag{2.7}$$

The sub-layer with multi-head self-attention in the encoder part of the Transformer receives its queries, keys, and values all from the previous layer, i.e., the input embeddings. In contrast, the second multi-head self-attention layer of the decoder only receives its queries from the previous layer, while the keys and values are obtained from the output of the last layer of the encoder. The masked multi-head self-attention sub-layer in the decoder receives all of its inputs from the previous layer, similar to the encoder.

Next Log

Another common approach, in contrast to classifying each log as either normal or anomalous directly, is to instead take a window of log entries and try to predict the next log entry. This method is used by Chen et al. [3], which they call *next log*. If the next log entry is among the top k predicted log entries, it is considered normal, where k is an integer parameter chosen by an operator. If not, the log entry is considered an anomaly. Intuitively, this approach handles contextual awareness and is a reasonable choice when the order of operations matters.

2.6 Logsy

While Vaswani et al. [28] published the first outline for the Transformer based machine learning model, *Logsy* — a Transformer based model for log anomaly detection proposed by Nedelkoski et al. [22] — was the first work to apply a Transformer to detecting anomalies in log messages.

In contrast to Vaswani et al.'s Transformer model [28], Logsy only uses the encoder part of the model in Figure 2.6 with multi-head self-attention for the specific application of log anomaly detection. The structure is therefore very similar to the original Transformer, except for the absence of the decoder. Instead, the output from the layer normalization after the feed-forward layer of the encoder represents the model's output.

An element of the Logsy model is the use of a special ['CLS'] token, which is pre-padded to the front of every message token sequence before entering the

Transformer. This ['CLS'] token is extracted to represent the entire log message sequence. This vector is then used as the sole input to the loss function during training and the objective function during evaluation and classification. In other words, the model is trained to provide the ['CLS'] token with an embedding such that the log can be classified correctly using this token.

The reason why Logsy and other projects that use a Transformer-based model for log anomaly detection omit the decoder side of the original Transformer model is because the decoder part is not necessary for this specific problem. The decoder's most practical use case is for *predicting* the next tokens in a sequence, such as in language translation or text generation (e.g., Chat-GPT). This becomes more apparent when considering the masking procedure in the decoder's attention mechanism. The main purpose of the masking is to prevent future tokens in a sequence from influencing the history, which is crucial when trying to predict something with a forward scope.

Therefore, it is sufficient to use and extract the encoder side of the Transformer model (as shown in Figure 2.6) since the attention mechanism there provides the semantic information over the entire sequence of tokens, which is the relevant part for the anomaly detection model. Since the goal is not to predict a new token each iteration through the model, there is always only one input to the Transformer each time it passes through the encoder, and no previous output needs to be processed by the decoder.

2.7 Loss Function

The loss function is a vital part of most machine learning models. Its primary purpose is to provide the model with information on how to adjust its parameters during training. This is achieved by minimizing the loss function during the training phase [5]. The characteristics and choice of a loss function vary depending on the specific task at hand. Some loss functions are better suited for classification problems, while others are better suited for regression tasks etc.

In this thesis, the problem is a binary classification problem, meaning the goal is to the best extent possible classify a sample (a window of log messages) as either normal or anomalous as accurately as possible.

Binary Cross-entropy Loss Function

One of the most commonly used loss functions for binary classification is the *binary cross-entropy loss function* [27]. This loss function is typically used together with a mapping function consisting of a linear layer followed by a sigmoid function for classifying the model output, $l(\phi(\mathbf{x_i};\theta))$. The sigmoid function, $\sigma(z)$, and the mapping function are defined in Equations 2.8 and 2.9, respectively.

$$\sigma(z) = \frac{1}{1+e^{-z}} \tag{2.8}$$

$$l(\phi(\mathbf{x_i};\theta)) = (1+\exp(-\mathbf{w^T}\phi(\mathbf{x_i};\theta)))^{-1} \tag{2.9}$$

The instance $\mathbf{x_i}$ is classified as anomalous if l is greater than 0.5 and as normal otherwise. In other words, an instance is classified based on what is deemed the most probable. Here, $\phi(\mathbf{x_i};\theta)$ is the model output, θ represents the model parameters, $\mathbf{x_i}$ is the input data, y_i are the labels, n is the batch size, and $\mathbf{w^T}$ are the weights of linear layer. The loss function is shown in Equation 2.10.

$$\begin{aligned} L(\dots) = -\frac{1}{n}\sum_{i=1}^{n}(1-y_i)\log((1+\exp(-\mathbf{w^T}\phi(\mathbf{x_i};\theta)))^{-1}) \\ +y_i\log(1-(1+\exp(-\mathbf{w^T}\phi(\mathbf{x_i};\theta)))^{-1}) \end{aligned} \tag{2.10}$$

Hyperspherical Loss Function

Another appropriate and relevant loss function for the log anomaly detection problem is the *hyperspherical loss function* [22]. This loss function projects the data onto a hypersphere, allowing it to identify anomalies that lie further from the dense region containing all the normal data. The mechanics and characteristics of this loss function fulfill the criteria of isolating two different target classes from each other by training the model parameters to place one class (normal) close to the origin, while placing the other class (anomalies) further away from the center. By then using a threshold, ε, the model can classify each sample based on whether the model output representation vector lies inside or outside the threshold. The hypersphere, with ideally placed data points (normal instances inside the sphere and anomalies outside the sphere), can be seen in Figure 2.7.

More concretely, the hyperspherical loss function can be expressed as Equation 2.11.

$$L(\dots) = \frac{1}{n}\sum_{i=1}^{n}(1-y_i)||\phi(\mathbf{x_i};\theta)||^2 - y_i\log(1-\exp(-||\phi(\mathbf{x_i};\theta)||^2)) \tag{2.11}$$

Here $\phi(\mathbf{x_i};\theta)$ is the model output, θ represents the model parameters, $\mathbf{x_i}$ is the data, y_i are the labels, and n is the batch size.

For classification, denote $A(\mathbf{x_i}) = ||\phi(\mathbf{x_i};\theta)||^2$. The criteria for a sample being classified as normal is if it scores $A(\mathbf{x_i}) < \varepsilon$, and as anomalous if it scores $A(\mathbf{x_i}) > \varepsilon$. In the rare case when the anomaly contamination rate in a labeled target training dataset is known, this information can be used to decide the threshold. If the ratio

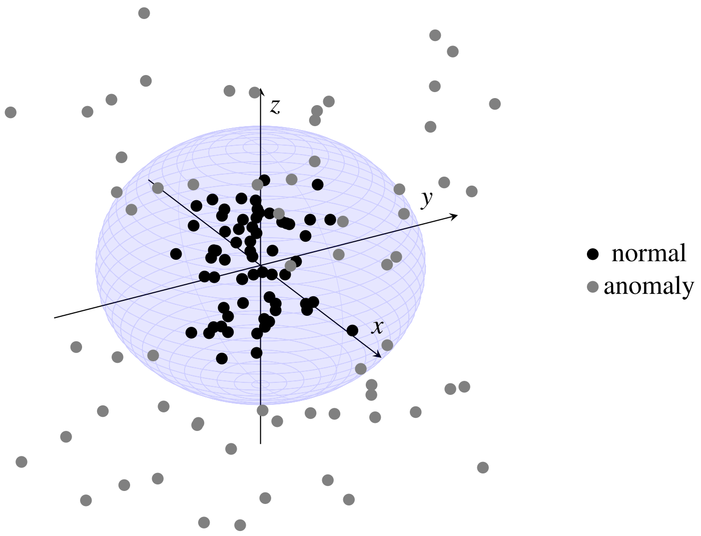


Figure 2.7 Visualization of a 3-dimensional hypersphere used for classification with normal and anomalous points ideally placed.

of anomalous samples is in the training data is known, the threshold can be set such that $c\%$ of the samples lie outside the threshold and the rest inside it, corresponding to $c\%$ of the training data being labeled as anomalies. The threshold from the best model is then saved as any other model parameter.

However, assuming this rate is known is most often unfeasible in real settings. Thus, an alternative method for determining the threshold is proposed. This method works for both labeled training data and the unsupervised case when auxiliary data is used to represent anomalies during training. This method does not require knowledge of the contamination rate or the labels in the validation data. The approach involves aggregating and taking the mean of all the model outputs $A(\mathbf{x_i})$ across all batches simultaneously on the validation data. Then, a factor of a standard deviation is added to or subtracted from the mean to determine the threshold. The choice of the factor to multiply the standard deviation by depends on the dataset and setup. Initially, this can be determined by observing the characteristics of the target training dataset.

For example, the HDFS dataset contains a majority of normal samples with few unique templates, causing their ['CLS'] embeddings to cluster closely together. This clustering places the mean somewhere in the "normal" area of the hypersphere. By adding a factor of the standard deviation to the mean, the threshold is expected to be placed outside the cluster of normal samples while still screening the outer anomalous samples from the center of the hypersphere. After observing the magnitude and absolute values of the model outputs for the HDFS dataset, it was concluded that this was a reasonable approach since the space between the larger clusters of normal and anomalous samples was large enough to typically place the threshold in the gap. The formula for choosing the threshold ε is defined in Equation 2.12.

$$\varepsilon = \mu + f * \sigma \tag{2.12}$$

Here, μ and σ are the mean and standard deviation, respectively, of the model output $A(\mathbf{x_i})$ for the validation data over all batches, and f is the dataset- and setting-specific factor.

The factor f for each dataset and setup can be seen in Table 3.1.

2.8 TF-IDF

Deep-loglizer [3] also included an approach and technique called *TF-IDF* [26] (Term Frequency-Inverse Document Frequency). TF-IDF is a statistical measure that evaluates the importance of a word in a document relative to a collection of documents, trained on all the training data. The idea is that by weighting terms, it is possible to identify the most informative features for distinguishing between normal and anomalous instances. This allows the model to focus on rare terms, which might be indicative of anomalies.

However, while this might have been a reasonable approach in the original deep-loglizer project, it did not combine well with the OOV-engine approach taken in this thesis. TF-IDF does not easily handle new words and therefore cannot assign a score to a word it has never seen before, such as a new word resulting from concept drift. Implementing a variant solution based on TF-IDF, where a score or similar is added to synonymous words, was considered. However, this was not implemented due to time constraints.

2.9 Windows

Instead of evaluating each log line individually, a more natural approach for identifying anomalous log messages is to incorporate the context in which they occur. An anomaly is often represented by a collection of logs rather than a single event. However, it is unfeasible to send the entire dataset to the Transformer at once. Therefore, the log data can be separated into different bundles of log lines, referred to as *windows*. These windows can be created in various ways, such as into *session windows* and *sliding window*.

Sliding Windows

A sliding window is created by slicing from the first index until it reaches a specified window size. Then, a step, known as the *stride*, determines where the next window begins. This means that a sliding window captures a specified number of events, maintaining the chronological order within the window. The stride could be set to one, implying overlaps between consecutive windows. A special case of this is a tumbling window, where the stride equals the window size, resulting in no overlap. The window size can be set in different ways depending on the task. For instance, it could be a fixed time span, such as one hour, or a fixed number of log entries per window.

Session Window

A session window includes all log entries stemming from the same *session*. A session can be defined differently depending on the log data. For example, it could be logs related to a specific login or user, or all activities related to a specific file on a file server. Normally, this window type does not have a fixed length since each session can vary in length.

In an online setting, this approach is less natural since a session might not be complete due to the continuous flow of new log messages. A potential solution is to define a session window as a window that includes log messages from a certain session but with a fixed size. This allows each session window to stay dormant in the data loader until it contains enough logs from the same session or until a certain time has passed before proceeding through the model.

Combining Sliding with Session Windows

The default approach of deep-loglizer [3] combines both of these window techniques. It first sorts all log entries by session. For the HDFS dataset, this is done per block ID. For example, all log entries associated with the block ID `blk_-8165149451366912526` from Figure 2.1 would create a session. For BGL, which does not contain natural or pre-made sessions, sessions are created based on the timestamps of each log message. This means that for BGL, each session is the same as a sliding window with a fixed time width. The same approach as was done with logs from BGL is used for the Advenica dataset, since it also lacks natural sessions or block IDs.

The model's predictions are then made on the sliding windows, each one being predicted as normal or anomalous. During evaluation, a session is defined as anomalous if it contains at least one log entry that is labeled as an anomaly. Thus, the performance metrics are calculated based on session predictions.

3

Method

The work will begin by utilizing existing log parsers and embedding algorithms, primarily using the method proposed by Murphy and Larsson [21]. The goal of this method is to develop of an improved and more efficient anomaly detection model compared to the Logistic Regression, Autoencoder, and Isolation Forest models used by Murphy and Larsson. The proposed model is a Transformer, inspired by Nedelkoski et al. [22]. Their Transformer model, *Logsy*, showed significant improvements compared to their baseline models, DeepLog [6] and PCA [30]. This is the main reason why we aim to use a similar Transformer model, but integrated with the log preprocessing work done by Murphy and Larsson to achieve even further improvements and better evaluation results.

Another important aspect is the evaluation of the developed model. This will be done using standard metrics such as precision, recall, and F1 scores, explained in Section 3.4. As explained in Section 2.3, the logs on which the model will be evaluated are from two freely available open-source datasets, HDFS and BGL, as well as a dataset provided by Advenica. The evaluation will also include exposing the model to concept drift, a challenge to which Murphy and Larsson [21] showed great resilience.

3.1 Environmental Setup

All programming was done in Python. Initially, the code was executed locally on MacBook laptops. However, due to insufficient computing power, we transitioned to using *Google Colab* and NVIDIA A100 GPUs via Paperspace.com for experimentation. Later, we gained access to WARA-Ops through Advenica, where the final models were developed and executed. Reliable access to the WARA-Ops environment allowed us to iterate on small changes without the need to rent and start up servers.

3.2 Preprocessing

The first step in constructing the Transformer-based log anomaly detection model is to create the embedding space, done almost identically to the method used by Murphy and Larsson [21]. The embedding space **Q** is trained using the Word2Vec [18] model on a combined corpus of words provided by *Text8* [19], a collection of the first 100MB of English Wikipedia articles containing 253,854 unique words, and the vocabulary acquired from the log dataset. This allows Word2Vec to train an embedding space on a large general collection of English words from Text8 and domain-specific words from the log messages.

After training is completed and a fully fitted embedding space is obtained, the LWET transformation is applied, as explained in Section 2.4, along with the merged WordNet and LLM lexicons containing synonymous and antonymous relations for the vocabulary.

However, at this point, Murphy and Larsson's method is no longer followed since they used a different approach from here onward. The primary interest lies in their method for creating the embedding space and their OOV engine for handling out-of-vocabulary words. To acquire the desired format of the log message representation for our model, their code is modified before the creation of event templates for each log message. Instead, it is slightly rewritten to only obtain the embedding space **Q**, from which each log message is represented as a sequence of embedding vectors for each of its tokens or words.

For this to work as input to a Transformer model, each log message must have an equal length. This is achieved by padding logs shorter than the desired maximum length and truncating those that are longer, ensuring all logs consist of D tokens. The token length D is set to 50, which is large enough to capture the majority of log messages' full lengths while keeping the model relatively small. Messages shorter than the maximum length are padded with a special [PADDING] token to represent the absence of information. The full proposed model architecture is presented in Figure 3.1.

Modified OOV Engine

When encountering OOV (Out-Of-Vocabulary) tokens, explained in Section 2.4, in the logs, Murphy and Larsson [21] uses a method that involves providing pre-generated synonyms from GPT responses, as an alternative to the use of the Levenshtein distance approach for assigning embeddings to OOV words. To automate this process while maintaining their approach, a GPT API was integrated. Specifically, we use *OpenAi's* GPT-3.5 Turbo, together with a function that requests synonyms for the incoming word based on Murphy and Larsson's Prompt No. 1, which they used for comparing GPT-3.5 and the *LLaMa 2* model [21]. Their prompt is as follows:

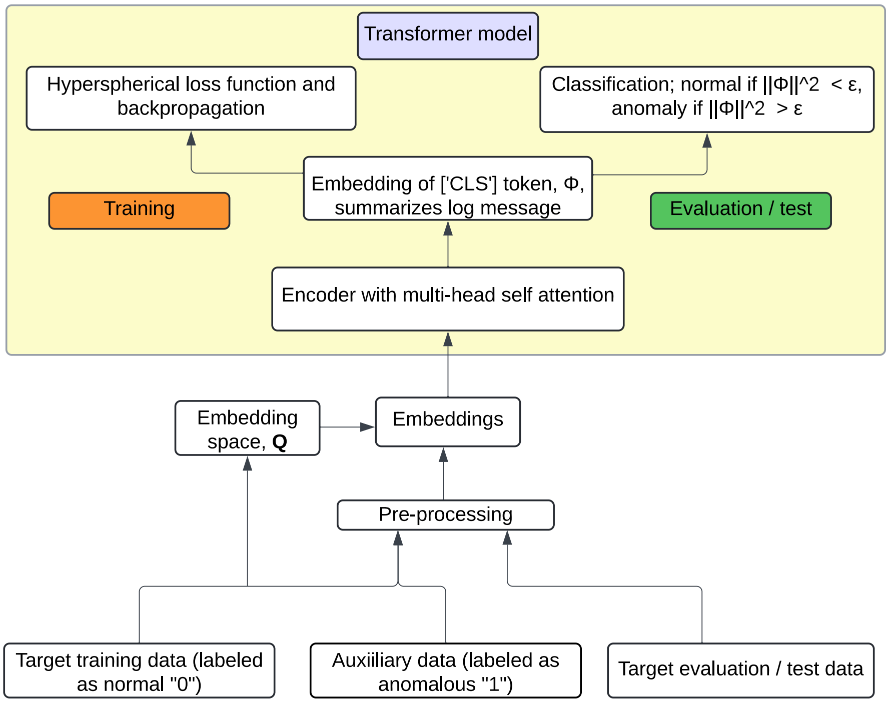

Transformer model
Hyperspherical loss function and backpropagation
Classification; normal if ||Φ||^2 < ε, anomaly if ||Φ||^2 > ε
Training
Embedding of ['CLS'] token, Φ, summarizes log message
Evaluation / test
Encoder with multi-head self attention
Embedding space, Q
Embeddings
Pre-processing
Target training data (labeled as normal "0")
Auxiliary data (labeled as anomalous "1")
Target evaluation / test data

Figure 3.1 Overview of the model architecture.

> *"Determine a maximum of 10 synonyms and antonyms each to the word "w". Respond in JSON format as 'synonyms': [], 'antonyms': []."*

To fit our usage and application, we modify the prompt to only provide synonyms, provide the word as a separate prompt, and respond with single words in lowercase. The reason for omitting antonyms from Murphy and Larsson's prompt is that our use case differs from theirs. In this context, we only need synonyms for OOV words to determine their embeddings. The original prompt is used for finding both synonyms and antonyms for the lexicon when applying LWET earlier in the process, as explained in Section 2.4. The modified version of the prompt is as follows:

> *"Determine a maximum of 10 one-word synonyms in lower case to the word provided. Respond in JSON format as 'synonyms': []."*

By using this prompt with the API, we can provide OOV words as input in real-time, automating the OOV engine. We are aware of a consideration here: using GPT-3.5 Turbo about half a year after Murphy and Larsson used their GPT responses from an older version of the engine means we might not be able to compare our API responses directly to theirs. This is because these engines, particularly OpenAi's, evolve very fast, and the responses we receive might differ in quality or content from those they acquired their results from. However, since this is only about finding synonyms, we assume that any differences are unlikely to significantly impact direct comparisons.

3.3 The Model

As a first draft of the anomaly detection model, an implementation from *deep-loglizer* [3] is used. Deep-loglizer is a log analysis toolkit for automated anomaly detection based on deep learning models. The main aim of deep-loglizer is to review and evaluate different neural networks applied to log anomaly detection. The methods consists of six state-of-the-art methods of which four are unsupervised and two are supervised. The four unsupervised methods are *DeepLog* [6], *LogAnomaly* [17], *Logsy* [22], and *Autoencoder* [7], while the supervised methods include *LogRobust* [31] and *CNN* [15].

The model of main interest for this thesis is the Transformer-based model Logsy [22]. The choice of using Chen et al.'s implementation of Logsy in deep-loglizer, instead of using Logsy directly, is partly due to the code simplifications and general adaptability that deep-loglizer provides. By first creating the embedding space as explained in Section 3.2, it can then be converted to the appropriate format to be compatible with the Transformer model from deep-loglizer, allowing it to work as a pretrained embedding matrix input.

Auxiliary Data

Since the main goal is to develop an unsupervised model that can train and run on non-prelabeled data, we use the method proposed by Nedelkoski et al. [22], where all the target training data are labeled as non-anomalous data or "normal". Auxiliary data from other sources and datasets, representing the anomalies, are appended during training. These auxiliary datasets do not have to consist exclusively of ”real” anomalies; they can include perfectly normal log messages from sources other than the target data. From the perspective of the target data, these logs will be seen as anomalous, which also helps prevent possible overfitting that can occur if only the target dataset is used. Nedelkoski et al. [22] explains the setup as follows:

PROBLEM DEFINITION [22]. *Let $D = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ be the training logs from the system of interest where $x_i \in \mathbb{R}^d$ is a log message where it words are represented in d-dimensional space (the log message is represented by $d \times |r|$ matrix, where $|r|$ is the number of words) and $y_i \in \{0, 1\}$, $1 < i \leq n$, assuming that the data in the system of interest is mostly composed of normal samples. Let $A = \{(x_n, y_n), \ldots, (x_{n+m}, y_{n+m})\}$, where m is the size of the auxiliary data and $y_i = 1$, $n < i \leq n+m$. Let $\phi(x_i, y_i, \theta) : \mathbb{R}^d \to \mathbb{R}^p$ be a function represented by a neural network, which maps the input log message embeddings to vector representations in $\mathbb{R}^p$, and $l : \mathbb{R}^p \to [0, a]$, $a \in \mathbb{R}$, be a function, which maps the output to an anomaly score. The task is to learn the parameters θ from the training data, and then for each incoming instance in the prediction phase $D_t = \{(x_{t1}), (x_{t2}), \ldots, (x_{tj}), \ldots\}$, t indicates the test sample, predict whether it is an anomaly or normal based on the anomaly scores obtained by $l(\phi(x_i, y_i, \theta))$.*

We follow this definition by Nedelkoski et al. [22] when defining how we handle auxiliary data. In this context, "words" refer to tokens, as our model treats each token as its own word.

The auxiliary data comprises various streams of system logs from two different systems [24]. The primary purpose of using these logs is to prevent overfitting, making the specific choice of logs less important. These labeled logs are publicly available for use:

– **Thunderbird**: A log dataset originally collected from a Thunderbird supercomputer system.

– **Spirit**: A log dataset originating from the Spirit distributed system, a software infrastructure designed to support large-scale computing tasks across distributed computing environments.

These two datasets are combined into one final auxiliary dataset. An extract of the total auxiliary data is then appended to the target training data. The size of the auxiliary data to be added is determined based on the desired anomaly contamination rate in the training set, see Table 3.1 for specific training anomaly contamination rates. The data is also shuffled to maintain a reasonable distribution of the different labeled logs in the training dataset.

Anomaly Detection

After the auxiliary data has been merged with the target training data and all logs have been preprocessed, as shown in Figure 3.1, the tokenized windows of log messages are sent as input to the Transformer model, explained in Section 2.5. Before entering the encoder, the embeddings for each separate log message are summed into a single embedding vector. For instance, with a batch size of 1024, window size of 10, an embedding dimension of 128, and a log message length of 50, the tensor of size $1024 \times 10 \times 128 \times 50$ is summed over the last dimension to create a new tensor of size $1024 \times 10 \times 128$. The ['CLS'] token is then appended at the front of each window, changing the tensor dimensions to $1024 \times 11 \times 128$. This input, in batches, is fed into the encoder part of the Transformer model (Figure 2.6), where multi-head self-attention is applied, followed by concatenation of the output from each head and the first layer normalization. The data then passes through a feed-forward neural network with two layers, followed by another layer normalization. Finally, to obtain the true model output, the pre-padded ['CLS'] token from each instance is extracted as a representation of each window of logs.

Next, the loss is calculated using the hyperspherical loss function (Equation 2.11), as described in Section 2.7. The loss is determined by taking the squared norm of all the batch outputs separately and using it as input to the hyperspherical loss function. The model then trains its parameters by minimizing the loss function through backpropagation. This procedure is repeated for all batches in the training data before evaluating the model on the validation data.

The model by Chen et al. [3], used as the foundation for our model, did not implement or use the positional encoding algorithm explained in Section 2.5. A decision was made to either implement and integrate it ourselves or follow Chen et al. and omit it. Ultimately, positional encoding was not included for several reasons. Because all tokens and embeddings of each log message are summed, applying positional encoding between each token is not sensible. The positional encoding, as defined by Equations 2.2 and 2.3, is independent of the contents of a log message and only depends on the total length and position of each individual token. This implies that with the same number of tokens in every log message, the positional encoding vector added to the original embeddings will be the same for each log message regardless of its contents. Summing all embeddings for each token together is equivalent to repeatedly adding the same number to each embedding index. Thus, the positional encoding function does not provide any benefit in our summing process, rendering it unnecessary.

It could be argued that positional encoding could be applied between log messages in a window, similar to how multi-head self-attention is performed. The position of a log message within a window could theoretically provide context, such as if a specific log message is expected to appear only after another specific log message. However, due to the relatively small window sizes used and the volume of logs generated from the datasets, these patterns are not certain to be captured within a single window. This approach was implemented and tested but did not yield any apparent improvement in results, which is why positional encoding was ultimately not used.

For each validation batch, the learned model weights for the current epoch are used as the data passes through the model. After generating the model output, the squared norm of all ['CLS'] tokens is calculated and used to determine the classification threshold ε as explained in Section 2.7. This threshold is then used to classify each window of logs in the validation data as normal or anomalous. By comparing the predicted labels for every instance in the data with the real labels, performance metrics can be calculated. If the F1-score, explained in Section 3.4, for an epoch is greater than the previously best acquired F1-score (or if it is the first epoch), the model weights and parameters, including the threshold, are saved as the current best model. This process continues until the maximum number of epochs is reached or until a certain number of epochs have passed without updating the current best model. The number of epochs that have to pass without any improvement in performance is known as the *patience*.

When the training phase is finished and the best model is identified, the model's performance is tested on the test data. This testing is done in the exact same way as for the validation data, except that it is only done once.

Model Parameters

The values of the model hyperparameters and other parameters used during training and inference are shown in Table 3.1.

Table 3.1 Table of values of the model parameters.

Parameter	Value
Training/Validation/Test split	80/10/10
Embedding dimension	128
# of layers in the Feed-forward NN	2
Hidden layer size	128
# of attention heads	2
Sliding window size	10
Window stride	1
Max. # of tokens per log message	50
Min. # of tokens per log message	1
Max. # of training epochs	100
Training batch size	1024
Validation/Test batch size	4096
Learning rate	0.001
Training patience	30
Max. # of API-contributed synonyms per OOV-word	10
Auxiliary data contamination rate	3%
Threshold SD factor f (supervised HDFS)	0.8
Threshold SD factor f (unsupervised HDFS)	0.6
Threshold SD factor f (supervised BGL)	-0.2
Threshold SD factor f (unsupervised BGL)	-0.05
Threshold SD factor f (Advenica)	-0.6

3.4 Evaluation

Performance Metrics

To test the model's performance, it was first evaluated using the same datasets and sizes as those used by Murphy and Larsson, specifically the HDFS and BGL datasets. This approach allows for a direct comparison to the previous part of the larger pipeline. The testing was conducted both with and without concept drift to ensure that the ability to handle concept drift, as shown in their work, was preserved. To ensure a fair comparison, the same word exchanges and methods for simulating concept drift in the log data from the training data to the test set were used.

Secondly, the model was trained and evaluated on the datasets provided by Advenica. These datasets include both raw "normal" data for training and emulations of intrusion to test the model's performance.

The evaluation is primarily assessed and presented through three standard metrics: *Precision*, *Recall*, and *F1-score*. The definitions of these three metrics are shown in Equations 3.1, 3.2 and 3.3 respectively.

$$\text{Precision} = \frac{\text{True anomalies reported}}{\text{True + False anomalies reported}} = \frac{TP}{TP+FP} \tag{3.1}$$

$$\text{Recall} = \frac{\text{True anomalies reported}}{\text{Total amount of true anomalies}} = \frac{TP}{TP+FN} \tag{3.2}$$

$$\text{F1} = \frac{2 * \text{Precision} * \text{Recall}}{\text{Precision} + \text{Recall}} = \frac{2 * TP}{2 * TP + FP + FN} \tag{3.3}$$

Here, the abbreviations *TP*, *FP*, *TN* and *FN* stands for true positives, false positives, true negatives and false negatives respectively and can be presented in a *confusion matrix* as seen in Table 3.2.

Table 3.2 Example of a confusion matrix.

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

In the context of this thesis, a sample is a:

- **True positive** if it is *accurately* classified as an *anomaly*.
- **False positive** if it is *inaccurately* classified as an *anomaly*.
- **True negative** if it is *accurately* classified as *normal*.
- **False negative** if it is *inaccurately* classified as *normal*.

Advenica Data

The data from Advenica consists of multiple log files, some recorded during normal operations and others during periods of anomalous activity. All logs recorded during anomalous activity are considered anomalous, even though some of the logs may be identical to non-anomalous logs.

The first step in preprocessing the logs from Advenica is to filter and retrieve only the logs from the service running on the upstream and downstream servers. Initially, the system logs contain all the raw logs generated by the system, including those from several different services, not just the upstream and downstream servers. For example, logs from `NetworkManager` are also included. An example of some raw log entries from Advenica's servers is shown in Figure 3.2.

```
Apr 05 18:14:52 manager-20EM0013MS service-file-up
    ↪ [73943]: 2024-04-05T16:14:52.288320Z  INFO
    ↪ Successfully transferred 6 file(s) with a total
    ↪ of 6262394 byte(s) in the last 60s
Apr 05 18:15:06 manager-20EM0013MS service-file-down
    ↪ [73943]: 2024-04-05T16:15:06.590336Z  INFO
    ↪ Successfully uploaded 6 file(s) with a total of
    ↪ 6494621 byte(s) in the last 60s
Apr 05 18:15:51 manager-20EM0013MS ddengine-odm-orc-
    ↪ downstream-unique-id[73943]: 2024-04-05T16
    ↪ :15:51.047853Z  INFO Last received data from
    ↪ upstream 0s ago
```

Figure 3.2 Example of raw log entries from Advenica.

Next, everything except the application name and the message field is removed. Additionally, the standard tokenization process is applied, removing special characters and numbers. The resulting data sent to the model for evaluation is shown in Figure 3.3.

```
service-file-up INFO Successfully transferred file with
    ↪  a total of byte in the last s
service-file-down INFO Successfully uploaded file with
    ↪ a total of byte in the last s
ddengine-odm-orc-downstream-unique-id INFO Last
    ↪ received data from upstream s ago
```

Figure 3.3 Example of preprocessed log entries from Advenica.

Sessions are then created in a manner similar to how they are created for the BGL dataset, defined by a fixed time length. Each session is then split into windows of a fixed length of log entries. If at least one window is considered anomalous, the entire session is labeled as anomalous.

It is almost certain that not all time sessions in the anomalous dataset contain anomalies since the logs are completely raw and not preprocessed. This means that if the model does not detect an anomaly in a session, it cannot be concluded whether it is a false negative or a true negative. Thus, the model cannot be evaluated using standard metrics such as the F1 score. Similarly, if the model finds an anomaly in the anomalous dataset, it is considered a true positive. However, given that it cannot be confirmed whether all time sessions in this dataset actually contain anomalies, this score is also not entirely reliable. However, it is certain that no time session from the normal dataset contains anomalies. Thus, any result indicating an anomaly in such a session is a confirmed false positive.

Given an equal amount of data from both datasets in the evaluation set, a perfect model is expected to have zero false positives and many true positives. A model that cannot distinguish between the two datasets would have a similar number of false positives and true positives. In other words, a "guessing model" would roughly classify 50% of all normal samples as anomalous and vice versa. These two numbers can therefore be compared to assess the evaluation, despite the limitations on the trustworthiness of the scores discussed above.

4

Results & Analysis

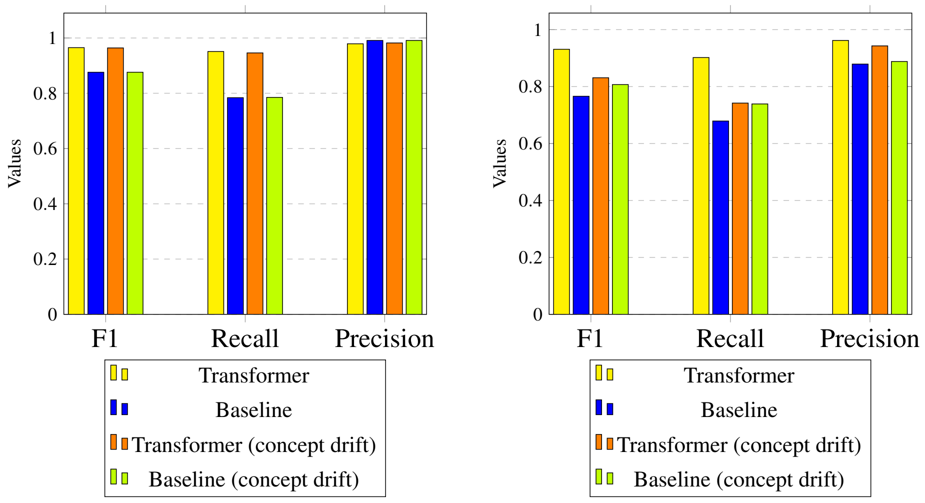
Below in Tables 4.1 and 4.2 are the performance metrics presented for the two datasets used for evaluating the model, HDFS and BGL, respectively. The results are also presented in bar charts in Figures 4.1 and 4.2. The metrics used are as discussed in Section 3.4 *F1 score*: *recall*, and *precision*, all presented with three decimal digits. For the two open-source datasets, HDFS and BGL, the presented results include:

- **Transformer supervised**: This refers to the Transformer model trained with labeled data, where the training data was labeled as normal or anomalous during the training phase.
- **Transformer unsupervised**: This refers to the Transformer model trained with only normal samples from the target dataset and some anomalous auxiliary data representing the anomalies during the training phase. This auxiliary data is from other external open-source datasets. It is considered an unsupervised approach in the sense that it does not use labels from the target system, but only from external sources.
- **Baseline supervised**: These are the supervised results from Murphy and Larsson [21], which our model aims to improve upon. The presented scores are all from their supervised *Logistic Regression* model, their only evaluated supervised model.
- **Baseline unsupervised**: These are results from one of Murphy and Larsson's unsupervised models (*Isolation Forest*), which is the unsupervised model with the best performance metrics in their evaluation.

In addition, all of the above are also evaluated with concept drift in the test data in the same fashion as Murphy and Larsson does. The highest scores for each metric, setting, and dataset are highlighted in **bold** in each respective table.

Table 4.1 Performance Metrics for the HDFS dataset.

Setting	***F1 Score***	***Recall***	***Precision***
Transformer supervised	**0.965**	**0.951**	0.979
Baseline supervised	0.876	0.784	**0.991**
Transformer supervised with concept drift	**0.964**	**0.946**	0.982
Baseline supervised with concept drift	0.876	0.785	**0.991**
Transformer unsupervised	**0.931**	**0.902**	**0.962**
Baseline unsupervised	0.766	0.679	0.879
Transformer unsupervised with concept drift	**0.831**	**0.742**	**0.943**
Baseline unsupervised with concept drift	0.807	0.739	0.888

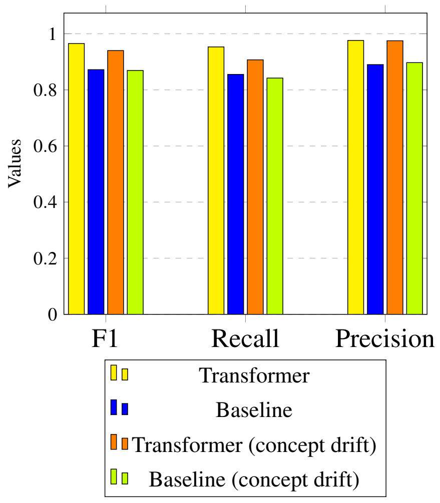


(a) HDFS data results (supervised). **(b)** HDFS data results (unsupervised).

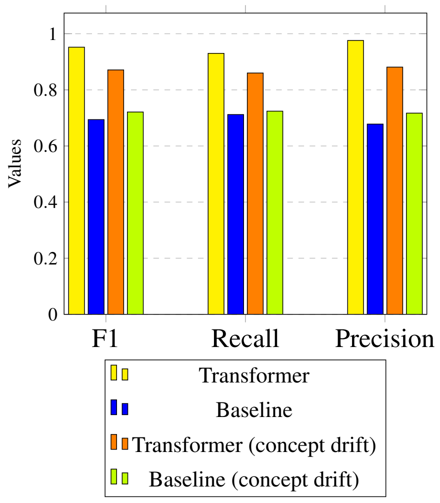
Figure 4.1

Table 4.2 Performance Metrics for the BGL dataset.

Setting	*F1 Score*	*Recall*	*Precision*
Transformer supervised	**0.965**	**0.953**	**0.976**
Baseline supervised	0.872	0.855	0.890
Transformer supervised with concept drift	**0.940**	**0.907**	**0.975**
Baseline supervised with concept drift	0.869	0.842	0.897
Transformer unsupervised	**0.952**	**0.930**	**0.976**
Baseline unsupervised	0.694	0.712	0.678
Transformer unsupervised with concept drift	**0.871**	**0.860**	**0.881**
Baseline unsupervised with concept drift	0.721	0.724	0.717



(a) BGL data results (supervised).



(b) BGL data results (unsupervised).

Figure 4.2

Table 4.3 Confusion matrix for the Advenica dataset.

		Actual	
		Positive	**Negative**
Predicted	**Positive**	TP (14)	FP (1)
	Negative	FN (60)	TN (62)

4.1 HDFS

As seen in Table 4.1 and Figure 4.1, the best overall scores when evaluating the HDFS dataset are obtained from using the supervised Transformer model, outperforming both the unsupervised approach and Murphy and Larsson's models. The supervised model performing better than the unsupervised one is not surprising since the labeled data in the former are "real" anomalies from the HDFS stream of logs, which makes it easier for the model to learn to distinguish between the two classes and recognize anomalous samples during evaluation. The data used for representing anomalies in the unsupervised model are external data samples that may not accurately represent what an anomaly would look like in an incoming stream of HDFS log messages.

The more interesting and practically applicable approach and result are those of the unsupervised model. As discussed earlier, it would be infeasible in a real-life setting to expect anyone to label a large target dataset in advance to train a model in a supervised manner. A quick comparison between the unsupervised Transformer results and Murphy and Larsson's unsupervised Isolation Forest results shows a significant improvement in F1 scores and recall, with similar precision. This pattern is observed across both the unsupervised and the supervised settings.

However, in the unsupervised setting with introduced concept drift, the scores are still better overall than Murphy and Larsson's results, but not by as significant an amount as they were without concept drift. This could be due to the introduced words from the concept drift being too similar to some of the data from the auxiliary data, which is labeled as anomalous during the training phase. This similarity could lead the model to incorrectly classify some of these modified log messages, which are still supposed to be interpreted as normal, as anomalies.

4.2 BGL

In a similar fashion, the best results when evaluating the BGL dataset, as observed from Table 4.2 and Figure 4.2, also come from using the model with supervised training data, for the same reasons as with the HDFS dataset. The Transformer model scores significantly better than the baseline for most settings, even in the unsupervised case when concept drift is introduced, where its scores are significantly better than the baseline, in contrast to the results from the HDFS dataset.

4.3 Advenica

The results from the Advenica dataset need to be interpreted differently compared to the two other datasets due to the validation and test data not being properly labeled. The entire original dataset containing anomalies is labeled as anomalous, while it actually only contains a minority of actual anomalies. This will inevitably lead to the model mislabeling a lot of the normal samples in that set as anomalies, resulting in poor performance metrics. Our goal here, however, is to instead try to achieve a low number of false positives since we do not want to predict any of the labeled normal instances as anomalies. Similarly, we also want many true positives, as this indicates that we have successfully identified many of the real anomalies.

The more relevant and interesting analysis can be done by examining the confusion matrix in Table 4.3 for the test. Here we notice that we are able to keep the false positive number down (1), while still achieving a relatively high true positive score (14). This indicates that out of all the sessions we have labeled as anomalous, 14 out of the 15 in total are from the anomalous labeled part of the data. On the contrary, we have also missed about 50% of the anomalies when looking at the true and false negatives. However, this dataset continues to be difficult to evaluate since the data labeled as anomalies during evaluation are not actually all anomalous. As will be discussed in Section 5.5, it is very possible and easy to impact these scores by only tweaking the threshold parameter slightly, which will make the model label more or fewer samples as anomalies or normal samples depending on the user's preferences.

5

Discussion

5.1 Limitations

One major limitation of using the same parsing and embedding space approach as Murphy and Larsson [21] is that digits and special characters are not considered when tokenizing the logs. The tokenized logs contain less information than the original logs. For example, a log entry might contain information about the sizes of data packages that have been extracted, the duration of a process execution, or the delay of a process. These scenarios are typically represented through numbers that for example indicate the magnitude of packages or times, information that is lost by using this tokenization.

The model's ability to detect intrusions is a important aspect of the pipeline's end goal. Although the HDFS and BGL datasets contain various types of anomalies, such as system failures, they do not include any obvious intrusion attempts. Therefore, without proper evaluation on the Advenica dataset or any other dataset containing intrusions, the model's performance on intrusion-related anomalies cannot be accurately determined.

5.2 Concept Drift

Our approach to handle concept drift is the same method used by Murphy and Larsson [21]. The core idea involves applying an LWET transformation to the embedding space of the model, as described in Section 2.4. This transformation allows the model to handle minor variations in the data by treating similar inputs the same, even if words are replaced with their synonyms.

Essentially, the model is trained to accept a broader array of input windows. It does not learn changes that signify concept drift. If a specific pattern is considered as normal, it will continue to be considered as such until the model is retrained. However, in a real-world scenario with significant concept drift, initial data points might be considered anomalous if they reappear later.

To evaluate the method, certain words are replaced between the training and evaluation datasets. This provides a limited test of concept drift but does not encompass the full range of potential concept drifts and variations that might occur over time. Our evaluation approach captures only a subset of potential concept drift scenarios.

5.3 Model Size

Another aspect that could impact the model's performance is its scale. This includes factors such as window sizes, the amount of log data, and the number of unique templates in the data. For example, the HDFS and BGL datasets only contain 31 and 340 unique templates, respectively. In an online setting with larger outgoing streams of logs from more extensive systems, the number of unique log messages and templates is expected to be significantly larger, requiring the model to handle a broader range of different and unique data.

This also involves using larger datasets and larger dimensions for embeddings. One reason for not employing a larger scale in this work is that other projects that have created similar Transformer-based models for anomaly detection, such as Logsy [22] and Deep-Loglizer [3], have used models of similar sizes to ours. The relatively small sizes of model parameters and dimensions allow the model to train faster and make it more efficient to work with and tweak minor aspects of the model.

Another approach to allow for a larger model size, thereby enabling the model to become more generalized by training on more data, is similar to the method proposed by Bogatinovski et al. [2]. Initially, the model could be trained on several large log datasets to generalize and learn the general semantics of log messages. This pre-training phase would introduce the model to a significant variety of log messages from different systems, making it understand and recognize common structures and patterns in log data.

Then, the model could be fine-tuned on the target dataset. Fine-tuning allows the model to learn the domain-specific semantics and structures of the target system. By training on a larger selection of log data followed by domain-specific fine-tuning, the model can overcome the limitation of having too small of a target dataset to train solely on. This ensures that the model to train on significantly more data while still being specialized on the target data's semantics.

5.4 Evaluation

To evaluate a model, we need to determine if its predictions are correct, which requires labeled data. This presents two main challenges: manually labeling data is time-consuming, and even with dedication, it might be difficult to achieve. Ideally, the model should detect anomalies and intrusion attempts that a human might miss, making manual labeling of entire datasets an unrealistic process for this task.

There are ways to improve the accuracy of some metrics. If certain patterns can be manually detected, these can be searched among the classified patterns to identify true positives. For example, identifying a specific log template that only appears during an intrusion attempt can help verify some anomalies.

Using traditional metrics to evaluate the model can be complicated. Even if the approximate time windows of an intrusion attempt are known, it is hard to determine how many windows are affected or how many contain general logs, even during an attack. In other words, the dataset's anomaly contamination rate is unknown.

However, generating a lot of normal data is feasible, allowing for the optimization of a model to minimize false positives. The challenge lies in optimizing and evaluating the number of false negatives. Alternative metrics could be used; instead of focusing on correctly classifying specific windows, an automated attack could be initiated, and the model's ability to quickly detect the ongoing attack could be measured.

To evaluate the model on unlabeled datasets, such as the Advenica dataset, a different approach is required compared to labeled data evaluation. One idea is to use statistical assumptions to develop a relatively effective evaluation method. By extracting a random sample of, for example, 1000 log entries from the target dataset, this significantly smaller proportion of logs could be labeled to determine the anomaly contamination rate in the sample. It can then be assumed that the same contamination rate applies to the entire dataset. By evaluating the model by counting the ratio of instances labeled as anomalous and normal, the results can be compared to the assumed contamination rate. A relatively similar ratio could indicate a well-functioning model.

Other methods that utilize statistical assumptions to evaluate unlabeled data have been proposed in different papers. For example, Meng et al. [16] proposes LogClass, which utilizes *partial labeling*. This method uses unlabeled data of both anomalous and normal nature, as well as a restricted selection of labeled anomalies. By ensuring that a well-functioning model labels all the known anomalies correctly, statistical assumptions and approximations can be used to estimate the probability of a sample being an anomaly given the data.

5.5 Practical Applications

During the evaluation of the model under various combinations of settings and datasets to achieve the best possible F1 scores, it was discovered that the user setting the parameters can significantly influence the metrics and characteristics prioritized. The most sensitive parameter affecting whether higher recall or precision is prioritized is the threshold parameter during classification. Increasing the factor of standard deviation added to the mean results in the model favoring a higher precision score at the cost of a lower recall score, and vice versa.

Intuitively, this is logical given the nature of the hyperspherical loss function and its operation. In a well-functioning model, anomalies are placed further from the origin with a larger density, while normal samples are placed closer to the origin. A larger threshold would therefore include more normal samples, resulting in fewer normal samples being incorrectly labeled as anomalies. Precision increases as false positives decrease. Conversely, a lower threshold results in more false negatives but captures more true positives, leading to higher recall.

While the HDFS and BGL datasets are effective for evaluating model performance and general capabilities, they do not fully emulate real-life scenarios where data labels are absent. Simplified, the model is designed to label incoming log messages and issue warnings if it detects anomalies. However, the Advenica data better represents how logs would appear in a real system with raw and untouched log streams. In this case, there is no quick way to evaluate the model's performance without manually checking the logs reported as anomalous. Even by checking these logs after being flagged by the model, some anomalous logs may still pass undetected, likely being missed altogether.

Addressing this issue depends on the specific task and user preferences. For example, in a scenario where the goal is to find landmines buried in a field, it is beneficial to report more false positives to capture as many true positives as possible since missing a real one could be devastating. Conversely, in a corporate setting with limited manpower to handle false alarms, it may be preferable to miss some anomalies to reduce operator costs.

6

Future work

While this thesis has explored the capabilities and performance of a Transformer-based machine learning model for log anomaly detection, it still leaves some interesting questions to be explored.

An interesting alternative approach to consider and implement is the tokenization method used by OpenAI [25]. Instead of assigning each word its own token, OpenAI tokenizes parts of words as separate tokens. For example, "tokenization" is decomposed into "token" and "ization". Additionally, they tokenize every character, including numbers and other miscellaneous characters, allowing them to be represented by embedding vectors. In this scenario, the concept of out-of-vocabulary words would not exist, as even unfamiliar words and abbreviations could be represented by multiple separate tokens. This approach would also address one of our model's significant limitations by allowing numbers and special characters to be used, providing additional information and context.

As discussed in Section 2.2, there are additional ways to explore the model's resilience against concept drift. For example, it would be interesting to see if the model could generalize better by training on concept drift or introducing other types of concept drift.

Developing the model into an online-learning model would also be an interesting task. This would allow the model to continuously retrain with new and more relevant incoming log data. This could be a solution for handling concept drift, as the model would retrain itself incrementally in response to non-anomalous changes in the target system's semantics.

7
Conclusion

This thesis aimed to build on and improve the work by Murphy and Larsson [21], who initiated a pipeline between LTH and Advenica with the goal of creating a log anomaly detection machine learning model. By utilizing the Transformer architecture in combination with the preprocessing methods developed byMurphy and Larsson [21], it has been demonstrated that the Transformer model can improve performance compared to earlier models. The model has been evaluated in both a supervised setting with labeled training data and an unsupervised setting where no labeled samples from the target data are included. The great results from the unsupervised setting are particularly important as they best emulate real-world scenarios, where it is impractical to expect manual labeling of log messages of realistic size.

A key takeaway from Murphy and Larsson's work was their handling of concept drift, to which their models showed great resilience. This capability was crucial to include in the development of our model and the continuation of the larger pipeline. By integrating their methods with some improvements, the model still performs well when introduced to concept drift, which indicates that the resilience against concept drift has been maintained.

To evaluate the model's performance and limitations on a more realistic dataset, it was trained and tested on a dataset of log messages provided by Advenica's own servers, which contained no labels. The lack of labels limited the ability to fully evaluate performance using metrics such as F1 scores. However, by studying and analyzing a confusion matrix, it was possible to draw some preliminary conclusions about the model's ability to separate normal from potentially anomalous log messages. This analysis involved considering whether the user would prefer to be alerted by a few too many false positives, thereby catching more real anomalies, or to increase the threshold to reduce false positives, but at the risk of missing some real anomalies.

Bibliography

[1] J. L. Ba, J. R. Kiros, and G. E. Hinton. *Layer normalization*. 2016. arXiv: `1607.06450 [stat.ML]`.

[2] J. Bogatinovski, G. Madjarov, S. Nedelkoski, J. Cardoso, and O. Kao. "Leveraging log instructions in log-based anomaly detection". In: *2022 IEEE International Conference on Services Computing (SCC)*. IEEE. 2022, pp. 321–326.

[3] 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).

[4] D. Borthakur. *Hadoop hdfs architecture guide*. `https://hadoop.apache.org/docs/r1.2.1/hdfs_design.pdf`. Accessed: May 7, 2024. 2008.

[5] DataRobot. *Introduction to loss functions*. Year of publication. URL: `https://www.datarobot.com/blog/introduction-to-loss-functions/`.

[6] 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*. CCS '17. Association for Computing Machinery, Dallas, Texas, USA, 2017, pp. 1285–1298. ISBN: 9781450349468. DOI: `10.1145/3133956.3134015`. URL: `https://doi.org/10.1145/3133956.3134015`.

[7] A. Farzad and T. A. Gulliver. "Unsupervised log message anomaly detection". *ICT Express* **6** (2020), pp. 229–237. DOI: `10.1016/j.icte.2020.06.003`.

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

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

[10] Y. Jia. "Attention mechanism in machine translation". *Journal of Physics: Conference Series* **1314**:1 (2019), p. 012186. DOI: `10.1088/1742-6596/1314/1/012186`. URL: `https://dx.doi.org/10.1088/1742-6596/1314/1/012186`.

[11] K. Kent and M. Souppaya. *Guide to Computer Security Log Management*. Special Publication 800-92. National Institute of Standards and Technology, 2006. URL: `https://csrc.nist.gov/pubs/sp/800/92/final`.

[12] V. I. Levenshtein et al. "Binary codes capable of correcting deletions, insertions, and reversals". In: *Soviet physics doklady*. Vol. 10. 8. Soviet Union. 1966, pp. 707–710.

[13] F. T. Liu, K. Ting, and Z.-H. Zhou. "Isolation forest". In: 2009, pp. 413–422. DOI: `10.1109/ICDM.2008.17`.

[14] J. Liu, Z. Liu, and H. Chen. "Revisit word embeddings with semantic lexicons for modeling lexical contrast". In: *2017 IEEE International Conference on Big Knowledge (ICBK)*. 2017, pp. 72–79. DOI: `10.1109/ICBK.2017.35`.

[15] S. Lu, X. Wei, Y. Li, and L. Wang. "Detecting anomaly in big data system logs using convolutional neural network". In: *2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC/PiCom/DataCom/CyberSciTech)*. 2018, pp. 151–158. DOI: `10.1109/DASC/PiCom/DataCom/CyberSciTec.2018.00037`.

[16] W. Meng, Y. Liu, S. Zhang, F. Zaiter, Y. Zhang, Y. Huang, Z. Yu, Y. Zhang, L. Song, M. Zhang, and D. Pei. "Logclass: anomalous log identification and classification with partial labels". *IEEE Transactions on Network and Service Management* **18**:2 (2021), pp. 1870–1884. DOI: `10.1109/TNSM.2021.3055425`.

[17] W. Meng, Y. Liu, Y. Zhu, S. Zhang, D. Pei, Y. Liu, Y. Chen, R. Zhang, S. Tao, P. Sun, and R. Zhou. "Loganomaly: unsupervised detection of sequential and quantitative anomalies in unstructured logs". In: *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*. International Joint Conferences on Artificial Intelligence Organization, 2019, pp. 4739–4745. DOI: `10.24963/ijcai.2019/658`. URL: `https://doi.org/10.24963/ijcai.2019/658`.

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

[19] T. Mikolov, S. Kombrink, L. Burget, J. Cernocký, and S. Khudanpur. "Statistical language models based on neural networks". *arXiv preprint arXiv:1106.0229* (2011).

[20] G. A. Miller. “Wordnet: a lexical database for english”. *Commun. ACM* **38**:11 (1995), pp. 39–41. ISSN: 0001-0782. DOI: `10.1145/219717.219748`. URL: `https://doi.org/10.1145/219717.219748`.

[21] A. Murphy and D. Larsson. *Towards Automated Log Message Embeddings for Anomaly Detection*. eng. Student Paper. 2024.

[22] 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)*. IEEE. 2020, pp. 1196–1201.

[23] S. Nedelkoski, J. Bogatinovski, A. Acker, J. Cardoso, and O. Kao. “Self-supervised log parsing”. In: *Machine Learning and Knowledge Discovery in Databases: Applied Data Science Track: European Conference, ECML PKDD 2020, Ghent, Belgium, September 14–18, 2020, Proceedings, Part IV*. Springer. 2021, pp. 122–138.

[24] 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)*. 2007, pp. 575–584. DOI: `10.1109/DSN.2007.103`.

[25] OpenAI. *Introduction to openai platform documentation*. `https://platform.openai.com/docs/introduction`. Accessed: 2024-05-14.

[26] J. Ramos et al. “Using tf-idf to determine word relevance in document queries”. In: *Proceedings of the first instructional conference on machine learning*. Vol. 242. 1. Citeseer. 2003, pp. 29–48.

[27] T. D. Science. *Understanding binary cross-entropy / log loss: a visual explanation*. 2018. URL: `https://towardsdatascience.com/understanding-binary-cross-entropy-log-loss-a-visual-explanation-a3ac6025181a`.

[28] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. “Attention is all you need”. *Advances in neural information processing systems* **30** (2017).

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

[30] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan. “Detecting large-scale system problems by mining console logs”. In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. SOSP ’09. Association for Computing Machinery, Big Sky, Montana, USA, 2009, pp. 117–132. ISBN: 9781605587523. DOI: `10.1145/1629575.1629587`. URL: `https://doi.org/10.1145/1629575.1629587`.

[31] X. Zhang, Y. Xu, Q. Lin, B. Qiao, H. Zhang, Y. Dang, C. Xie, X. Yang, Q. Cheng, Z. Li, J. Chen, X. He, R. Yao, J.-G. Lou, M. Chintalapati, S. Furao, and D. Zhang. “Robust log-based anomaly detection on unstable log data”. *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2019). URL: https://api.semanticscholar.org/CorpusID:191140040.

Lund University **Department of Automatic Control** **Box 118** **SE-221 00 Lund Sweden**	*Document name* MASTER'S THESIS *Date of issue* June 2024 *Document Number* TFRT-6232
Author(s) Johan Sundin Linus Särud	*Supervisor* Ola Angelsmark, Advenica, Sweden Fanny Söderlund, Advenica, Sweden Johan Eker, Dept. of Automatic Control, Lund University, Sweden Karl-Erik Årzén, Dept. of Automatic Control, Lund University, Sweden (examiner)

Title and subtitle

AI-driven Log Analysis for Intrusion Detection

Abstract

Today's security systems generate system logs that contain information about important events such as intrusion attempts and hardware failures. However, the large volume of data makes manual analysis impractical. Instead, this thesis proposes a method of using AI for classification.

Building on previous research, a transformer model has been integrated with a hyperspherical loss function and a Large Language Model (LLM). This combination handles the context of new logs and enhances the detection of anomalies. In collaboration with Advenica, the work contributes to the cybersecurity field by integrating a transformer model with a previously proposed embedding approach to create a model with better accuracy than previous approaches.

The model demonstrated an overall improvement in performance on both benchmark datasets (HDFS and BGL) when concept drift was not considered, with F1- scores of 0.931 compared to 0.766 for HDFS, and 0.952 compared to 0.694 for BGL. When concept drift was taken into account, the F1-scores were 0.831 compared to 0.807 for HDFS, and 0.871 compared to 0.721 for BGL.

Keywords
system logs, AI, transformer models, anomaly detection, cybersecurity, AIOps

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title 0280-5316		*ISBN*
Language English	*Number of pages* 1-58	*Recipient's notes*
Security classification		

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