Towards Automated Log Message Embeddings for Anomaly Detection

Adrian Murphy

Daniel Larsson

LUND UNIVERSITY

Department of Automatic Control

MSc Thesis
TFRT-6222
ISSN 0280-5316

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

Printed in Sweden by Tryckeriet i E-huset
Lund 2024

Abstract

Log messages are implemented by developers to record important runtime information about a system. For that reason, system logs can provide insight into the state and health of a system and potentially be used to anticipate and discover errors. Manually inspecting these logs becomes impractical due to the high volume of messages generated by modern systems. Consequently, the research field of machine learning-based log anomaly detection has emerged to automatically identify irregularities. Parsing log messages into a structured, tractable format is a vital step in log anomaly detection. This degree project investigates the application of log message embeddings, a recently proposed log parsing method, for anomaly detection in complex IT systems and measures their resilience to concept drift, where the format of log messages changes over time, in comparison with a traditional parsing approach. Empirical analyses are conducted on two benchmark datasets, revealing that log message embeddings not only achieve anomaly detection results on par with traditional methods but also demonstrate considerable robustness against concept drift. A key focus of this project is on the application of large language models to automate the log embedding pipeline by handling out-of-vocabulary words and extracting synonymous and antonymous word relationships. These capabilities are important for distinguishing log messages that are identical except for one or more synonymous or antonymous word pairs. While large language models show promise in these tasks, experiments highlight the need for further refinement to match the performance achieved through manual operator feedback.

***Keywords*—** IT System Monitoring, Log Anomaly Detection, Large Language Models, Log Message Embeddings, Concept Drift

Acknowledgements

First of all, we would like to express our sincere gratitude to Advenica and their employees for providing us with office space and equipment throughout this degree project. Not least, the daily office breakfast has fuelled much of the thought in this project. We would particularly like to thank Ola Angelsmark and Fanny Söderlund at Advenica for their continuous support, supervision, and input. Moreover, we would like to thank Johan Eker at the Department of Automatic Control, LTH, for his guidance and support. The guidance of Ola, Fanny, and Johan has been invaluable to the direction of this project. Finally, we would like to thank Karl-Erik Årzén, also at the Department of Automatic Control, for his flexibility and swiftness in all matters pertaining to our examination.

Contents

1

Introduction

"Log messages, one of humanity's few infinite resources"

Ola Angelsmark
Advenica

In the modern digital landscape, we have come to rely on systems designed to operate 24 hours a day, serving millions of global users. For instance, vital societal functions, such as hospitals, public transport, and banks, all rely on complex IT systems for smooth operation. Reliability and availability are crucial, as outages can lead to serious revenue losses, or worse, for service providers and cause major disruptions for end users.

Despite rigorous efforts to ensure the stability of these systems, errors inevitably occur. For example, a system may face an attack by an adversary or malfunction due to errors or weaknesses in its implementation. To aid in debugging and troubleshooting, developers use so-called system logs: digital traces of system state and behavior that can be used to track down and address errors. Manually sifting through the entirety of these logs is infeasible due to their volume; for instance, a large-scale modern system might generate log messages at a rate of $\sim$120-200 million lines per hour, corresponding to a file size of about 50 gigabytes [7].

The process of automatically reading system logs and detecting potential anomalies is an active research field. Machine learning-based methods are often employed for anomaly detection, with a pronounced emphasis on unsupervised learning. Expert labeling of anomalous logs is an arduous and costly task, especially considering the scale at which modern systems produce log messages, thereby rendering supervised learning approaches impractical in real-world scenarios. In recent years, neural network approaches – employing technologies like *Autoencoders*, *Long short-term memory* (LSTM) networks or even *Transformers* – have shown promising results, emerging as predominantly favored over more traditional approaches such as *Principal component analysis* (PCA), *Cluster analysis* or *Isolation forests* (iForest) for anomaly detection.

1.1 Thesis Purpose

Anomaly detection in log analysis is typically preceded by the parsing and feature extraction of log messages. Traditionally, parsing methods such as *Drain* [6], that uses a *parsing tree* to extract templates from log messages, have been favored. However, in recent years, with the supposition that the natural language in log messages matters, researchers have taken an interest in models established in natural language processing to analyze the language in system logs. Efforts have been made, for example by Meng et al. [13], to look at how word embeddings can be used for a more robust parsing of log messages that considers the language within them.

To the best of our knowledge, current log embedding models rely on manual operator feedback, particularly in the extraction of synonyms (words with the same or similar meaning) and antonyms (words with opposite meanings) to account for lexical contrast in word embeddings. The purpose of this thesis is twofold: firstly, we want to explore the feasibility, the challenges, and the potential advantages with using embeddings of log messages over traditional parsing methods. In particular, we are interested in the resilience against concept drift, where the format of log messages changes over time. Secondly, we want to explore the opportunities to automate the log embedding pipeline by using large language models. In particular, we are interested in finding in-vocabulary candidates for words that do not exist in the embedding space and in extracting synonymous and antonymous word relationships used to account for lexical contrast.

The findings and the topic of log anomaly detection hold significant importance for *Advenica*, a key stakeholder in this degree project. *Advenica*, a company specializing in cybersecurity, provides a wide variety of software and hardware solutions dedicated to enhancing the information security of nations, authorities and corporations alike. Guided by the signature phrase, “Cybersecurity solutions that protect what matters most”, *Advenica* prioritizes the utmost reliability and security of their products. This degree project marks the first of a comprehensive series focusing on log anomaly detection at *Advenica*. The ultimate objective is to integrate log anomaly detection into new products and enhance existing solutions, aligning with the company’s commitment to advancing the field of cybersecurity.

1.2 Contributions

The scientific contributions of this project emanate from (1) our comparison between *Drain* and a log embedding approach, and (2) our quest to automate the log embedding pipeline using large language models. The main findings from the comparison with *Drain* is that a log embedding approach can achieve results comparable with *Drain* in regular anomaly detection tasks, but excels and outperforms *Drain* when introducing concept drift in log messages. This verifies and supports initial findings by Meng et al. [13] [14]. The main findings from our attempt to automate the log embedding pipeline are that large language models show some potential in extracting synonymous and antonymous word relations and finding in-vocabulary candidates for out-of-vocabulary words. There are, however, limitations and further challenges.

The authors have contributed equally in the execution of the project.

2

Background

2.1 System Logs

Log messages, to put it briefly, are implemented by developers to record run-time information about a system. An example implementation, provided by He et al. [8], could look like the code snippet in Listing 2.1.

```
try {
    renew();
    lastRenewed = Time.monotonicNow();
} catch (IOException ie) {
    LOG.warn("Failed to renew lease for "
        + clientsString() + " for " + (elapsed / 1000)
        + " seconds. Will retry shortly...", ie)
}
```

Listing 2.1 System log implementation example

As explained by He et al. [8], developers and system operators can inspect the produced log messages to monitor system behavior and to perform error analysis. In general, a log message will follow a structure with a timestamp, a verbosity level – such as *INFO, WARNING*, or *ERROR* – indicating the severity of a message, and some free text content. The article presents a real-world example of a log message, taken from a collection of *Hadoop Distributed File System* (*HDFS*) logs:

```
>> 2008-11-09 20:46:55 INFO dfs.DataNode$PacketResponder: Received
   block blk_3587508140051953248 of size 6710
```

Per the general structure of a log message described above, this example can be divided into three separate components as seen in Table 2.1.

Timestamp	Verbosity Level	Free Text Content
2008-11-09 20:46:55	INFO	(...) Received block (...) of size (...)

Table 2.1 Example log message structure

He et al. [8] have made a commendable contribution to the research field of AI-driven log analytics by releasing *Loghub*, a collection of 19 publicly available system log datasets. The datasets are generated from a wide variety of systems, ranging from server applications to supercomputers. In this thesis, we use two datasets from *Loghub* to test and validate our methodology: *HDFS_v1*, log messages generated by a *Hadoop Distributed File System* set-up provided by Xu et al. [22], and *BGL*, log messages generated by the *Blue Gene/L* supercomputer provided by Oliner and Stearley [18].

Both datasets are labeled, in the sense that domain experts have classified the log messages as either anomalous or normal. As mentioned in the introduction, supervised learning for log anomaly detection is impractical in real-world scenarios. In this thesis, however, we use both supervised and unsupervised learning to evaluate the efficiency of log message embeddings for anomaly detection. In the unsupervised approaches, we discard the labels during model training, instead using them as a ground truth that we can compare the model predictions with.

Log Anomalies

Meng et al. [14] categorize log anomalies into two separate classes: sequential and quantitative anomalies. Programs are executed according to a certain flow. In fact, programs may even be represented as control-flow graphs that describe the paths a program can take during execution where log messages are produced as a deterministic sequence of the program's execution path. During normal execution of a program, paths are generally traversed in a typical order. If a pattern of log events deviates from normal program flow, Meng et al. [14] describe it as a sequential anomaly.

Quantitative anomalies, on the other hand, arise from the fact that log events often come in "pairs", or perhaps more generally in "groups". Meng et al. [14] provide an example where a log message "`Interface ae3, changed state to down`" is produced. This log message is most likely often followed by a message "`Interface ae3, changed state to up`". If this quantitative relationship between related logs is violated, Meng et al. [14] describe it as a quantitative anomaly.

HDFS_v1. Per the documentation by Borthakur [2], *HDFS* is, as the name suggests, a distributed file system where data is stored across multiple machines, referred to as nodes. *HDFS* breaks files down into block-sized chunks that are stored independently on the nodes. Each block is stored on multiple nodes for redundancy. The blocks of data are stored in so-called DataNodes, responsible for serving read and write requests, whereas a so-called NameNode manages the namespace of the system and controls access to the files by users. As explained by He et al. [8], the *HDFS_v1* dataset of logs is generated by a 203-node system using a benchmark with a typical workload.

Xu et al. [22] have manually labeled the *HDFS_v1* logs with anomaly descriptions. One

example of an anomaly is that the system may attempt to delete a block that no longer exists on a data node. Other examples include how write operations may fail or how the system may receive blocks that do not belong to any file.

BGL. As described by Oliner and Stearley [18], *Blue Gene/L* was ranked as one of the world's 500 most powerful supercomputers in 2006. The *BGL* logs are records of various events and messages generated by the supercomputer, providing insight into the system's operational status.

Oliner and Stearley [18] divide log anomalies in the *BGL* dataset into three types based on their origin: hardware, software, or indeterminate. For instance, the hardware anomalies, among others, have an alert category `KERNSTOR` with the example message "`data storage interrupt`". An example of a software alert category is `KERNRTSP` with the example message "`rts panic! stopping execution`".

Some important features of the two datasets are summarized in table 2.2 below.

Dataset	Time span	#Lines	#Anomalies	Size
HDFS_v1	38.7 hours	11,175,629	16,838 (blocks)	1.47GB
BGL	214.7 days	4,747,963	348,460	708,76MB

Table 2.2 Summary of the datasets

In summary, what constitutes an anomaly completely depends on the system at hand. There is often a discrepancy between what an anomaly detection model, often trained to recognize normal behavior, considers to be an anomaly and what is truly malignant system behavior. This discrepancy is difficult to bridge,

2.2 Log Anomaly Detection

System log anomaly detection usually comprises three separate steps: parsing, feature extraction, and anomaly detection.

Parsing

As described by He et al. [7], parsing is the process of matching raw log messages to structured *event templates*. If we consider the log message that is printed in Listing 2.1, we see that some parts of the free text component are variable: `clientsString()` and `(elapsed/1000)` will be different from execution to execution. The purpose of log parsing is to extract the constant parts of a log message, corresponding to a so-called *event*. Consider the example provided by He et al. [8] once again.

```
>> 2008-11-09 20:46:55 INFO dfs.DataNode$PacketResponder: Received
   block blk_3587508140051953248 of size 6710
```

Which parts of the log message will vary from execution to execution? Trivially, the timestamp depends on when the message is printed and should not be included in the corresponding event. Moreover, the resource `dfs.DataNode$PacketResponder`, the block identifier `blk_3587508140051953248`, the block size `6710`, and even the verbosity level are likely variable and different depending on execution. If we let `<*>` symbolize the varying parameters, as a type of placeholder, we obtain an *event template*:

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

This event template is the constant part of the log message that does not depend on the internal parameters that the system is considering at the time of printing the message. Additionally, most log parsers provide an opportunity to incorporate expert knowledge about varying parameters; if a system administrator knows the exact format of, for example, a timestamp or a block identifier, they can map these parameters to specific placeholders by using regular expressions. The event template will subsequently be on the following, arguably more informational, form:

```
<TIMESTAMP> <LEVEL> <RESOURCE> Received block <BLOCK_ID> of
size <*>
```

Drain. One popular log parser, that we also use in this thesis for our baseline approach, is *Drain*, suggested by He et al. [6]. *Drain* employs a *fixed-depth tree*, meaning that the depth is provided a priori as a hyperparameter and may not be increased during parsing, to create and match raw log messages to event templates. The method requires minimal previous domain-specific information or assumptions of the system being analyzed; the raw log messages are sufficient. A log message is assigned a template by traversing the fixed-depth tree, where the path is decided by internal rules stored in the nodes until a leaf node is reached. The first layer of nodes, right after the root node, simply divide log messages based on their length. The subsequent layer(s) divide the log messages based on the content in terms of the words, or *tokens*, contained in the messages. The leaf nodes, in turn, contain lists of log groups. A log message is ultimately assigned one of the log groups in a leaf node based on a similarity measure *simSeq*, defined in Equation 2.1.

$$simSeq = \frac{\sum_{i=0}^{n} equ(seq_1(i), seq_2(i))}{n} \tag{2.1}$$

In Equation 2.1, n denotes the length of the log message, seq_1 and seq_2 denote the log message and a log template respectively, $seq(i)$ refers to the ith token of a sequence, and the function *equ* is defined as in Equation 2.2. Simply put, *simSeq* calculates the share of matching tokens between a log message and a log template.

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

The structure of *Drain*'s parse tree is visualized in Figure 2.1.

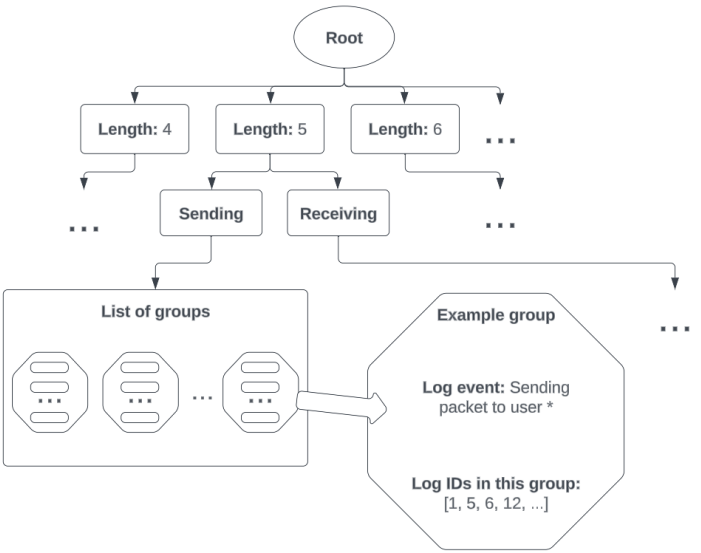

Figure 2.1 Structure of the parse tree in *Drain* with depth 3, showing how a log message is parsed by first considering the length of a message and subsequently by considering individual words. A leaf node contains suitable log event candidates in a list of groups

As is apparent in Figure 2.1, the first layer of nodes, right after the root, in the parse tree decides the path of a log message depending on the length of the message. The second layer of nodes considers the first token of a log message. For example, if we have a log message comprising five tokens, where the first token is "`Sending`", we will reach the leaf node with the log groups shown in Figure 2.1. The asterisk `*` is a so-called wildcard, a special token that matches with anything.

If the similarity score for a log message and the most similar template in the list of log groups does *not* exceed a certain similarity threshold, *Drain* updates the parse tree and creates a new template based on that log message. This process is visualized in Figure 2.2.

In some cases, the initial words of a log message could include varying parameters. This is not an ideal scenario, since *Drain* parses a message from left to right; for example, messages on a form such as "`120 bytes received`" could lead to branch explosion since every leading numerical parameter would lead to a new branch being created. To handle this, tokens containing numerical values are automatically changed into wildcard tokens by *Drain*. Furthermore, operators with domain knowledge can apply an optional masking in a preprocessing step, using regular expressions to replace any varying parameters with a wildcard or a given mask name, akin to what we previously have referred to as a placeholder, such as `NUM` for any numerical tokens.

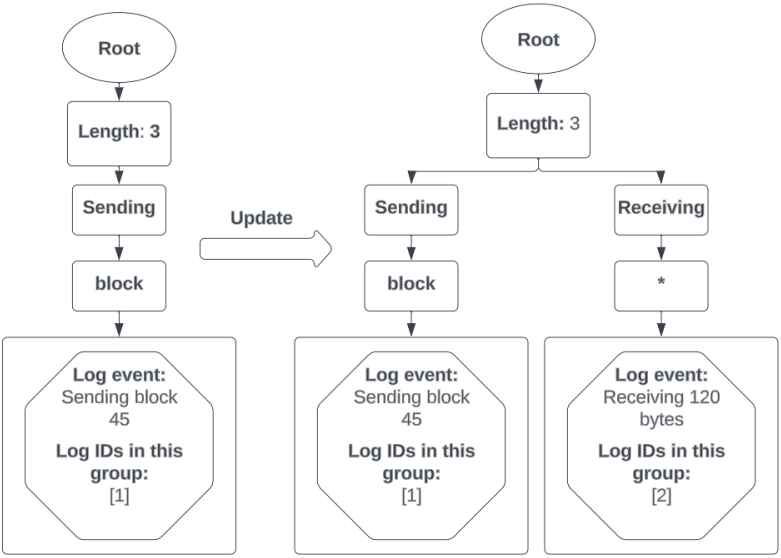

Figure 2.2 Parse tree update example (depth = 4). The log message being added is "`Receiving 120 bytes`"

Feature Extraction

The purpose of feature extraction, in the context of log analysis, is to obtain computable, numerical features from the parsed log messages that can be used in an anomaly detection model. While some types of data have natural numerical interpretations – such as temperature, weight, or even the color values for individual pixels in an image – log messages, containing natural language, do not. Consequently, a different approach is needed. Ideally, the numerical representation of incoming log messages should be able to capture the information present in the logs and how they relate to each other. One common way to represent log messages numerically is through a so-called *event count matrix*, described by He et al. [7].

The Event Count Matrix. The event count matrix consists of rows of so-called *event count vectors*. An event count vector, in turn, counts the occurrences of event templates in a given window. For example, consider a scenario where a log parser, such as *Drain*, has learned the following two event templates during training:

```
(1): <TIMESTAMP> <LEVEL> <RESOURCE> Received block
     <BLOCK_ID> of size <*>
(2): <TIMESTAMP> <LEVEL> <RESOURCE> Added block <BLOCK_ID>
```

Now, if we consider a window of three incoming log messages, where *Drain* maps two logs to the first event template and one log to the second event template, the resulting event count

vector would be $\mathbf{e}_1 = [2,1]$. If we consider yet another window where the first event template does not occur at all and the second event template occurs thrice, the event count vector would be $\mathbf{e}_2 = [0,3]$. $\mathbf{e}_1$ and $\mathbf{e}_2$ constitute the event count matrix $\mathbf{E} = [2,1;\ 0,3]$. If we let k denote the number of event templates learned during training, an event count vector $\mathbf{e}_i$ will generally be on the form $\mathbf{e}_i \in \mathbb{Z}^k_{\geq 0}$. Moreover, if we let w denote the number of *windows* considered, an event count matrix $\mathbf{E}$ will generally be on the form $\mathbf{E} \in \mathbb{Z}^{w \times k}_{\geq 0}$.

Windows are simply groups of log messages collected through some chosen strategy, used for the purpose explained above. He et al. [7] describe three different types of windowing techniques: *fixed windows*, *sliding windows*, and *session windows*. A visual example of how the windowing techniques work can be found in Figure 2.3.

Fixed Windows. Fixed windows are, in some sense, the most basic technique out of the three. This technique relies on a fixed time interval, a "window size", Δt that divides the log messages into time windows.

Sliding Windows. Sliding windows consider two parameters: window size Δt, that is also used in a fixed window approach, and step size. The step size is generally smaller than the window size. For example, we might define a window with a size of one hour and a step of five minutes, yielding an hour long window that slides or shifts forward every five minutes. Since the step size is generally smaller than the window size, sliding windows may overlap.

Session Windows. Session windows rely on identifiers in the log messages. Identifiers, which are only present in some types of log messages, are used to mark different execution paths. An example would be the *HDFS_v1* dataset, where each log message is associated with a block identifier. Session windows, i.e. windows containing log messages with a certain identifier, might be able to better represent the internal state of a system, given that the windowing technique is applicable.

Anomaly Detection

The anomaly detection step is aimed at, given the numerical representations from the feature extraction step, detecting anomalous windows of log messages. In this thesis, we evaluate three different models that can be used for anomaly detection: *Isolation forests*, *Autoencoders*, and *Logistic Regression*.

Isolation Forests. An Isolation forest (iForest) is an unsupervised anomaly detection model proposed by Liu et al. [10]. Under the assumption that anomalies are "few and different", in other words that they are the minority in a dataset and that they have attributes that differ from normal data, an iForest tries to isolate, or separate, anomalous points through an ensemble of so-called *Isolation trees* (iTrees).

An iTree is a data structure comprising several nodes T. Each node T is either (1) an external node without children or (2) an internal node with exactly two children T_l and T_r with a "test" that stores an attribute q and a split value p such that $q < p$ decides the traversal to either one of the internal node's children. An iTree is constructed by recursively partitioning a sample of data X by repeatedly picking a q and p, such that $\min(q) \leq p \leq \max(q)$, uniformly at random. As explained by Liu et al. [10], the recursive partitioning of X continues until either (1) the tree reaches a height limit, (2) when the cardinality of a set equals one ($\#X = 1$), or (3)

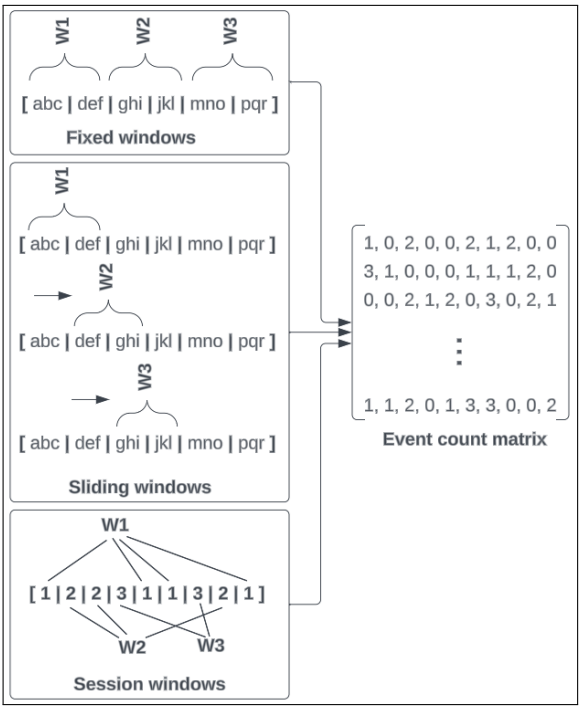

Figure 2.3 Visual examples of windowing techniques. W1, W2, and W3 refer to the created log windows. In this example, the window size for the fixed windows and sliding windows is 2, and the step size for sliding windows is 1. The numbers in the demonstration of session windows correspond to session IDs

all the elements in X are equivalent. To clarify, the cardinality, here and henceforth denoted as #, refers to the number of elements, or items, within a set. In a grown iTree, an anomalous point x will likely have a shorter path length $h(x)$, such that the corresponding external node is reached faster from the root, given the assumption that anomalies are "few and different". An example of isolating a 2-dimensional normal point x_i and an anomalous point x_o in an iTree, is shown in Figure 2.4; the split value affects the position of the dividers, while the choice of the two possible attributes affects the orientation of the dividers, either vertical or horizontal.

As illustrated in Figure 2.4, the path length to x_o's external node is only 2, while the path length to x_i's external node is greater than 10 in the example iTree. Intuitively, we can see that anomalous points are more likely to be isolated in fewer steps.

Ultimately, an iForest, comprising a collection of iTrees, wants to calculate an anomaly score $s(x,n)$ for each data point x in the test dataset, where a point is considered anomalous if its anomaly score exceeds a certain threshold. Liu et al. [10] accomplish this by normalizing the path length $h(x)$ with an estimation of the average path length of an iTree, $c(n)$. The structure

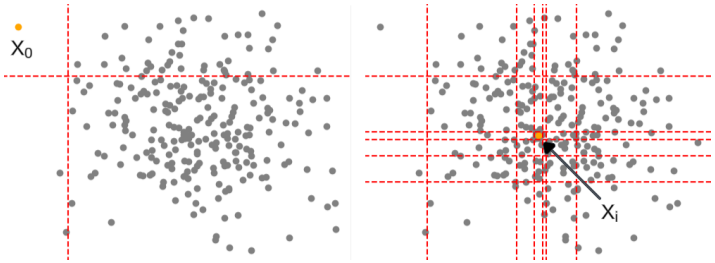

Figure 2.4 Example isolation of an anomalous point x_o (left) and a normal point x_i (right) in an iTree

of an iTree, with at most two children in each node, is equivalent to that of a binary search tree (BST). Moreover, the path length $h(x)$ to reach an external node is, on average, equivalent to the average path length of an unsuccessful search in a BST, given by Equation 2.3.

$$c(n) = 2H(n-1) - \frac{2(n-1)}{n} \tag{2.3}$$

In Equation 2.3, n denotes the size of the testing data and H is the harmonic number, estimated as $H(i) = \ln(i) + 0.5772156649$. Thus, an anomaly score s for a point x can be calculated according to Equation 2.4.

$$s(x,n) = 2^{-\frac{\mathbb{E}(h(x))}{c(n)}},\ 0 \leq s(x,n) \leq 1 \tag{2.4}$$

In Equation 2.4, $\mathbb{E}(h(x))$ denotes the average path length to reach the external node that isolates x in a collection, or ensemble, of iTrees.

Autoencoders. Autoencoders, as explained by Farzad and Gulliver [4], are feed-forward neural networks with an *encoder-decoder* structure. An autoencoder aims to learn a compact representation of an input, meaning that the input is transformed to one or several hidden layers with smaller dimension, referred to as the *encoder*. The *decoder* is subsequently tasked with reconstructing the input from the compact hidden representation. Thus, the training consists of minimizing a loss function that ensures that the output is close to the input, i.e., that the reconstruction error is low. This means that they must be trained of data not contaminated by anomalies. Autoencoders are as such often called semi-supervised, since the anomalies have to be filtered out but otherwise do not require any labels during training. The architecture of an example autoencoder with an input layer, one hidden layer, and an output layer is presented in Figure 2.5.

Farzad and Gulliver [4] explain the encoder output as $y = a(Wx+b)$, where a is the *activation function*, W is the *encoder weight matrix*, x is the input, and b a *bias vector*. The activation function $a(x)$ can, for example, be the *ReLU* function $\max(0,x)$ or the *sigmoid* function $\frac{1}{1+e^{-x}}$. The decoder output can be explained as $z = a(\hat{W}y+\hat{b})$, where $\hat{W}$ is the decoder weight matrix.

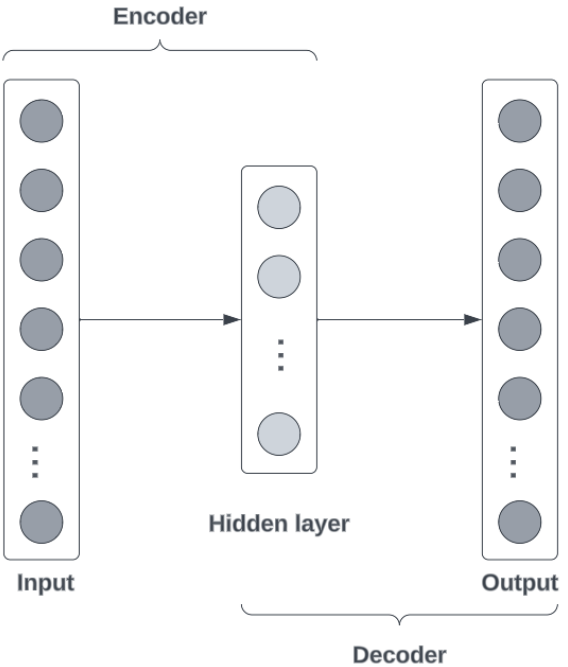

Figure 2.5 Example architecture of an autoencoder

In log anomaly detection, such as in the work by Chen et al. [3], autoencoders can conceptually be seen as models that learn the distribution of normal log event sequences. If an anomalous sequence of logs is passed through the network, in other words an event count vector that should deviate from the distribution learned by the autoencoder, the reconstruction error should be high. If the contamination rate of a dataset is 3%, meaning that 3% of the entries are anomalous, any event count vector with a reconstruction error in the 97th percentile may be classified as an anomaly.

Logistic Regression. Logistic regression, as explained by Agresti [1], is a statistical method to estimate the probability of a certain event, for example an anomaly in a sequence of log messages, occurring. We say that we have a so-called dependent variable Y with two discrete outcomes. In our case, the two discrete outcomes correspond to either a normal (0) or anomalous (1) sequence of logs. Y is said to follow a Binomial distribution with one attempt, $Y_i \sim \text{Bin}(1, p_i)$, where Y_i may denote the outcome for one specific window of logs. From the properties of a Binomial distribution, $\mathbb{P}(Y_i = k) = p_i^k(1-p_i)^{1-k}$, where $k = 0, 1$.

Logistic regression differs from Isolation forests and autoencoders in the sense that it is a type of supervised learning; labels are used during training. The goal is to find appropriate coefficients β during a training phase such that some explanatory variables $\mathbf{x}$, in our case the event count vectors, predict Y well. The model uses the so-called *logit function*, defined in Equation 2.5.

$$\text{logit} = \log \frac{\mathbb{P}(Y_i = 1)}{\mathbb{P}(Y_i = 0)} = \mathbf{x}_i \beta \tag{2.5}$$

The logit function in Equation 2.5 can be rewritten as in Equation 2.6.

$$\frac{\mathbb{P}(Y_i = 1)}{\mathbb{P}(Y_i = 0)} = e^{\mathbf{x}_i \beta} = \frac{\mathbb{P}(Y_i = 1)}{1 - \mathbb{P}(Y_i = 1)} = \frac{p_i}{1 - p_i} \iff p_i = \frac{e^{\mathbf{x}_i \beta}}{1 + e^{\mathbf{x}_i \beta}} \tag{2.6}$$

To estimate β, we want to maximize the likelihood function $\mathscr{L}(\beta; Y)$, which describes the probability of observing the dependent variable, in our case the labels, given the explanatory variables, corresponding to the event count vectors. The likelihood function is defined in Equation 2.7.

$$\begin{aligned} \mathscr{L}(\beta; \mathbf{Y}) &= \prod_{i=1}^{n} p_i^{Y_i} (1 - p_i)^{1 - Y_i} = \prod_{i=1}^{n} \left(\frac{e^{\mathbf{x}_i \beta}}{1 + e^{\mathbf{x}_i \beta}} \right)^{Y_i} \left(1 - \frac{e^{\mathbf{x}_i \beta}}{1 + e^{\mathbf{x}_i \beta}} \right)^{1 - Y_i} \\ &= \ldots = \prod_{i=1}^{n} \frac{e^{Y_i \mathbf{x}_i \beta}}{1 + e^{\mathbf{x}_i \beta}} \end{aligned} \tag{2.7}$$

Maximizing the likelihood function in Equation 2.7 is equivalent to maximizing its more pleasant logarithm, the log-likelihood function $l(\beta; \mathbf{Y})$, defined in Equation 2.8.

$$l(\beta; \mathbf{Y}) = \ln(\mathscr{L}(\beta; \mathbf{Y})) = \sum_{i=1}^{n} \left(Y_i \mathbf{x}_i \beta - \ln(1 + e^{\mathbf{x}_i \beta}) \right) \tag{2.8}$$

Using the log-likelihood function in Equation 2.8, β can be estimated by using some numerical optimization technique to maximize $l(\beta; \mathbf{Y})$.

After estimating β using a training set of labels and event count vectors, the logistic regression model can output a probability score for new observations, indicating the likelihood of an event count vector being a normal or anomalous window of logs. To make a final prediction, i.e., normal or anomalous, a threshold or cutoff value is used to classify an observation.

Evaluation. *Precision*, *recall*, and F_1 *score* are three common measures to evaluate the performance of a log anomaly detection model. As described by He et al. [7], precision reports the percentage of correctly classified anomalies, whereas recall reports the percentage of detected true anomalies. For example, consider a scenario where a dataset contains 100 true anomalies. If a model successfully detects 80 out of the 100 true anomalies, that would equate to a recall of 80%. However, if the model reports 80 false anomalies in addition to the 80 true detected anomalies, that would equate to a precision of 50%. The F_1 score is simply

the harmonic mean of the precision and recall, providing a single measure for both concepts, defined in Equation 2.9 below.

$$\text{precision} = \frac{\text{\#Anomalies detected}}{\text{\#Anomalies reported}}$$

$$\text{recall} = \frac{\text{\#Anomalies detected}}{\text{\#All anomalies}}$$

$$F_1 = \frac{2 \cdot \text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} \tag{2.9}$$

2.3 Embeddings

Embeddings, in the context of natural language processing, are vector representations of words, capturing the features of words as numerical values.

Word2vec

A tried and proven method to create word embeddings is a group of models called *Word2vec*, proposed by Mikolov et al. [15]. The models are shallow, two-layer neural networks aimed at embedding words that occur in a corpus, i.e., a dataset containing natural language, in a continuous vector space. *Word2Vec* relies on the so-called distributional hypothesis, suggesting that words occurring in similar contexts tend to have similar meanings. In their paper, Mikolov et al. [15] propose two primary approaches to create word embeddings: *continuous bag-of-words* (*CBOW*) and *continuous skip-gram*.

Continuous Bag-of-Words. The *CBOW* approach tries to predict a target word, $w(t)$, given its context of surrounding words $w(t+R)$, $R = \pm 1, \pm 2, \ldots, \pm C$, where $2 \cdot C$ is the amount of surrounding words that the model considers. The architecture is summarized in Figure 2.6.

The words w are typically represented as one-hot encoded vectors, where the dimension of each vector is equal to the size of the vocabulary. To clarify, one-hot encoding is a process in which a categorical variable is converted into binary vectors, where *exactly one* value in each vector is "hot" (1) and the rest are "cold" (0). In the input layer, the sum of the one-hot encoded word vectors of the context words is typically used. In other words, the input-projection layer of weights is a fully connected layer with a linear activation function. In the projection-output layer, the target word $w(t)$, also represented as a one-hot encoded vector, is predicted, often using softmax as the activation function. An example could be the sentence "He jumped over the stream", where we want to predict the word $w(t)$ corresponding to "over" given $w(t-2)$ ("he"), $w(t-1)$ ("jumped"), et cetera. If we only have a vocabulary size equal to 5, the one-hot encoded vectors could be $[1,0,0,0,0]$ for "he", $[0,1,0,0,0]$ for "jumped", et cetera.

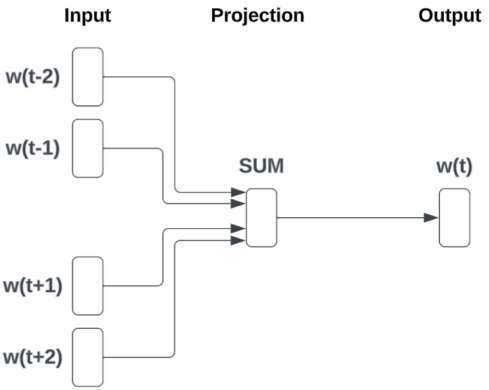

Figure 2.6 Continuous bag-of-words architecture

Continuous Skip-Gram. The *skip-gram* approach tries to predict the context of surrounding words, $w(t+R),\ R=\pm1,\pm2,\ldots,\pm C$, given a particular word $w(t)$. The architecture is summarized in Figure 2.7.

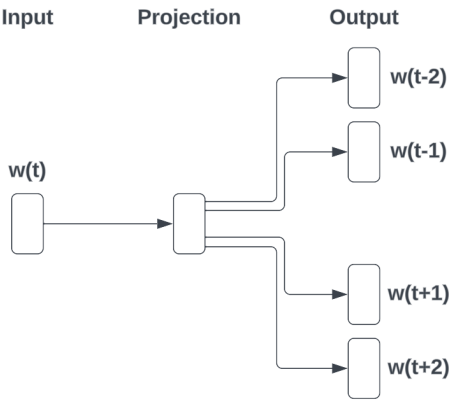

Figure 2.7 Continuous skip-gram architecture

Under the assumption that more distant words, in terms of a larger $|R|$, are less related to $w(t)$, they are given smaller weights in a *skip-gram* approach. A *skip-gram* approach is generally more computationally expensive than a *CBOW* approach, since it handles multiple context words for each target word. A *skip-gram* approach does, however, usually perform better on small datasets and in handling rare/infrequent words. Given the same example as for CBOW, we instead try to predict the surrounding words $w(t-2)$ ("he"), $w(t-1)$ ("jumped"), et cetera, from the one-hot encoding of the word $w(t)$ ("over").

After training a network either in a *CBOW* approach, where the error in predicting a target word is minimized, or *skip-gram* approach, where the error in predicting the context of a

target word is minimized, each word in the vocabulary will have certain associated weights. These weights are used as the word embeddings, since they constitute dense vector representations of how a word relates to the rest of the vocabulary.

As observed by Mikolov et al. [15], *Word2Vec* can capture complex relationships between words. For example, vector operations can represent analogies; the vector representation for "king" minus the embedding for "man" plus "woman" will result in a vector closest to the embedding for "queen".

Cosine Similarity

The *cosine similarity* is a way to calculate how similar two vectors are. Two similar words, such as "hot" and "warm", should ideally have vector representations with a high cosine similarity in an embedding space. The cosine similarity is bounded as $-1 \leq \cos(\theta) \leq 1$, where θ is the angle between the two vectors. Cosine similarity scores closer to 1 would then mean a small angle θ between the vectors (since $\cos(0) = 1$) indicating that vectors, and thus the words themselves, are similar, and vice versa for scores closer to -1. The formal definition for two d-dimensional vectors $\mathbf{a}, \mathbf{b} \in \mathbb{R}^d$ is given in Equation 2.10.

$$\cos(\theta) = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\|\|\mathbf{b}\|} = \frac{\sum_{i=1}^{d} \mathbf{a}_i \mathbf{b}_i}{\sqrt{\sum_{i=1}^{d} \mathbf{a}_i^2} \cdot \sqrt{\sum_{i=1}^{d} \mathbf{b}_i^2}} \tag{2.10}$$

A more in-depth explanation of the cosine similarity can be found in Singhal [21].

K-means Clustering

Per the description provided by MacQueen [12], *k-means* aims to partition observations $(\mathbf{x}_1, \ldots, \mathbf{x}_n)$, $\mathbf{x}_i \in \mathbb{R}^d$, into $k (\leq n)$ sets $\mathbf{S} = \{S_1, \ldots, S_k\}$ where k is chosen as a hyperparameter. For instance, we can use k-means clustering to partition n log message embeddings into k clusters that ideally share similar features. If μ_i denotes the mean, or centroid, of points in S_i, where $\mu_i = \frac{1}{\#S_i} \sum_{\mathbf{x} \in S_i} \mathbf{x}$, the clustering algorithm attempts to minimize the *within-cluster sum of squares* (WCSS), defined in Equation 2.11.

$$\arg\min_{\mathbf{S}} \sum_{i=1}^{k} \sum_{\mathbf{x} \in S_i} \|\mathbf{x} - \mu_i\|^2 = \arg\min_{\mathbf{S}} \sum_{i=1}^{k} \#S_i \cdot \mathrm{Var}[S_i] \tag{2.11}$$

Optimizing Equation 2.11 should be thought of as finding centroids μ such that the variance of points in each cluster S_i, $\mathrm{Var}[S_i]$, is minimized.

The initial centroids μ are often chosen at random. K-means clustering is iterative, in the sense that the algorithm alternates between assigning points to the nearest centroid and recalculating the centroid of each cluster until it converges. Typically, the convergence criteria of k-means clustering is that the centroids no longer move significantly or that points stay in the same clusters.

Log Message Embeddings

Early demonstrations of the benefits of log message embeddings can be attributed to Meng et al. [14]. The authors discuss two primary problems with traditional log anomaly detection, based on learning event templates from logs and matching incoming logs to template indices, that embeddings potentially can mitigate:

1. In a traditional approach, false alarms can be induced if two log events are indexed differently but semantically similar. If they are semantically similar, they may essentially represent the same event but are, nonetheless, treated as two separate events by a model that maps them to two separate event templates
2. Traditional models lack a method to naturally deal with concept drift, a change in the distribution and especially the format of log messages over time, since they cannot match new, previously unseen and distinct, log messages with the templates learned during the training phase

Concept drift, in the context of log anomaly detection, emanates from log messages being edited, for example by simply changing a word or rephrasing the message. Completely new types of log messages can also be added. Concept drift is a natural occurrence, since log messages can be updated at any time during the development of a project. New developers joining a project might also introduce some variation in how logs are written.

The task of actually generating log message embeddings poses two main challenges: (1) handling so-called *out-of-vocabulary* (OOV) words and (2) incorporating knowledge about antonym and synonym pairs, often referred to as lexical contrast.

Out-of-vocabulary Words. OOV words are words that do not occur in the lexica or corpora used to train an embedding model. In *Word2vec*, where words are mapped to their vector representations in a key-value structure, an OOV word will simply result in a key lookup error.

MIMICK, proposed by Pinter et al. [20], is an approach trained over an existing vocabulary to generate embeddings for previously unseen, OOV words. Given a language $\mathscr{L}$, a vocabulary $V \subseteq \mathscr{L}$, and a pre-trained embeddings table $\hat{\mathbf{Q}} \in \mathbb{R}^{\#V \times d}$ where each word $\{w_i\}_{i=1}^{\#V}$ in the vocabulary is assigned a vector representation $q_i \in \mathbb{R}^d$, *MIMICK* is trained to find a function $f : \mathscr{L} \mapsto \mathbb{R}^d$, where the projected function $f|_V$ approximates, or more abstractly mimics, $f(w_i) \approx q_i$. Thus, an OOV word $w_{i^*} \in \mathscr{L} \setminus V$ can be assigned a vector representation $q_{i^*} = f(w_{i^*})$.

MIMICK utilizes a so-called *Word Type Character Bi-LSTM* network, which considers single characters c_i for a given word $w = \{c_i\}$, $1 \leq i \leq n$. Each character c_i is converted into a vector representation, a character embedding, and the Bi-LSTM network, a combination of one forward-LSTM and one backward-LSTM network, processes the character embeddings from first to last and last to first character respectively. The final hidden state of the forward-LSTM network, h_n^f, and the final hidden state of the backward-LSTM network, h_0^b, are combined to form a representation of words based on the contextual information from both directions. The final word embedding is computed using h_n^f and h_0^b by a multilayer perceptron, where the

training task is to minimize the Euclidean distance between the computed word embedding and the true word embedding.

A mis- or alternative spelling of a word can potentially be OOV. In these cases, the *Levenshtein distance* can be used to find the most similar word that exists in the vocabulary. The Levenshtein distance was first proposed by Levenshtein [9] and can be used to measure the difference between two strings. For two strings a and b with length $|a|$ and $|b|$ respectively, the Levenshtein distance is defined as in Equation 2.12.

$$\text{lev}(a,b) = \begin{cases} |a| & \text{if } |b| = 0 \\ |b| & \text{if } |a| = 0 \\ \text{lev}(\text{tail}(a), \text{tail}(b)) & \text{if } \text{head}(a) = \text{head}(b) \\ 1 + \min \begin{cases} \text{lev}(\text{tail}(a), b) \\ \text{lev}(a, \text{tail}(b)) \\ \text{lev}(\text{tail}(a), \text{tail}(b)) \end{cases} & \text{otherwise} \end{cases} \tag{2.12}$$

The tail in Equation 2.12 is defined as all but the first character, whereas the head is only the first character. Intuitively, the Levenshtein distance corresponds to the number of changes (deletions, substitutions, and insertions) that must be made to one string to transform it to the other.

Lexical Contrast in Embeddings. As explained by Liu et al. [11], traditional word embedding models rely on the distributional semantics hypothesis, which assumes that words occurring in the same contexts resemble each other and should be grouped closely together in a vector space. Since both synonyms and antonyms often occur in the same contexts, for example "hot" and "cold" in a sentence such as "the coffee is …", they are ultimately placed closely together in most embedding spaces.

Meng et al. [14] recognize that antonyms with similar vector representations, in other words embeddings with a lack of lexical contrast, are problematic in the matter of log messages. We further demonstrate this by defining 6 hypothetical log event templates:

1. `Sending block <*>`
2. `Receiving block <*>`
3. `Reciving block <*>`
4. `Server <*> in <*> is in state poweredOn`
5. `Server <*> in <*> is in state poweredOff`
6. `Adding an already existing block <*>`

In the context of the hypothetical event templates, "`sending`" and "`receiving`" are antonyms; event template number 1 and 2, in some sense, represent the opposite events and should ideally have dissimilar vector representations. This also applies to template number 4 and 5, with the added complexity that "`poweredOn`" and "`poweredOff`" likely are OOV for

most embedding models. Furthermore, event template number 3 should ideally have a similar vector representation to template number 2, since they are identical aside from a minor spelling mistake.

OpenAI provides an API with text embedding functionality. We call the API to create embeddings for the hypothetical log event templates, using the *text-embedding-ada-002* engine, and calculate the cosine similarities between the vectors, presented in Figure 2.8.

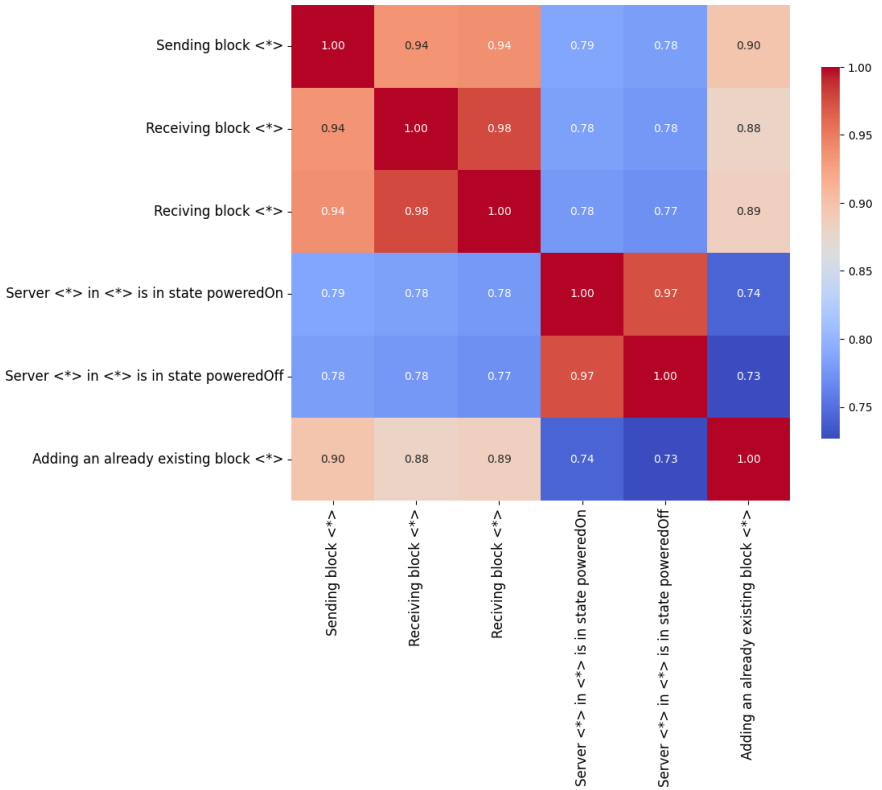

Figure 2.8 Cosine similarities between vector representations of the 6 hypothetical event templates using *OpenAI* embeddings

In Figure 2.8, we observe that "`Sending block <*>`" and "`Receiving block <*>`", as well as "`Server <*> in <*> is in state poweredOn`" and "`Server <*> in <*> is in state poweredOff`" have high cosine similarities, 0.94 and 0.97 respectively, even though they represent opposite events. "`Receiving block <*>`" and "`Reciving block <*>`" also have a high cosine similarity; in this case, however, it is desired.

Evidently, we need a way to transform the embedding space so that templates containing antonyms are easily separable, while still preserving their relation to other templates. For this

purpose, Liu et al. [11] have proposed an approach called *Lexicon-based Word Embedding Tuning* (LWET) to include knowledge about lexical contrast in embeddings. To explain the approach, we first introduce some terminology described in the paper:

- Let $V = \{w_1, w_2, \ldots, w_n\}$ denote the vocabulary, consisting of n words $w_i,\ i \in [1,n]$
- Let $\hat{\mathbf{Q}}$ denote the matrix with the pre-trained embeddings, where each column $\hat{q}_i \in \mathbb{R}^d$ is the d-dimensional word vector of w_i
- Let $\mathbf{Q}$ denote the matrix with the retrofitted embeddings, i.e., the embeddings after applying LWET
- Let $\mathbf{E}_S = \{(i,j) \mid w_i, w_j \in V \text{ and } w_i, w_j \text{ are synonyms}\}$ denote the synonymous relationships in the vocabulary
- Let $\mathbf{E}_A = \{(i,k) \mid w_i, w_k \in V \text{ and } w_i, w_k \text{ are antonyms}\}$ denote the antonymous relationships in the vocabulary
- Let $\mathbf{E}_I = \{(i,l) \mid w_i, w_l \in V \text{ and } w_i, w_l \text{ are semantically unrelated}\}$ denote the irrelevant relationships in the vocabulary
- Let $\mathfrak{D}(\hat{q}_i, \hat{q}_j)$ denote the Euclidean distance between two vectors $\hat{q}_i, \hat{q}_j \in \mathbb{R}^d$, which is defined in Equation 2.14

The retrofitted word vectors in $\mathbf{Q}$ should ideally have the following three properties:

1. q_i and q_j should be spaced closely together if $(i,j) \in \mathbf{E}_S$
2. q_i and q_k should be spaced far apart if $(i,k) \in \mathbf{E}_A$
3. The distance between q_i and q_l, given that $(i,l) \in \mathbf{E}_I$, should be greater than the distance to the synonyms of w_i but shorter than the distance to the antonyms of w_i

Liu et al. [11] express these properties in two mathematical conditions:

1. $\forall i \in [1,n],\ \mathfrak{D}(q_i,q_j) < \mathfrak{D}(q_i,q_l),\ \forall (i,j) \in \mathbf{E}_S,\ \forall (i,l) \in \mathbf{E}_I$
2. $\forall i \in [1,n],\ \mathfrak{D}(q_i,q_k) > \mathfrak{D}(q_i,q_l),\ \forall (i,k) \in \mathbf{E}_A,\ \forall (i,l) \in \mathbf{E}_I$

To fulfil the conditions above, Liu et al. [11] propose a cost function $\Psi(\mathbf{Q})$ that should be minimized:

$$\Psi(\mathbf{Q}) = \sum_{i=0}^{n} \Bigg[\alpha \cdot \mathfrak{D}(q_i, \hat{q}_i) + \beta \cdot \sum_{(i,j) \in \mathbf{E}_S} \frac{\mathfrak{D}(q_i,q_j)}{N_S^i} - \gamma \cdot \sum_{(i,k) \in \mathbf{E}_A} \frac{\mathfrak{D}(q_i,q_k)}{N_A^i} + \delta \cdot \sum_{(i,l) \in \mathbf{E}_I} \frac{\mathfrak{D}(q_i,q_l)}{N_I^i} \Bigg] \tag{2.13}$$

where N_S^i, N_A^i, and N_I^i denote the number of synonyms, antonyms, and irrelevant words for w_i respectively. The hyperparameters α, β, γ, and δ control the relative strength of each component. Based on grid search, Liu et al. [11] propose $\alpha = 1$, $\beta = 2$, $\gamma = 3$, and $\delta = 4$.

The first component in Ψ can be interpreted as a general penalty for increasing the distance between the pre-trained word vector $\hat{q}_i$ and the retrofitted vector q_i. The second and third component of the cost function relate to penalizing increased distances for synonyms and decreased distances for antonyms respectively, in accordance with the first and second desired properties of the retrofitted word vectors. The fourth and final component penalizes increased distances for irrelevant words per the third desired property of $\mathbf{Q}$.

Since most words are unrelated, we have that $n = \#V \approx \#\mathbf{E}_I$. In other words, the number of irrelevant words is nearly equal to the size of the entire vocabulary. Consequently, the computational complexity of Equation 2.13 is $O(n^2)$. To speed up computation, Liu et al. [11] propose two approximation algorithms: *positive sampling* and *quasi-hierarchical softmax*. In this thesis, we will only consider positive sampling.

The idea behind positive sampling is to avoid iterating over all of the irrelevant words by only considering a subsample of $\mathbf{E}_I$. Liu et al. [11] suggest that $m_i = \max(N_A^i, N_S^i)$ samples should be randomly selected from $\mathbf{E}_I$. They denote this as $C(w_i) = \text{sample}(i, m_i) \subseteq \mathbf{E}_I$. Consequently, the time complexity is reduced from $O(n^2)$ to $O(nm)$, where $m = \frac{1}{n}\sum_{i=1}^{n} m_i$. The pseudocode for LWET with positive sampling, which uses gradient descent to optimize Ψ, is provided by Liu et al. [11] and presented in Listing 2.2.

```
Input: Initial word embeddings Q̂, Learning rate step,
Threshold λ, Iterating times t
Output: Word embeddings which have been improved Q
Initialize: α, β, γ, δ, Ψ_old = 0, Ψ_new = 0, Q = Q̂, i = 0

while true do
    i = i + 1
    Ψ_old = Ψ_new
    Ψ_new = 0

    for w_i in V do
        get w_i's synonyms from lexicons, S_{w_i}
        get w_i's antonyms from lexicons, A_{w_i}
        sample m irrelevant words and get C(w_i)
        update Ψ_new according to cost function
        update the first derivative of Ψ(Q) with respect to w_i
        update w_i's vector according to the first derivative
    end for

    if Ψ_new − Ψ_old < λ or i > t then
        break
    end if
```

```
end while

return Q
```

Listing 2.2 LWET with positive sampling pseudocode, provided by Liu et al. [11]

The positive sampling algorithm in Listing 2.2 requires the first derivative of Ψ with respect to w_i. Liu et al. [11] do not provide this equation, and we therefore derive it ourselves. We must first find which terms in the cost function Ψ, defined in Equation 2.13, depend on w_i. It is easy to see that $\mathfrak{D}(q_i,\hat{q}_i)$, $\mathfrak{D}(q_i,q_j)$, $\mathfrak{D}(q_i,q_k)$, and $\mathfrak{D}(q_i,q_l)$ all depend on w_i since they all contain q_i, the word vector corresponding to w_i. The Euclidean distance between two vectors, for instance $\hat{q}_i,\hat{q}_j \in \mathbb{R}^d$, is defined as:

$$\mathfrak{D}(\hat{q}_i,\hat{q}_j) = \|\hat{q}_i - \hat{q}_j\| = \sqrt{(\hat{q}_i^{(1)} - \hat{q}_j^{(1)})^2 + (\hat{q}_i^{(2)} - \hat{q}_j^{(2)})^2 + \ldots + (\hat{q}_i^{(d)} - \hat{q}_j^{(d)})^2} \tag{2.14}$$

If we only consider the first element of the vector $\hat{q}_i$, we have the partial derivative:

$$\frac{\partial \mathfrak{D}}{\partial \hat{q}_i^{(1)}} = \frac{2(\hat{q}_i^{(1)} - \hat{q}_j^{(1)})}{2\sqrt{(\hat{q}_i^{(1)} - \hat{q}_j^{(1)})^2 + (\hat{q}_i^{(2)} - \hat{q}_j^{(2)})^2 + \ldots + (\hat{q}_i^{(d)} - \hat{q}_j^{(d)})^2}} = \frac{\hat{q}_i^{(1)} - \hat{q}_j^{(1)}}{\mathfrak{D}(\hat{q}_i,\hat{q}_j)}$$

We can subsequently determine the gradient of $\mathfrak{D}$ with respect to $\hat{q}_i$:

$$\nabla_{\hat{q}_i}\mathfrak{D} = \begin{bmatrix} \frac{\partial \mathfrak{D}}{\partial \hat{q}_i^{(1)}} \\ \vdots \\ \frac{\partial \mathfrak{D}}{\partial \hat{q}_i^{(d)}} \end{bmatrix} = \begin{bmatrix} \frac{\hat{q}_i^{(1)} - \hat{q}_j^{(1)}}{\mathfrak{D}(\hat{q}_i,\hat{q}_j)} \\ \vdots \\ \frac{\hat{q}_i^{(d)} - \hat{q}_j^{(d)}}{\mathfrak{D}(\hat{q}_i,\hat{q}_j)} \end{bmatrix} = \frac{\hat{q}_i - \hat{q}_j}{\mathfrak{D}(\hat{q}_i,\hat{q}_j)} \tag{2.15}$$

Using Equation 2.15, we can obtain the first derivative of Ψ, defined in Equation 2.13, with respect to w_i as:

$$\begin{aligned} \frac{\partial \Psi}{\partial w_i} = \frac{\partial \Psi}{\partial q_i} &= \alpha \cdot \frac{q_i - \hat{q}_i}{\mathfrak{D}(q_i,\hat{q}_i)} + \beta \cdot \sum_{(i,j)\in \mathbf{E}_S} \frac{q_i - q_j}{N_S^i \cdot \mathfrak{D}(q_i,q_j)} \\ &- \gamma \cdot \sum_{(i,k)\in \mathbf{E}_A} \frac{q_i - q_k}{N_A^i \cdot \mathfrak{D}(q_i,q_k)} + \delta \cdot \sum_{(i,l)\in \mathbf{E}_I} \frac{q_i - q_l}{N_I^i \cdot \mathfrak{D}(q_i,q_l)} \end{aligned} \tag{2.16}$$

Lexical Databases

To apply LWET to our embeddings, we first need to extract the synonymous and antonymous relationships between words. We propose a method that combines *WordNet*, used by Meng

et al. [14] and Meng et al. [13], with a *Large Language Model* (LLM) for domain-specific words.

WordNet. *WordNet* is a lexical database for the English language, containing meanings and relations between more than 118 000 words, proposed by Miller [16]. Synonyms are the fundamental relation, since *WordNet* organizes words into sets of synonyms called *synsets*. A synset, in turn, is linked to other synsets through various semantic relations, such as antonymy.

Large Language Models. *WordNet* contains many, but far from all, synonymous and antonymous word relationships in the English language. We conjecture that modern LLMs are capable of inferring the meaning of domain-specific and OOV words, such as "`poweredOn`" and "`poweredOff`", while simultaneously being able to detect and correct spelling mistakes such as "`reciving`". As a consequence, we propose a novel approach to use LLMs as a lexical database, capable of automatically extracting lexical contrasts for OOV words.

To quickly demonstrate the capabilities of an LLM in extracting lexical contrasts, we prompt *ChatGPT* (GPT-4) as in Figure 2.9 with the vocabulary of the hypothetical event templates.

```
Determine which of these words are synonyms and which are
antonyms:

{sending, block, receiving, reciving, server, is, in, state,
poweredOn, poweredOff, adding, an, already, existing}

Consider spelling mistakes and give your response in
JSON format.
```

Figure 2.9 Example prompt to ChatGPT (GPT-4) to extract lexical contrasts

Using the prompt in Figure 2.9, the model responds as in Figure 2.10.

Open Source LLMs. *LocalAI*, developed by Giacinto [5], allows a user to run large language models using endpoints compatible with the *OpenAI* API specifications in a Docker container locally. *LocalAI* can be used with a wide variety of models. We settle for a fine-tuned version of Meta's *LLaMA* 2, available on Hugging Face. More specifically, we use the *Q5_K_M* method, which is one of the larger versions with very low quality loss.

Prompt Engineering. For an LLM to properly interpret a prompt, and give the user a more satisfactory answer, careful consideration should be made in regard to how the instruction is phrased. Recently, the area of prompt engineering has established itself along with the recent breakthroughs in LLMs.

In the documentation by *OpenAI* [19], different strategies are described for writing better prompts. While the importance of the tactics may vary depending on the task, it is generally true that writing clear and descriptive prompts will give more relevant responses. An example

```
(...) based on the list provided, and taking spelling mistakes
into account, I can categorize the words as follows:

{
    "synonyms": {
        "receiving": ["reciving"],
    },
    "antonyms": {
        "poweredOn": ["poweredOff"],
        "sending": ["receiving"],
        "adding": ["existing"]
    }
}
```

Figure 2.10 Response from ChatGPT (GPT-4) with lexical contrasts

given in the documentation recommends modifying the prompt "How do I add numbers in Excel?" into "How do I add up a row of dollar amounts in Excel? I want to do this automatically for a whole sheet of rows with all the totals ending up on the right in a column called 'Total'...". Additionally, the prompts can include examples of the desired output format, the desired length of the output, and delimiters – for example using quotes "·" – to accentuate important details in the input.

2.4 Similar Approaches

Two approaches that have laid the foundation and explored the topic of of log message embeddings for anomaly detection are *Template2Vec* and *Log2Vec*.

Template2vec

Template2Vec, proposed by Meng et al. [14], is a method to extract the semantic information in log message templates. The method uses *Word2Vec* to create word embeddings from a corpus of log message templates, subsequently using a method called *dLCE* proposed by Nguyen et al. [17] to introduce lexical contrast for antonym and synonym pairs. *Template2Vec* constructs a log embedding as the weighted average of the log message's word embeddings. The method uses *WordNet* to extract synonyms and antonyms, but relies on manual operator feedback for domain-specific synonym and antonym pairs.

Log2vec

In many ways, *Log2Vec* is a continuation of *Template2Vec* proposed by the same authors: Meng et al. [13]. As opposed to *Template2Vec*, *Log2Vec* can create embeddings from raw, unstructured log messages. Meng et al. [13] propose a method called log-specific word embedding (LSWE), that accounts for lexical contrast, to create word embeddings. The synonyms

are extracted from *WordNet* and, again, from manual feedback provided by an operator for domain-specific synonym and antonym pairs. *Log2Vec* uses *MIMICK* to handle OOV words.

3

Method

Our method aims to evaluate the efficacy of using log message embeddings for anomaly detection. To accomplish this, we use two datasets, *HDFS_v1* and *BGL*, to compare an embeddings approach to a baseline in two tasks:

1. Anomaly detection
2. Concept drift resilience

In the anomaly detection task, we want to compare the baseline, that uses *Drain* to learn and match messages to clusters of logs, to an embeddings approach that uses *k-means* clustering to learn clusters of log message embeddings. The event count matrices obtained from these two clustering approaches are ultimately used in three different anomaly detection models.

In the concept drift resilience task, we introduce different phrasing for some of the log messages in the test dataset. We are interested in evaluating how resilient the baseline and the embeddings approach are to these changes, i.e., how the anomaly detection performance is affected when the test dataset is modified with a slightly different wording.

3.1 Baseline

We first implement a baseline for the *HDFS_v1* and *BGL* datasets, with an 80/20 split between training and test data, using *Drain* as a parser. The training and test datasets are split in a so-called class-uniform way, meaning that the share of anomalous windows of logs should be the same in both datasets. We subsequently construct event count matrices for the training and test data, using session windows for *HDFS_v1*, based on block identifiers, and sliding windows for *BGL*, using 6 hour long windows with a 2 hour step size.

We set *Drain*'s tree depth to 4, the similarity threshold to 0.7, the maximum number of children to 100, and the maximum amount of clusters to 1024. For the *HDFS_v1* dataset, we also mask or substitute the datetimes, block identifiers, IP addresses, and resources in the raw log messages with fixed tokens. *Drain* is trained over the training dataset, and subsequently tries to match the raw log messages in the test dataset to the learned clusters. Each cluster has

an index, which is used as an index in the resulting event count vector for a sequence of logs. If *Drain* fails to match a raw test log message to a cluster, we increment an index specifically designated to non-matched logs in the event count vector. We ultimately train three different anomaly detection models: an *Isolation forest*, an *autoencoder*, and *logistic regression*. The hyperparameters for the *Isolation forest* are presented in Table 3.1, and the hyperparameters for the *autoencoder* in table 3.2. The hyperparameters for the *logistic regression* are the standard parameters in scikit-learn.

	BGL	*HDFS_v1*
n_estimators	100	100
max_samples	0.9999	0.9999
n_jobs	4	4
Contamination	46%	3%

Table 3.1 *Isolation Forest* hyperparameters

The high contamination rate for *BGL* arises from the sliding windows. Since sliding windows may overlap, the anomalous logs are present in and contaminate many more windows.

	BGL	*HDFS_v1*
Hidden Activation Function	ReLU	ReLU
Output Activation Function	Sigmoid	Sigmoid
Loss	MSE	MSE
Optimizer	Adam	Adam
Epochs	30	30
Batch Size	32	32
Dropout Rate	0.2	0.2
L2 Regularizer	0.1	0.1
Validation Size	0.1	0.1
Contamination	46%	3%

Table 3.2 *Autoencoder* hyperparameters

For the *HDFS_v1* dataset, we use four hidden layers in the *autoencoder*: 32 nodes $\rightarrow$ 20 nodes $\rightarrow$ 20 nodes $\rightarrow$ 32 nodes. For the *BGL* dataset, which has a much larger input layer, we also use four hidden layers: 275 nodes $\rightarrow$ 150 nodes $\rightarrow$ 150 nodes $\rightarrow$ 275 nodes.

3.2 Embeddings Approach

The embeddings approach contains several components. In essence, we aggregate the embeddings for individual words in a log message to a log message embedding. The log message embeddings are clustered using *k-means*, where the identifier of a cluster corresponds to a dimension in an event count vector. We set k in the clustering algorithm to the same amount of templates that *Drain* finds in the baseline.

Lexicons

We evaluate the capability of an LLM to obtain synonyms and antonyms by a simple point system. By first manually annotating the vocabulary obtained from the *HDFS_v1* logs with suitable synonyms and antonyms, we award one point for every synonym and antonym that the LLM correctly responds with in JSON format. Moreover, we deduct one point for every synonym that the LLM wrongly considers to be an antonym and every antonym that the LLM wrongly considers to be a synonym.

Given the background on prompt engineering, we write three different prompts that clearly state the desired output format, the desired length of the output, and use delimiters to accentuate the words of interest. The three prompts are presented in table 3.3.

Prompt No.	Prompt Content
1	Determine a maximum of 10 synonyms and antonyms each to the word "*w*". Respond in JSON format as {'synonyms': [], 'antonyms': []}.
2	Please provide synonyms and antonyms for the word "*w*". List up to 10 synonyms and 10 antonyms. Format your response in JSON with the structure {'synonyms': ['synonym1', 'synonym2', ...], 'antonyms': ['antonym1', 'antonym2', ...]}. For example, for the word 'fast', the response should look like {'synonyms': ['quick', 'rapid', ...], 'antonyms': ['slow', 'lethargic', ...]}.
3	For the word "*w*", identify synonyms and antonyms. Limit the list to 10 each. Please present your answer in a simple JSON format, like this: {'synonyms': ['synonym1', 'synonym2', ...], 'antonyms': ['antonym1', 'antonym2', ...]}. Ensure clarity and relevance in the selection of words.

Table 3.3 Tested prompts to obtain synonyms and antonyms from an LLM

We test the locally running *LLaMA* model with all three prompts by calling the chat completion API endpoint. For comparison, we also manually prompt *ChatGPT* (GPT-3.5) with the first prompt. Ideally, all prompts should be tried using *ChatGPT*, but the effort of manually prompting over a thousand words, with a maximum of 70 allowed prompts per hour, for two additional prompt formats would simply be too time-consuming.

The Embedding Space

Firstly, we use *Word2vec* to create an embedding space. We consider two corpora, a general corpus $\mathscr{C}$, with a vocabulary V, and a corpus with log messages $\mathscr{C}_{logs}$, with a vocabulary V_{logs}. For our general corpus, we use the so-called *Text8* corpus, a collection of English Wikipedia articles. After training *Word2vec* on $\mathscr{C} \cup \mathscr{C}_{logs}$, we obtain an embeddings table $\hat{\mathbf{Q}} \in \mathbb{R}^{\#(V \cup V_{logs}) \times d}$.

We use 128 dimensions for the word embeddings in *Word2Vec*. Moreover, we set the window

size, in other words the amount of surrounding words that the model considers, to 5, and the minimum count of a word to be included in the vocabulary to 5.

Subsequently, we use *WordNet* to find general synonymous word relationships, $\mathbf{E}_S^g$, and general antonymous word relationships, $\mathbf{E}_A^g$, in $V \cup V_{logs}$. We also use an LLM to find domain-specific synonymous word relationships, $\mathbf{E}_S^d$, and domain-specific antonymous word relationships, $\mathbf{E}_A^d$, in $V_{logs} \setminus V$. We combine this information in $\mathbf{E}_S = \mathbf{E}_S^g \cup \mathbf{E}_S^d$ and $\mathbf{E}_A = \mathbf{E}_A^g \cup \mathbf{E}_A^d$. To be clear, the g in $\mathbf{E}_S^g$ refers to the *general* synonym relationships (and general antonym relationships in $\mathbf{E}_A^g$), while the d in $\mathbf{E}_S^d$ refers to the *domain-specific* synonym relationships (and domain-specific antonym relationships for $\mathbf{E}_A^d$).

We finally apply *LWET*, using $\hat{\mathbf{Q}}$, $\mathbf{E}_S$, and $\mathbf{E}_A$, to obtain a retrofitted embeddings table that considers lexical contrast $\mathbf{Q} \in \mathbb{R}^{\#(V \cup V_{logs}) \times d}$. The method for constructing the embedding space is summarized in Figure 3.1.

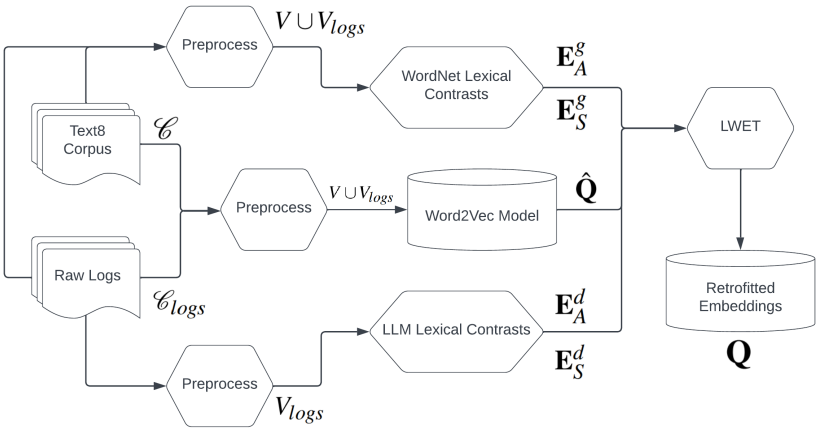

Figure 3.1 Pipeline for constructing the embedding space

The preprocessing of the general *Text8* corpus simply consists of splitting every sentence on spaces to obtain lists of words. Thus, we ultimately obtain a list of sentences, where each sentence is a list of words. The preprocessing of the raw logs differs between *HDFS_v1* and *BGL*. For the *HDFS_v1* dataset, we remove all non-alphabetical characters from each log, with the motivation that we are only interested in the natural language content of each log message. For the *BGL* dataset, we remove hexadecimal numbers and all non-alphabetical characters. If we do not remove the hexadecimal numbers before the non-alphabetical characters, the hexadecimal numbers will persist as nonsensical letter combinations in the messages. Finally, we split each log message on spaces to obtain a list of word lists, analogous to the preprocessed general corpus.

Parameter	Value
$step$	0.1
λ	0.5
t	100
α	1
β	2
γ	3
δ	4

Table 3.4 Hyperparameters for experimental *LWET* with positive sampling, $step$ refers to the step size in gradient descent, λ refers to the threshold (difference in the cost function Ψ as a stopping condition), t to the maximum number of iterations, and α, β, γ, as well as δ to the parameters that control the relative strength of each component in Ψ.

LWET. As an initial experiment, we implement the positive sampling approximation algorithm, presented in Listing 2.2, for the *HDFS_v1* dataset using the hyperparameters in Table 3.4.

To evaluate the transformation of the embedding space, we consider the cosine similarities between synonym and antonym pairs to be arbitrarily distributed. We can consequently plot *cumulative distribution functions* (*CDF*s), F, over the cosine similarities for the pairs of pretrained synonym and antonym embeddings found in $\hat{\mathbf{Q}}$ and the pairs of retrofitted synonym and antonym embeddings found in $\mathbf{Q}$.

Consider the following example for some intuition about what the *CDF*s represent: let $X_{\mathbf{Q}}^{syn}$ denote the distribution of the cosine similarities between synonym pairs in the retrofitted embedding space $\mathbf{Q}$, with the corresponding *CDF* $F_{\mathbf{Q}}^{syn}$. Then, $F_{\mathbf{Q}}^{syn}(0.8) = \mathbb{P}(X_{\mathbf{Q}}^{syn} \leq 0.8) = 0.2$ would mean that 20% of the synonym pairs in the retrofitted embedding space have a cosine similarity smaller than 0.8.

Naturally, we want the *CDF* corresponding to the cosine similarities of the antonym pairs to be shifted to the left, signifying lower similarities, and the *CDF* corresponding to the cosine similarities of the synonym pairs to be shifted to the right, signifying higher similarities. Moreover, the curves should ideally have a sharp slope, corresponding to lower variance in the cosine similarities.

Figure 3.2 demonstrates how the synonym pairs in our lexicon generally exhibit higher cosine similarities after applying *LWET*. The antonym pairs, however, also exhibit slightly higher cosine similarities after applying *LWET*. Nonetheless, the difference in cosine similarities between the synonym and antonym pairs has increased, potentially enabling us to separate antonyms and synonyms better. In general, however, this transformation does not entirely accomplish what we want; the cosine similarities between antonyms seem to be distributed very similarly to the cosine similarities between irrelevant words, and the difference between synonyms and antonyms may not be large enough to separate the two classes. Interestingly, the cosine similarity distribution of irrelevant word pairs has been shifted to generally higher values. It is reasonable to raise concerns about how the relation between seemingly irrelevant

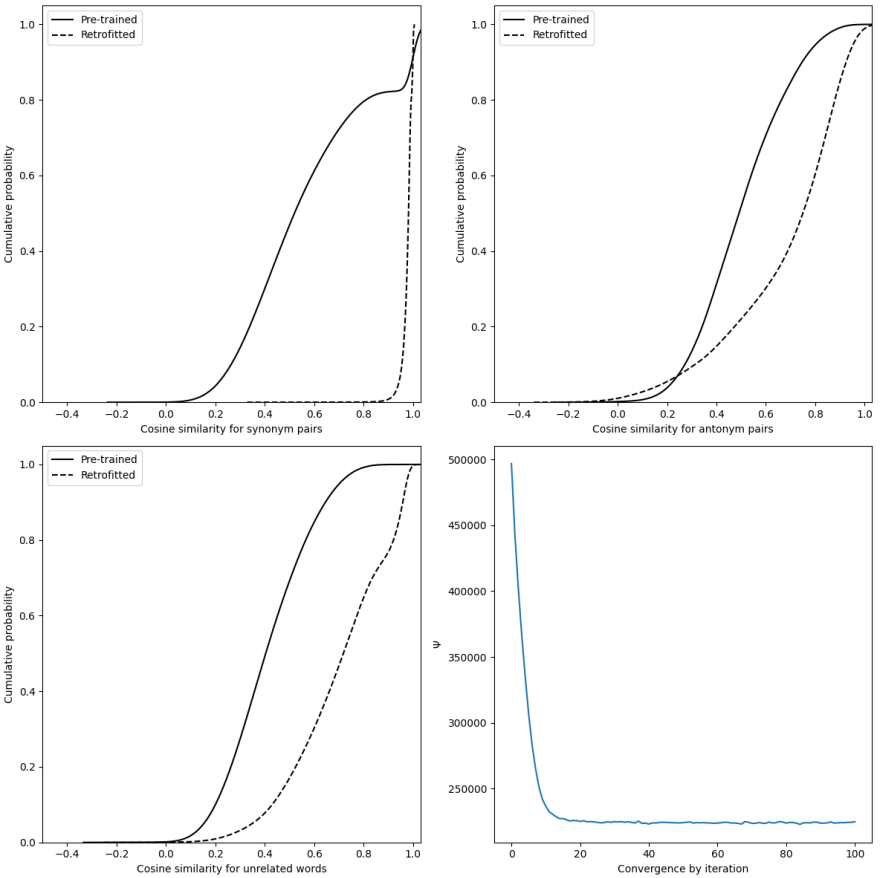

Figure 3.2 Cumulative distributions of the cosine similarities between the synonym, antonym, and irrelevant/unrelated word pairs for the pre-trained embeddings $\hat{\mathbf{Q}}$ and retrofitted embeddings $\mathbf{Q}$ using the hyperparameters in Table 3.4. The convergence of Ψ in LWET is shown in the bottom right plot.

words, in the context of synonyms and antonyms, is significantly altered by *LWET*, giving a higher similarity score. The *CDF* corresponding the the distribution of cosine similarities between irrelevant words in the pre-trained embeddings resembles the *CDF* of a Gaussian distribution. The *CDF* corresponding to the distribution in the retrofitted embeddings, however, does not look Gaussian. The convergence of ψ, shown in the bottom right image, is relatively fast, which might be a problem. We will make our reasoning as to why this might be, and how we have remedied this, later.

Empirically, we find that the total number of synonyms and antonyms in the lexicon affects the transformation of the embedding space. For example, the lexicon we use to tune our embedding space contains 147 195 total synonym pairs, but only 7187 antonym pairs. While N_S^i and N_A^i in the denominators of the cost function in Equation 2.13 attempt to compensate for these differences, there will inevitably be cases where a word has no antonyms but several synonyms in the lexicon; in these cases, the term in 2.13 that tries to decrease the distance between synonyms will contribute to the transformation of the embedding space, while the term that tries to increase the distance between antonyms will not. To further compensate for this disparity in the occurrences of synonyms and antonyms in the lexicon, we should scale β and γ so that fewer antonyms, relative to the number of synonyms, in the lexicon results in a larger γ and smaller β, and vice versa for relatively fewer synonyms. We therefore suggest a method to choose β and γ, where $\lceil x \rceil$ denotes the ceiling function, as in Equation 3.1.

$$\begin{aligned} \beta &= \max\left(2,\ \left\lceil \delta \cdot \frac{\#\text{antonyms}}{\#\text{synonyms} + \#\text{antonyms}} \right\rceil\right) \\ \gamma &= \max\left(3,\ \left\lceil \delta \cdot \frac{\#\text{synonyms}}{\#\text{synonyms} + \#\text{antonyms}} \right\rceil\right) \end{aligned} \tag{3.1}$$

The strategy in Equation 3.1 effectively means that $2 \leq \beta \leq \delta$ and $3 \leq \gamma \leq \delta$. The motivation for using these intervals is that we do not want β and γ to be smaller than the values suggested by Liu et al [11]. We furthermore do not want the values to overpower δ, that controls the distance to irrelevant words. An optimal choice of β and γ could be the one that maximizes the area between the *CDF* of the retrofitted antonyms and the *CDF* of the retrofitted synonyms, i.e., $\int F_{\mathbf{Q}}^{ant} - F_{\mathbf{Q}}^{syn}$, while simultaneously encapsulating a well-formed *CDF* of the irrelevant words. A more extensive hyperparameter tuning of *LWET* is, however, unfortunately outside the scope of this thesis. Applying the strategy in Equation 3.1, we obtain $\beta = 2$, as before, and $\gamma = 4$, yielding the distributions shown in Figure 3.3.

Considering Figure 3.3, it is evident that the distributions in the retrofitted embedding space have been shifted significantly. Almost 50% of the antonym pairs have a cosine similarity lower than 0.25, and more than 95% of the synonym pairs have a cosine similarity higher than 0.75. Furthermore, the cosine similarity distribution of irrelevant word pairs is, as intended, somewhere in between that of the synonym and antonym pairs. Nonetheless, the worrisome shift in distribution of cosine similarities between irrelevant word pairs persists.

Given the relatively fast convergence of Ψ, visualized in Figure 3.3, we try to decrease *step* from 0.1 to 0.01 and t, the maximum number of iterations, from 100 to 50. The reason, or

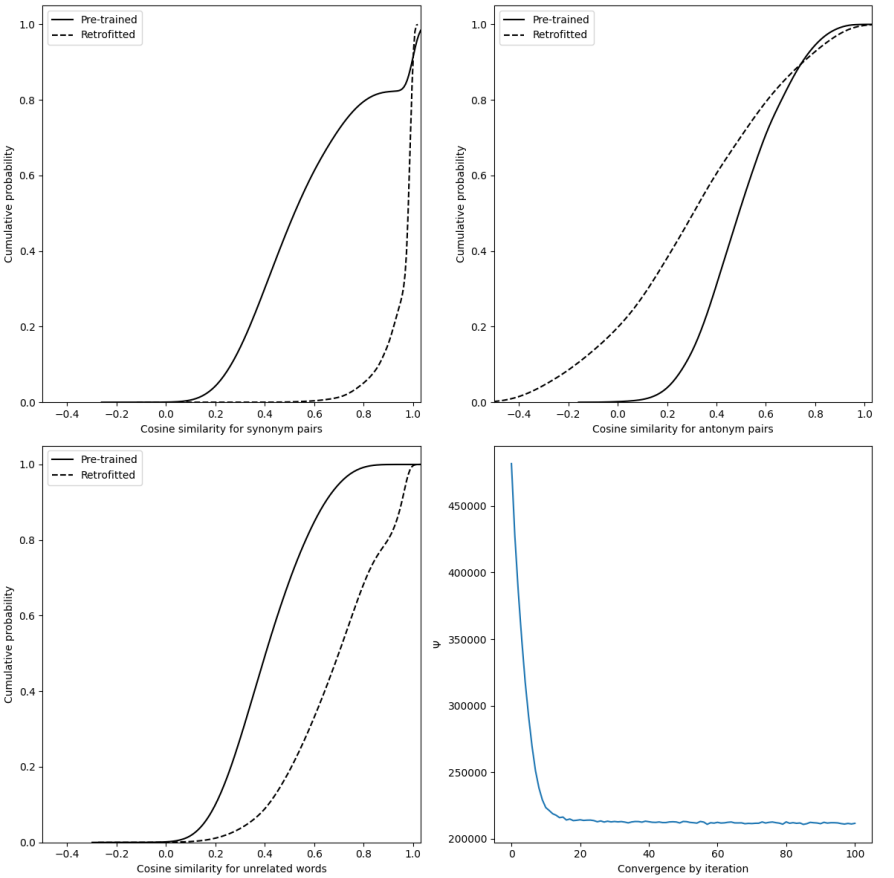

Figure 3.3 Cumulative distributions of the cosine similarities between the synonym, antonym, and irrelevant/unrelated word pairs for the pre-trained embeddings $\hat{\mathbf{Q}}$ and retrofitted embeddings $\mathbf{Q}$ using the hyperparameters in table 3.4, **except** for $\gamma = 4$. The convergence of Ψ in LWET is shown in the bottom right plot

our suspicion, is that continuously updating the retrofitted embedding space even after Ψ has converged to a local minimum might scramble the embedding space, potentially destroying some of the relations between irrelevant word pairs. As seen in Figure 3.4, this set of hyperparameters preserves the structure of the distributions better.

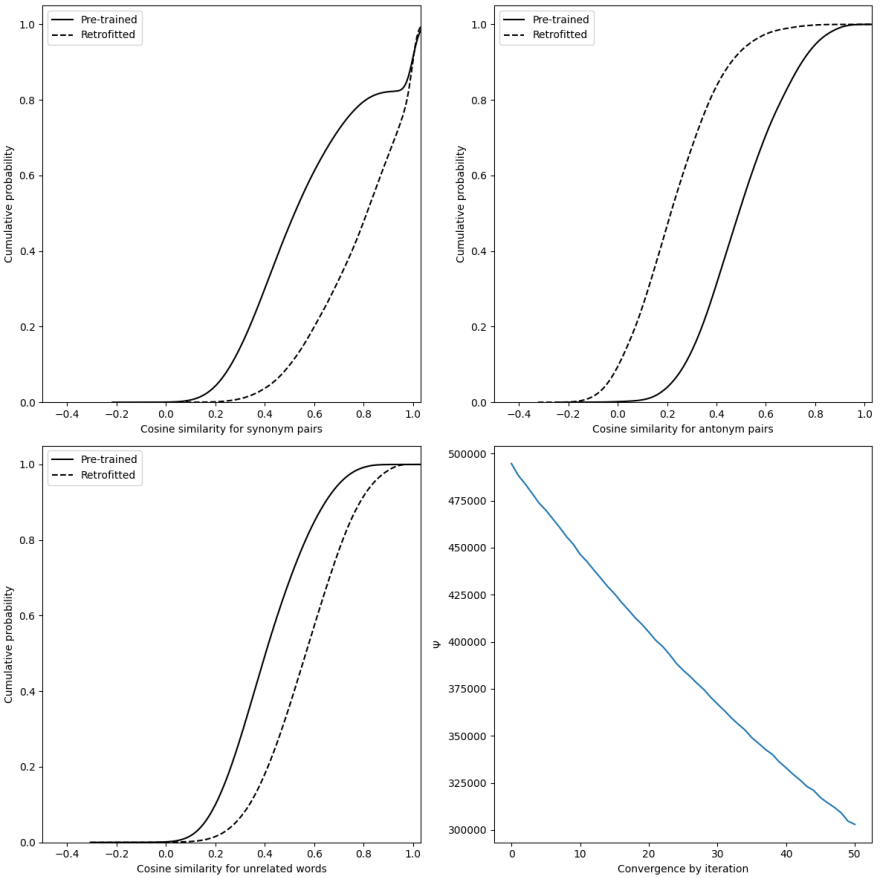

Figure 3.4 Cumulative distributions of the cosine similarities between the synonym, antonym, and irrelevant/unrelated word pairs for the pre-trained embeddings $\hat{\mathbf{Q}}$ and retrofitted embeddings $\mathbf{Q}$ using the hyperparameters in Table 3.4, **except** for $\gamma = 4$, $step = 0.01$, and $t = 50$. The convergence of Ψ in LWET is shown in the bottom right plot

OOV Words

Words that do not occur in the training data but do occur in the test data, in other words OOV words, do not exist in our embedding space. Therefore, we need an OOV engine to obtain embeddings for OOV words and accurately construct log message embeddings. Initially, we planned on using *MIMICK* to obtain embeddings for OOV words. However, as we will dis-

cuss later in the report, we found that *MIMICK*, at least for our purposes, did not work as intended. Therefore, we have opted for a different approach.

Our OOV engine first prompts an LLM to ask for synonyms and antonyms of the OOV word. If one of the provided synonyms occurs in our vocabulary, we use the synonym's embedding in place of the OOV word. If several of the provided synonyms occur in our vocabulary, we use the weighted average of the synonyms' embeddings in place of the OOV word. If none of the provided synonyms occur in our vocabulary, we in a sort of best-effort fallback use the Levenshtein distance to find a word and its corresponding embedding that most resembles the OOV word.

Log Message Embeddings

We construct a log message embedding by first extracting n individual words in the log message in a preprocessing step. If our vocabulary contains a word, we look it up in the retrofitted embeddings table to obtain its embedding. If our vocabulary does not contain a word, we use the OOV engine to obtain an embedding for it. The embeddings for each individual word, $q_i \in \mathbb{R}^d$, $i \in [1,n]$, are aggregated according to a simple strategy $\mathbf{e} = \frac{1}{n}\sum_{i=1}^{n} q_i$, where $\mathbf{e} \in \mathbb{R}^d$ is the log message embedding. The method for constructing the log message embedding is summarized in Figure 3.5.

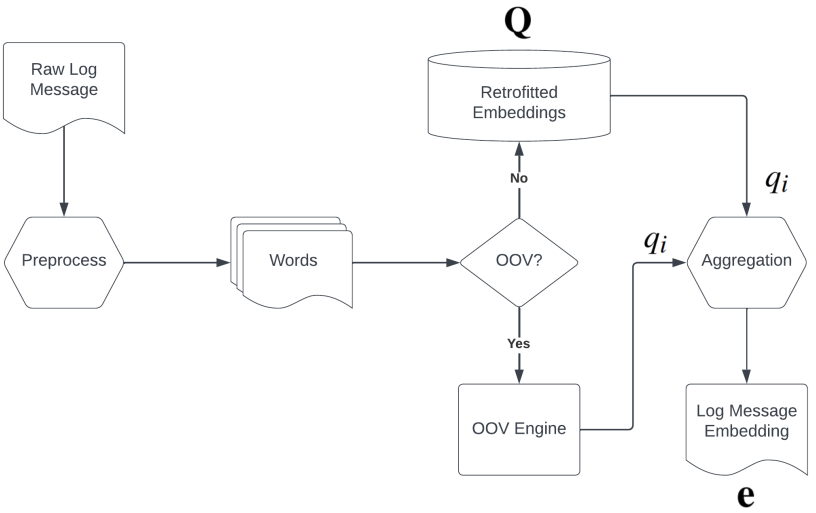

Figure 3.5 Pipeline for constructing a log message embedding

Using a class-uniform 80/20 split between training and test data, we cluster the training log message embeddings using *k-means*. In an inference step, we match each test log message embedding to the most similar cluster. Every cluster has a numerical identifier, and we can thus create event count vectors for each window of log messages to be used in three different

anomaly detection models: an *Isolation forest*, an *autoencoder*, and *logistic regression* using the same hyperparameters as in the baseline.

3.3 Evaluation

In summary, we test the baseline and the embeddings approach in two tasks: anomaly detection and concept drift resilience. The only difference between the anomaly detection and the concept drift resilience task is that we, in the concept drift resilience task, substitute some of the words in the test dataset before we create the event count matrices. The specific changes made to the log messages to induce concept drift can be found in the appendix, in Listing 7.1 and 7.2.

Using both the baseline with *Drain* and the embeddings approach, we evaluate the precision, recall, and F_1 score for both the training dataset and test dataset with the three anomaly detection models. Since we are interested in how the different components affect the embeddings approach, we test slightly different set-ups, presented in Table 3.5.

Standard params	The pipeline in Figure 3.5
W/o *LWET*	Like the standard parameters, except we do not use *LWET* to transform the embedding space
W/o OOV engine	Like the standard parameters, except we do not use the OOV engine to handle OOV words. Instead, every OOV word is simply assigned a vector of ones, $v = [1, \ldots, 1]$, as its embedding
CBOW	Like the standard parameters, except we use *CBOW* instead of *skip-gram* in *Word2Vec*

Table 3.5 Evaluated set-ups for the embedding approach

4

Results

4.1 LLM Lexicons

First, the synonyms and antonyms of different words produced by LLMs are compared against a set of manually annotated synonyms and antonyms. The scoring is explained in the method section. With the total amount of manually annotated synonyms and antonyms being 386, the maximum score was also 386. The "No JSON" column shows the number of times the LLM response did not conform to the given JSON format, either by explaining that the given word was unknown to the model or simply by giving the response in text. The "Prompt" column refers to the prompts in Table 3.3.

LLM	Prompt	Synonyms	Antonyms	No JSON	Score
LLaMA 2	1	54	30	21	84
	2	18	8	84	24
	3	29	11	49	39
GPT-3.5	1	83	42	3	125

Table 4.1 Synonym/antonym extraction performance on annotated *HDFS_v1* vocabulary

To be clear, this test was only performed on the *HDFS* dataset, since the amount of unique words in the *BGL* dataset was far too large to annotate.

Considering Table 3.3, prompt 1 undoubtedly performs the best. Having tested the three prompts with *LLaMA 2*, prompt 1 extracted more correct synonyms and antonyms compared to the other prompts, while also responding in the correct format more often. While *GPT-3.5* performs considerably better than *LLaMA 2*, none of the models came close to achieving a maximum score of 386. For more details on the performance of large language models as lexicons, we refer to Section 5.1 in the discussion.

4.2 Anomaly Detection

The results, with scores rounded to three decimals from three experiments each, for the anomaly detection task are presented in Tables 4.2 and 4.3. The rows refer to the set-ups described in Table 3.5. The *baseline* refers to the scores produced by using *Drain*. The highest scores for each model are marked in bold.

A quick glance at Tables 4.2 and 4.3 reveals similar anomaly detection performance when applying both *Drain* and log message embeddings in the parsing step. Comparing results between anomaly detection on the *HDFS_v1* and *BGL* datasets, we can also observe a small drop in scores when for log message embeddings on the *BGL* data. We refer to Section 5.2 in the discussion for further details and explanations. The impact of using LWET and the OOV engine seems, in most cases, to be negligible. For further details about the impact of LWET and the OOV engine, we refer to Section 5.4 in the discussion.

HDFS_v1

The embeddings were fit into $k = 45$ clusters, the same amount that *Drain* produced after parsing the *HDFS* logs.

Isolation Forest						
	Training Dataset			**Test Dataset**		
	Precision	Recall	F_1 score	Precision	Recall	F_1 score
Baseline	0.942	0.707	0.808	**0.943**	0.690	**0.797**
Standard params	0.880	0.682	0.768	0.874	0.666	0.756
W/o LWET	0.878	0.688	0.771	0.870	0.670	0.757
W/o OOV engine	0.883	0.692	0.776	0.879	0.679	0.766
CBOW	0.904	0.705	0.792	0.904	**0.694**	0.785
Logistic Regression						
	Training Dataset			**Test Dataset**		
	Precision	Recall	F_1 score	Precision	Recall	F_1 score
Baseline	0.994	0.798	0.890	**0.992**	**0.785**	**0.876**
Standard params	0.994	0.797	0.885	0.991	0.784	0.876
W/o LWET	0.994	0.797	0.885	0.991	0.784	0.876
W/o OOV engine	0.994	0.798	0.885	0.991	0.785	0.876
CBOW	0.994	0.798	0.885	0.991	0.785	0.876
Autoencoder						
	Training Dataset			**Test Dataset**		
	Precision	Recall	F_1 score	Precision	Recall	F_1 score
Baseline	0.890	0.670	0.761	0.886	0.658	0.756
Standard params	0.889	0.683	0.772	**0.886**	**0.674**	**0.765**
W/o LWET	0.889	0.683	0.772	0.886	0.674	0.765
W/o OOV engine	0.889	0.683	0.772	0.886	0.674	0.765
CBOW	0.889	0.682	0.772	0.886	0.674	0.765

Table 4.2 Anomaly detection performance on *HDFS_v1* dataset

BGL

The embeddings were fit into $k = 390$ clusters, the same amount that *Drain* produced after parsing the *BGL* logs.

Isolation Forest						
	Training Dataset			**Test Dataset**		
	Precision	Recall	F_1 score	Precision	Recall	F_1 score
Baseline	0.744	0.744	0.744	**0.731**	**0.75**	**0.74**
Standard params	0.700	0.700	0.700	0.678	0.712	0.694
W/o LWET	0.700	0.700	0.700	0.659	0.728	0.692
W/o OOV engine	0.684	0.683	0.683	0.695	0.728	0.711
CBOW	0.687	0.687	0.687	0.653	0.711	0.681
Logistic Regression						
	Training Dataset			**Test Dataset**		
	Precision	Recall	F_1 score	Precision	Recall	F_1 score
Baseline	0.896	0.868	0.882	**0.900**	0.833	0.866
Standard params	0.911	0.899	0.905	0.885	0.842	0.863
W/o LWET	0.908	0.891	0.900	0.890	**0.855**	**0.872**
W/o OOV engine	0.909	0.884	0.896	0.888	0.833	0.860
CBOW	0.893	0.892	0.893	0.864	0.838	0.851
Autoencoder						
	Training Dataset			**Test Dataset**		
	Precision	Recall	F_1 score	Precision	Recall	F_1 score
Baseline	0.754	0.754	0.754	**0.718**	**0.781**	**0.748**
Standard params	0.639	0.639	0.639	0.634	0.675	0.654
W/o LWET	0.641	0.641	0.641	0.633	0.658	0.645
W/o OOV engine	0.638	0.638	0.638	0.633	0.658	0.645
CBOW	0.641	0.641	0.641	0.628	0.658	0.642

Table 4.3 Anomaly detection performance on *BGL* dataset

4.3 Concept Drift Resilience

The results for the concept drift resilience task, in the same format as in the anomaly detection task, are presented in Table 4.4 and 4.5.

HDFS_v1

Isolation Forest						
	Training Dataset			*Test Dataset*		
	Precision	Recall	F_1 score	Precision	Recall	F_1 score
Baseline	0.942	0.707	0.808	0.104	**0.800**	0.185
Standard params	0.880	0.682	0.768	0.878	0.741	0.804
W/o LWET	0.878	0.688	0.771	0.862	0.664	0.751
W/o OOV engine	0.883	0.692	0.776	0.888	0.739	**0.807**
CBOW	0.904	0.705	0.792	**0.897**	0.660	0.760
Logistic Regression						
	Training Dataset			*Test Dataset*		
	Precision	Recall	F_1 score	Precision	Recall	F_1 score
Baseline	0.994	0.798	0.890	0.047	**0.999**	0.089
Standard params	0.994	0.797	0.885	0.972	0.785	0.869
W/o LWET	0.994	0.797	0.885	0.991	0.785	0.876
W/o OOV engine	0.994	0.798	0.885	0.972	0.785	0.869
CBOW	0.994	0.798	0.885	**0.991**	0.785	**0.876**
Autoencoder						
	Training Dataset			*Test Dataset*		
	Precision	Recall	F_1 score	Precision	Recall	F_1 score
Baseline	0.890	0.670	0.761	0.615	0.332	0.431
Standard params	0.889	0.683	0.772	**0.886**	**0.674**	**0.765**
W/o LWET	0.889	0.683	0.772	0.886	0.674	0.765
W/o OOV engine	0.889	0.683	0.772	0.886	0.674	0.765
CBOW	0.889	0.683	0.772	0.886	0.674	0.765

Table 4.4 Anomaly detection performance with concept drift on *HDFS_v1* dataset

BGL

Isolation Forest						
	Training Dataset			*Test Dataset*		
	Precision	Recall	F_1 score	Precision	Recall	F_1 score
Baseline	0.744	0.744	0.744	0.476	**0.987**	0.642
Standard params	0.664	0.664	0.664	0.644	0.675	0.66
W/o LWET	0.662	0.662	0.662	0.668	0.68	0.674
W/o OOV engine	0.673	0.673	0.673	**0.717**	0.724	**0.721**
CBOW	0.675	0.675	0.675	0.665	0.68	0.672
Logistic Regression						
	Training Dataset			*Test Dataset*		
	Precision	Recall	F_1 score	Precision	Recall	F_1 score
Baseline	0.896	0.868	0.882	0.712	**0.873**	0.783
Standard params	0.916	0.897	0.906	**0.897**	0.842	**0.869**
W/o LWET	0.910	0.900	0.905	0.888	0.838	0.862
W/o OOV engine	0.915	0.901	0.908	0.888	0.838	0.862
CBOW	0.889	0.890	0.890	0.871	0.823	0.849
Autoencoder						
	Training Dataset			*Test Dataset*		
	Precision	Recall	F_1 score	Precision	Recall	F_1 score
Baseline	0.754	0.754	0.754	0.478	**0.987**	0.644
Standard params	0.645	0.645	0.645	**0.637**	0.662	**0.649**
W/o LWET	0.642	0.642	0.642	0.632	0.662	0.647
W/o OOV engine	0.642	0.642	0.642	0.628	0.667	0.647
CBOW	0.639	0.639	0.639	0.623	0.654	0.638

Table 4.5 Anomaly detection performance with concept drift on *BGL* dataset

The anomaly detection scores after introducing concept drift, presented in Tables 4.4 and 4.5, display significant differences in performance between using *Drain* and log message embeddings. Here, the baseline scores have dropped substantially, especially in terms of precision.

The log message embeddings scores, however, remain consistent. This phenomenon is most noticeable when performing anomaly detection on the *HDFS_v1* data. For further details and explanations on concept drift resilience, we refer to Section 5.3 in the discussion.

5

Discussion

To facilitate a discussion, we first establish some trends in the results.

1. As shown in table 4.1, the first prompt most effectively identifies the synonyms and antonyms of the *HDFS_v1* vocabulary when evaluated using *LLaMA* 2. Notably, *ChatGPT* (GPT-3.5) demonstrates superior performance compared to *LLaMA* 2 in this task
2. The results in tables 4.2 and 4.3 indicate that in the anomaly detection task, the embeddings approach yields results comparable to, albeit marginally inferior to, the *Drain* baseline
3. According to the findings in tables 4.4 and 4.5, the embeddings approach exhibits significantly better performance for the *HDFS_v1* dataset and a modest improvement for the *BGL* dataset in the concept drift resilience task
4. The omission of *LWET* or the OOV engine in tables 4.2, 4.3, 4.4, and 4.5 does not seem to noticeably impact the performance of the embeddings approach in either the anomaly detection or concept drift resilience task

5.1 LLMs as Lexicons

Given the scores in Table 4.1, there is undoubtedly room for improvement when it comes to using LLMs as lexicons for synonyms and antonyms. On a brighter note, GPT-3.5 still performed relatively well, missing out on some key relationships, such as not finding that "dest" should be synonymous with "destination", but also finding some key relationships, such as "transmitted" being antonymous with "received" and "source" being synonymous with "src". The score in Table 4.1, not even reaching half of the maximum score, may not reflect this fact properly. Nonetheless, given that there is no standardized way to compare the synonyms and antonyms, the score should not be regarded as an absolute truth. In some ways, *ChatGPT* actually outperformed the manually annotated synonyms and antonyms, finding words that should have been included among the annotated words.

A notable challenge in using *ChatGPT* for this task was its tendency to generate exactly ten synonyms and ten antonyms. Sometimes, there was a drift in the meaning and context of the

generated words as the model continued to provide synonyms and antonyms. For instance, the model claims that "tendon" is synonymous with "thread", which may be true in a few logical leaps but is generally not true. The reason for this behaviour may be the model misinterpreting the instruction to find a *maximum* of ten synonyms and antonyms as a requirement to always find ten. If the quality of the generated synonyms and antonyms is poor, the log message embeddings would be contaminated by the embeddings of these words, skewing the resulting vector away from the intended meaning. Some more extreme/nonsensical examples of this would be claiming "chaos creator" is antonymous with "sorter" (very creative!), and that "disorganization" is antonymous with "file".

The *LLaMA* 2 model performed worse than *ChatGPT*, especially in conforming to the provided JSON format. Surprisingly, prompt 2 and 3, that contain more information and examples, perform significantly worse than prompt 1. This could potentially be the effect of an "information overload"; *LLaMA* simply loses attention and focus on the target word with a longer prompt.

Another issue was the differentiation between parts of speech. Some words are both nouns and verbs, for example "block", that may mean completely different things. A potential solution lies in refining the prompts. Additionally, the context within which synonyms and antonyms were found often lacked precision. Words can have different meanings in various fields, and *ChatGPT* tended to default to the most generic usage. For instance, *ChatGPT* listed "brook", "creek", and "current" as synonyms for "stream", potentially leading to irrelevant log message embeddings. Drawing from the principles of prompt engineering in *OpenAI* [19], the LLMs could possibly have been instructed to act as a computer engineer, or at least have been given the proper context, along with being specified to only give one word answers.

Finally, the models frequently produced synonyms and antonyms that were not part of our established vocabulary. A possible solution to this issue is to refine the prompts using a list of words from our vocabulary that exhibit the highest cosine similarity to the target word. Assuming that synonyms and antonyms typically appear in similar contexts, they should demonstrate high cosine similarities within an embedding space. Therefore, by requesting an LLM to identify potential synonyms and antonyms from a curated list of highly similar words, we could likely achieve more precise and relevant word associations.

After postexperimental readjustments of the prompts, it was noted that adding "no need to give exactly 10, include only strict synonyms" improved the model's tendency to always respond with a set number of synonyms and antonyms. Unfortunately, we did not notice this during initial testing. In the initial tests, we used words with 10 clear synonym and antonym candidates. As such, we did not notice the problem until much later in the process.

5.2 Anomaly Detection

Considering the results in Table 4.2 and 4.3, there is not a large difference in scores between the baseline and the embeddings approach. One difference between the implementations of the baseline and the embeddings approach, that may make a difference, is that the baseline

has a designated dimension in the event count vectors for "non-matchable" log messages. The messages that are not matched to a cluster in the baseline (that is, the messages that *Drain* is not able to parse) are presumably rare and distinct, since the *parsing tree* that is generated during the training phase can not accommodate them. In the embeddings approach, these rare and distinct messages are simply matched to the closest cluster. The clustering approach does not consider how well the message fits into the cluster; it simply considers what the closest cluster is. This may lead to overestimating the occurrence of certain events, using the embeddings approach, in the event count vectors.

An important remark to make is that we split the training and test dataset in a class-uniform way, meaning that the datasets are divided so that both have an equal share of anomalies. This is, of course, not viable in a live setting where labels do not exist. Furthermore, it is far from true that anomalies will occur at the same rate in the data used to train a model and the new incoming data in a live setting. Nonetheless, a class-uniform training-test split is common practice in many articles on log anomaly detection. Many algorithms, such as *Isolation forests* and *autoencoders* that use a contamination rate in order to classify anomalies, are arguably more sensitive to the share of anomalies in the datasets. For example, an *autoencoder* that uses the contamination rate as a threshold for classifying anomalies based on reconstruction error may either under- or overestimate anomalies if the contamination rate is too low or too high. Models based on recurrent neural networks, such as *LSTM*-based *DeepLog* and *LogAnomaly*, generally try to predict the next log template in a sequence. These models are more robust to varying shares of anomalies since they do not rely on a contamination rate to classify anomalies. Considering the aim of this thesis, which was to explore the feasibility of using automated log message embeddings for anomaly detection and not the anomaly detection itself, we decided on a class-uniform approach to obtain results comparable to those in much of the literature.

In general, models that require a contamination rate as a hyperparameter may not be ideal in an anomaly detection context. It is simply difficult to estimate what the rate of anomalies in a dataset is in a real, unsupervised scenario. Moreover, the contamination rate may differ between training and other data. Other models, such as the previously mentioned *LSTM*-based models, should perhaps be favored.

5.3 Concept Drift Resilience

The ability of a log anomaly detection model to tolerate concept drift, changes in the format of log messages over time, is important. For example, a developer may rewrite a log message when refactoring a code base. Of course, one approach to account for concept drift could be to occasionally retrain a model. This can, however, often be a computationally expensive task. Some models also propose an online update mechanism, where previously unseen events can be added to the model while running. Arguably, online updates have two primary disadvantages: they further convolute the models, making them harder to implement, and they potentially increase the dimension of input by adding new clusters of logs. Log message embeddings conceptually have some understanding of the natural language content of the messages and can potentially match a previously unseen message to a semantically similar

cluster. The results in Table 4.4 and 4.5 demonstrate a significantly better ability to absorb concept drift in the *HDFS_v1* dataset, and a marginally better ability for the *BGL* dataset.

The embeddings approach's superior performance in the concept drift resilience task for the *HDFS_v1* dataset can potentially be attributed to several factors. Firstly, the *HDFS_v1* dataset is relatively "homogeneous", in the sense that there is not a large variety in log messages. For example, *Drain* only finds 45 templates for the dataset, compared to 390 for the *BGL* dataset, indicating a smaller variety of messages. This, in turn, leads to a small set of changes in phrasing influencing the *HDFS_v1* dataset to a greater extent than, for example, in the *BGL* dataset. A more heterogeneous log dataset, such as *BGL*, can absorb more word substitutions simply because they do not impact the same number of log messages.

An important note is that *Drain* is especially sensitive against first-word changes. This is, most likely, due to the fact that *Drain* parses a log message from left to right. The first level in the parse tree considers the length of the log message, while the second level considers the first word. If the first word is changed to one that *Drain* has not seen before, it will simply fail in parsing the log message, which was the case for some of the word substitutions in Listing 7.1, such as "received" and "transmitted".

Interestingly, the *autoencoder* seems more robust against concept drift considering the baseline performance which, despite a large decrease, did not deteriorate as much as the *Isolation forest* or *logistic regression*. The reason for this robustness is not entirely clear; it could be an inherent quality of the *autoencoder* or a result of our particular implementation. In our specific implementation, we do use a dropout regularization strategy which can decrease overfitting. A key distinction that might give the *autoencoder* an edge over models like the *Isolation forest* is that the *autoencoder* considers the input "all at once". That is, the event count vectors are passed through the network and combined in the nodes, as opposed to an *Isolation forest* that considers one feature at a time. In a concept drift scenario, the distribution of one feature, i.e., one event, may change drastically. If we imagine an *Isolation tree* that considers *one* feature with *one* split value at a time, a shift in the distribution of the feature may render the split value outdated and decrease anomaly detection performance. With an *autoencoder*, however, all of the features are multiplied with a set of weights in the network and passed through an activation function. Consequently, even if one feature undergoes a drastic distribution change, the other features continue to contribute to the value processed by the activation function, potentially enhancing the model's ability to handle concept drift.

5.4 LWET and the OOV Engine

Given the scores in Table 4.2, 4.3, 4.4, and 4.5, omitting *LWET* and the OOV engine from the embeddings approach did not noticeably impact performance. In terms of omitting the OOV engine, we did not find the negligible performance difference surprising. Our training dataset was rather large, and more or less the entire test dataset's vocabulary was contained in the training dataset. In fact, we only found 3 OOV words in the *HDFS_v1* test dataset and 4 in the *BGL* test dataset. In general, the LLM could find suitable synonyms for OOV words. However, the same challenges we had with using LLMs as lexicons were also present here. The Levenshtein distance was a laudable fallback. For example, some OOV words, such

as “grepa” and “grepb”, had trailing letters. It is possible that the trailing letters emanate from flags for the Unix command “grep”, such as in “grep -a”. The Levenshtein distance successfully managed to match both examples with the in-vocabulary word “grep”.

Before we settled on the current OOV engine strategy, we intended to use *MIMICK*. Simply put, *MIMICK* would be trained on the embeddings produced by *Word2Vec* and learn to generate new embeddings for OOV words. Nonetheless, after an initial testing of *MIMICK*, we opted for a different approach. Given that *MIMICK* analyzes words on the basis of single characters, we expected that it would be able to find the proper embeddings for words where a single character is wrong. However, the average cosine similarity between the embeddings for the correctly spelled word and the embeddings produced by *MIMICK* for misspelled words was only 0.412, close to the average cosine similarities between irrelevant words in Figure 3.4.

With the suspicion that our specific training of *MIMICK* was sub-optimal, we also tested *MIMICK* in *Log2Vec*'s implementation. Initially, this implementation seemed to perform better; the embeddings produced by *MIMICK* for misspelled OOV words had an average cosine similarity around 0.99 with the corresponding correctly spelled words. To further validate whether this implementation indeed performed better, we also calculated the cosine similarities between a misspelled words and *all* the other correctly spelled words. In this specific test, we found that the OOV words had an average cosine similarity around 0.99 with *all* the other correctly spelled words. Words such as “block” and valid OOV misspellings such as “blook” could as such have promising cosine similarities of 0.998, but also share the same similarity with gibberish OOV words such as “dfj”. The *Log2Vec* implementation only tests the cosine similarities between an OOV word and its corresponding correctly spelled word, meaning that the other similarities, such as the one demonstrated, went unnoticed. It is difficult to deduce whether this is an inherent weakness in *MIMICK* or an error on our part. Nevertheless, it caused enough concern to make us opt for a different approach.

In our specific case, *LWET* did not alter the clustering of log message embeddings. The only candidates for two “opposite” events in the *HDFS_v1* dataset are “received block...” and “transmitted block”. However, the messages were, given our specific set-up, different enough to be placed in separate clusters, with or without *LWET*. GPT-3.5 indeed finds “transmitted” as an antonym to “received”, and vice versa, and *LWET* places the embeddings for each respective word further apart. However, the aggregation method, which is the weighted average of each word embedding q, does to some extent suppress this difference. Intuitively, as n in $\mathbf{e} = \frac{1}{n}\sum_{i=1}^{n} q_i$ grows, the significance of a single antonym pair in a total log message embedding $\mathbf{e}$ is diminished.

Arguably, the choice of k in *k-means clustering* largely affects the importance of handling lexical contrast in embeddings. Intuitively, there is a maximum number of possible event templates in a dataset of log messages. As k potentially exceeds this maximum number of possible event templates, the importance of handling lexical contrast decreases, since the log embeddings, even though they may contain synonym and antonym pairs, will be sufficiently dissimilar to end up in different clusters. We can think of it as the *pigeonhole principle*, where the maximum number of possible event templates, say n, must be put into k containers. It is possible that $k = 45$ exceeds this theoretical maximum number of possible event templates n.

If $k > n$, that would imply that at least one event template must be placed into two separate clusters. If we manually inspect the clusters that *Drain* found with the tested parameters, there are signs of essentially the same events occurring several times. For example, there are two clusters:

1. `<DATETIME> 19 INFO <RESOURCE>: BLOCK* ask <*> to replicate <BLOCK > to datanode(s) <*> <*>`
2. `<DATETIME> 19 INFO <RESOURCE>: BLOCK* ask <*> to replicate <BLOCK > to datanode(s) <*>`

These two clusters essentially represent the same event. Moreover, there are three clusters with some variation of "receiving block". On the other hand, if $k < n$, handling OOV words and lexical contrast correctly may be more important; several templates must now be placed in the same clusters and it is important that we can differentiate between events that are semantically similar but represent widely different system events.

However, even if two "opposite" events were merged into the same dimension in an event count vector, it may not, depending on the type of anomalies in the dataset and depending on the model, be to the detriment of performance. For instance, consider an event count vector $\mathbf{e} = [2,2,0]$ where the first and second dimension correspond to two opposite events: "sending..." and "receiving...". If the opposite events are merged, producing $\hat{\mathbf{e}} = [4,0]$, is there truly a complete loss of information about the event sequence? In fact, when we reduce the number of clusters k in the embeddings approach to only 10 for the *HDFS_v1* dataset, much lower than the 45 clusters used in Table 4.2, the performance is not significantly affected, as demonstrated in Table 5.1.

Model	Precision	Recall	F_1 score
Isolation Forest	0.898	0.692	0.781
Logistic Regression	0.974	0.627	0.763
Autoencoder	0.736	0.577	0.647

Table 5.1 Anomaly detection performance for the *HDFS_v1* dataset using the embeddings approach with standard parameters and $k = 10$ in *k-means clustering*

The results in Table 5.1 are somewhat disconcerting, especially considering that supposedly opposite events such as "received block..." and "transmitted block..." are placed in the same cluster with $k = 10$. It is possible that, given our choice of feature extraction method and anomaly detection models, it suffices to look at the quantitative relationship between very general groups of log messages to find log anomalies.

If we recall the theory about quantitative and sequential anomalies (where quantitative anomalies refer to an anomaly in the relation, quantitatively, between one event and another, and sequential anomalies refer to an anomaly in the typical flow between event types) event count vectors, as a feature extraction method, in some sense try to capture both. Even though the vectors do not contain the exact sequence of events, the count of each event type does

provide some information about the particular sequence of events in the sense that one typical event may be missing, that a rare event may have occurred, or that the events, in relation to each other, may have occurred with an unusual frequency. There may be an inherent weakness, at least when looking at anomalies in the context of quantitative and sequential anomalies, in using event count vectors as a feature extraction method. By trying to account for both quantitative and sequential aspects, event count vectors may not be optimally tailored for either. Models that look at a sequence of events, such as *DeepLog* and *LogAnomaly*, may be more suitable since they potentially have more information about a specific sequence of log events. For these specific models, it may also be increasingly important to accurately cluster the log events. For example, it may be more important to precisely know that "received block..." occurred after "transmitted block...", rather than knowing that either one occurred after the other.

5.5 Future Work

Several aspects of the model and method can be improved. Here are a selection of them.

- **Prompts**. As mentioned before, the used prompts could, and should, be further refined to obtain more contextually accurate synonyms and antonyms. The most interesting way forward would be to instruct the language model to act as an engineer, and using a list of candidate words from our vocabulary that exhibit the highest cosine similarity to the target word.
- **Word Embeddings**. More sophisticated embedding models than *Word2Vec* could be used. For example, *BERT* can potentially provide more accurate word embeddings. Moreover, it would be interesting to approach the "lexical contrast problem" from a fine-tuning perspective instead of an optimization perspective.
- **Aggregation Method**. Another interesting task would be to improve the aggregation method, with the purpose of increasing the importance of antonym pairs. Currently, the log message embedding is calculated as the weighted average of all the contained word embeddings, which, as the amount of words in a log message increases, diminishes the effect of any potential synonyms and antonyms.
- **Negations**. Similar to how synonyms and antonyms help separate "opposite" events, negations should be considered as well. In log messages containing segments such as "...received file..." and "...did not receive file...", "received" and "not receive" are essentially an antonym pair. One naive solution could be to consider *2-grams* instead of single words.
- **Anomaly Detection Models.** More anomaly detection models should be tested. Models that do not depend on contamination rate are especially interesting. Moreover, it would be interesting to explore how the log message embeddings themselves, which are already numerical representations, could be used instead of converting log sequences into event count vectors as a feature extraction method. If we look even further past the scope of this thesis, transformers also have potential, as highlighted by several recent papers, in log anomaly detection.

6

Conclusion

In conclusion, our exploration of log message embeddings as an alternative to traditional parsing methods, exemplified by *Drain*, has yielded promising results and raised valuable insights. The primary objective of this thesis was twofold: firstly, to investigate the feasibility, challenges, and potential advantages of employing embeddings for log analysis, particularly in terms of resilience against concept drift; and secondly, to explore the possibilities of automating the log embedding pipeline using large language models.

Our findings suggest that log message embeddings indeed present a viable and promising alternative to traditional parsing methods such as *Drain*. The main challenges lie in handling lexical contrast, such as antonyms occurring in separate log events, and out-of-vocabulary words. In our experiments, we have demonstrated how an embeddings approach can achieve anomaly detection results comparable with *Drain*. Moreover, we have demonstrated how an embeddings approach, with embedded knowledge about the language in log messages, can absorb and improve performance, in comparison with *Drain*, with concept drift over time.

Experiments pertaining to the opportunities to automate the log embedding pipeline (that is, in extracting synonyms and antonyms as well as finding suitable in-vocabulary candidates for out-of-vocabulary words) by using large language models have also demonstrated some potential. However, a fair amount of work remains to achieve performance comparable to manual operator feedback. To improve accuracy, refining prompts and contextualizing them within specific domains, like computer engineering, could be beneficial.

Bibliography

[1] A. Agresti. *An introduction to categorical data analysis*. Wiley, New York, 1996, pp. –. ISBN: 0471113387 9780471113386. URL: http://www.worldcat.org/search?qt=worldcat_org_all&q=0471113387.

[2] D. Borthakur. *Hdfs architecture guide*. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html. Accessed: [2023-12-07]. 2008.

[3] Z. Chen, J. Liu, W. Gu, Y. Su, and M. R. Lyu. "Experience report: deep learning-based system log analysis for anomaly detection". *CoRR* **abs/2107.05908** (2021). arXiv: 2107.05908. URL: https://arxiv.org/abs/2107.05908.

[4] A. Farzad and T. A. Gulliver. "Unsupervised log message anomaly detection". *ICT Express* **6**:3 (2020), pp. 229–237. ISSN: 2405-9595. DOI: https://doi.org/10.1016/j.icte.2020.06.003. URL: https://www.sciencedirect.com/science/article/pii/S2405959520300643.

[5] E. D. Giacinto. *Localai: the free, open source openai alternative*. https://github.com/go-skynet/LocalAI. 2023.

[6] P. He, J. Zhu, Z. Zheng, and M. R. Lyu. "Drain: an online log parsing approach with fixed depth tree". In: *2017 IEEE International Conference on Web Services (ICWS)*. 2017, pp. 33–40. DOI: 10.1109/ICWS.2017.13.

[7] S. He, J. Zhu, P. He, and M. R. Lyu. "Experience report: system log analysis for anomaly detection". In: *27th IEEE International Symposium on Software Reliability Engineering, ISSRE 2016, Ottawa, ON, Canada, October 23-27, 2016*. IEEE Computer Society, 2016, pp. 207–218. DOI: 10.1109/ISSRE.2016.21. URL: https://doi.org/10.1109/ISSRE.2016.21.

[8] S. He, J. Zhu, P. He, and M. R. Lyu. "Loghub: A large collection of system log datasets towards automated log analytics". *CoRR* **abs/2008.06448** (2020). arXiv: 2008.06448. URL: https://arxiv.org/abs/2008.06448.

[9] V. I. Levenshtein. "Binary codes capable of correcting deletions, insertions, and reversals". In: *Doklady Akademii Nauk SSSR*. Vol. 163. 4. Available at MathNet: http://mi.mathnet.ru/dan31411, MathSciNet: http://mathscinet.ams.org/mathscinet-getitem?mr=0189928, Zentralblatt MATH: https://zbmath.org/?q=an:0149.15905. 1965, pp. 845–848. URL: http://mi.mathnet.ru/dan31411.

[10] F. T. Liu, K. M. Ting, and Z.-H. Zhou. "Isolation forest". In: *2008 Eighth IEEE International Conference on Data Mining*. 2008, pp. 413–422. DOI: 10.1109/ICDM.2008.17.

[11] 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.

[12] J. B. MacQueen. "Some methods for classification and analysis of multivariate observations". In: L. M. L. Cam et al. (Eds.). *Proc. of the fifth Berkeley Symposium on Mathematical Statistics and Probability*. Vol. 1. University of California Press, 1967, pp. 281–297.

[13] W. Meng, Y. Liu, Y. Huang, S. Zhang, F. Zaiter, B. Chen, and D. Pei. "A semantic-aware representation framework for online log analysis". In: *2020 29th International Conference on Computer Communications and Networks (ICCCN)*. 2020, pp. 1–7. DOI: 10.1109/ICCCN49398.2020.9209707.

[14] 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.

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

[16] 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.

[17] K. A. Nguyen, S. Schulte im Walde, and N. T. Vu. "Integrating distributional lexical contrast into word embeddings for antonym-synonym distinction". In: K. Erk et al. (Eds.). *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. Association for Computational Linguistics, Berlin, Germany, 2016, pp. 454–459. DOI: 10.18653/v1/P16-2074. URL: https://aclanthology.org/P16-2074.

[18] 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.

[19] OpenAI. *Prompt engineering*. Accessed: [2023-12-11]. 2023. URL: https://platform.openai.com/docs/guides/prompt-engineering.

[20] Y. Pinter, R. Guthrie, and J. Eisenstein. “Mimicking word embeddings using subword RNNs”. In: *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Copenhagen, Denmark, 2017, pp. 102–112. DOI: 10.18653/v1/D17-1010. URL: https://aclanthology.org/D17-1010.

[21] A. Singhal and I. Google. “Modern information retrieval: a brief overview”. *IEEE Data Engineering Bulletin* **24** (2001).

[22] 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.

7

Appendices

7.1 Concept Drift Changes

Word changes made to induce concept drift in the HDFS logs. The keys in the dictionary are the former words, and the values are their replacements.

```
hdfs_changes = {
    'reset': ' restarted',
    'received': 'got',
    'user': 'usr',
    'but': 'though',
    'delete': 'remove',
    'while': 'when',
    'transmitted': 'sent',
    'millis': 'millisecond',
    'replicate': 'copy',
    'got': 'received',
    'interrupted': 'stopped',
    'starting': 'initiating',
    'terminating': 'stopping',
    'src': 'source',
    'dest': 'destination'
}
```

Listing 7.1 HDFS word changes to induce concept drift

Word changes made to induce concept drift in the BGL logs. The keys in the dictionary are the former words, and the values are their replacements.

```
bgl_changes = {
    'interrupt': 'stop',
    'critical': 'crucial',
    'program': 'prog',
    'corrected': 'fixed',
```

```
    'instruction': 'instr',
    'file': 'resource',
    'source': 'src',
    'detected': 'found',
    'correctable': 'fixable',
    'max': 'maximum',
    'maximum': 'max',
    'single': 'singular',
    'missing': 'absent',
    'image': 'img',
    'directory': 'dir',
    'message': 'msg',
    'generated': 'produced',
    'severed': 'cut',
    'received': 'got',
    'receiving': 'getting',
    'functional': 'working',
    'terminated': 'stopped',
    'fully': 'completely',
    'information': 'info',
    'attempting': 'trying',
    'further': 'additional',
    'imprecise': 'inaccurate',
    'usr': 'user',
    'configuration': 'config',
    'redundant': 'unnecessary',
    'start': 'initiate',
    'started': 'initiated',
    'starting': 'initiating',
    'disable': 'deactivate',
    'number': 'nbr'
}
```

Listing 7.2 BGL word changes to induce concept drift

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

Document name
MASTER'S THESIS

Date of issue
February 2024

Document Number
TFRT-6222

Author(s)
Adrian Murphy
Daniel Larsson

Supervisor
Ola Angelsmark, Advenica AB, Sweden
Fanny Söderlund, Advenica AB, Sweden
Johan Eker, Dept. of Automatic Control, Lund University, Sweden
Karl-Erik Årzén, Dept. of Automatic Control, Lund University, Sweden (examiner)

Title and subtitle
Towards Automated Log Message Embeddings for Anomaly Detection

Abstract

Log messages are implemented by developers to record important runtime information about a system. For that reason, system logs can provide insight into the state and health of a system and potentially be used to anticipate and discover errors. Manually inspecting these logs becomes impractical due to the high volume of messages generated by modern systems. Consequently, the research field of machine learning-based log anomaly detection has emerged to automatically identify irregularities. Parsing log messages into a structured, tractable format is a vital step in log anomaly detection. This degree project investigates the application of log message embeddings, a recently proposed log parsing method, for anomaly detection in complex IT systems and measures their resilience to concept drift, where the format of log messages changes over time, in comparison with a traditional parsing approach. Empirical analyses are conducted on two benchmark datasets, revealing that log message embeddings not only achieve anomaly detection results on par with traditional methods but also demonstrate considerable robustness against concept drift. A key focus of this project is on the application of large language models to automate the log embedding pipeline by handling out-of-vocabulary words and extracting synonymous and antonymous word relationships. These capabilities are important for distinguishing log messages that are identical except for one or more synonymous or antonymous word pairs. While large language models show promise in these tasks, experiments highlight the need for further refinement to match the performance achieved through manual operator feedback.

Keywords
IT System Monitoring, Log Anomaly Detection, Large Language Models, Log Message Embeddings, Concept Drift

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280-5316

ISBN

Language
English

Number of pages
1-64

Recipient's notes

Security classification

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