Technische Universität Berlin

Department of Telecommunication Systems
Distributed and Operating Systems

Fakultät IV
Marchstraße 23
10587 Berlin

Master Thesis

Anomaly Detection in Infrastructure-agnostic Cloud Environments

Harald Ott

Matriculation Number: 367884
August 21, 2020

Supervised by
Prof. Dr. Odej Kao

Assistant Supervisor
Sasho Nedelkoski and Alexander Acker

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbst- ständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

Berlin, August 21, 2020

. . . . . .

(Signature [Harald Ott])

Abstract

As modern cloud systems are becoming larger and increasingly complex, offering more computing power and services, they are becoming harder to manage and to maintain. Automatic and accurate detection of anomalies is crucial in order to ensure the reliability, security and safe operation of a cloud system. System logs are the primary source for troubleshooting. Recent studies focused on inflexible approaches, which are unable to learn semantically meaningful representations of logs, leading to an insufficient generalisation, which is necessary due to the evolution of logs through version updates and changing environments. This work proposes a model which utilises language models in order to capture the linguistic nature of system log data and extract log vector representations and use them in a Bi-LSTM model for anomaly detection. Several experiments on cloud system logs are performed, evaluating different language models which produce numerical log representations, including Bert, GPT-2 and XL-Transformers. Evaluations are performed on the robustness by injecting various alterations, thus simulating a new dataset. The results show that the log vector representations from language models can achieve a high performance and are robust to semantic alterations.

Zusammenfassung

Moderne Cloudsysteme, die immer mehr Rechenleistung und Services anbieten, werden immer größer und komplexer und sind dadurch schwerer zu warten. Die automatische und präzise Erkennung von Anomalien ist wichtig, um die Zuverlässigkeit, Sicherheit und den korrekten Betrieb eines Cloudsystems zu gewährleisten. Systemlogs sind die primäre Quelle bei der Fehlerdiagnose. Zahlreiche Studien verfolgen Lösungsansätze, die auf einer unflexiblen Darstellung von Logs basieren. Dies führt zu unzureichender Generalisierung, die jedoch aufgrund sich ständig verändernder Logs durch Updates und Umgebungsveränderungen nötig ist. Die vorliegende Arbeit stellt ein System vor, welches Sprachmodelle verwendet, um die natürlichsprachlichen Eigenschaften von Systemlogs zu erfassen und numerische Repräsentationen von ebendiesen als Eingabe eines Bi-LSTMs für Anomalieerkennung einzusetzen. Mehrere Experimente werden mit Cloudsystemlogs durchgeführt, um die Qualität der Sprachmodelle im Hinblick auf Anomalieerkennung zu evaluieren. Insbesondere wird die Robustheit gegenüber Veränderungen in normalen Logverläufen überprüft und die Übertragbarkeit der gelernten Eigenschaften eines Log-Datensatzes auf einen anderen Log-Datensatz simuliert. Es wird gezeigt, dass das Modell mit Hilfe der durch Sprachmodelle erzeugten numerischen Repräsentationen in der Lage ist, sehr gute Ergebnisse in der Anomalieerkennung zu erzielen.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

The Internet is permeating almost every aspect of modern human life. Large numbers of online services ease our ways of retrieving information, purchasing goods and staying in contact with each other. New opportunities for businesses emerge with the availability of reliable and easy to use public cloud infrastructures. These cloud systems are environments which make it possible to abstract and share distributed hardware resources, allowing the operation of vast numbers of multiple simulated environments on hardware systems. Virtualisation allows the separate, yet simultaneous allocation of resources for various services at once.

As these cloud systems are becoming increasingly complex, they are getting harder to maintain and to operate, with system failures, outages and other unwanted behaviour occurring on a regular basis. Detecting such failures is indispensable for the correct, safe and reliable operation of complex systems. The complete outage of the paying system of an E-Commerce shop for example, can potentially result in high losses in revenue and the disruption of user experience. The software which operates these cloud environments, like most software, produces log data during execution - text-strings which contain information about interactions between data, files, services and applications. Log files can be used to conduct failure and anomaly analysis, and to help understand the root causes of failures and errors. System operators would examine logs manually and determine if certain log events can be linked to a given system failure. At the scale of mentioned systems, analysing such log files manually is infeasible. It is therefore necessary to develop automised methods for this purpose. Naive approaches like matching certain keywords (e.g. "*error*"), constructing a set of log lines indicating anomalies or regular expressions are not adequate to capture the complex nature of anomalies. For example, errors can happen and can even be output as errors explicitly, but at the same time it can be normal behaviour of a system to then automatically recover from given errors, so triggering an alarm is not wanted in these cases [1]. There are several cases where even more complex environments can be on hand for anomaly detection. Typically, a reduction in the performance of an anomaly detection model can arise in two cases: when the system or its services are re-configured or updated in terms of operating system, library version or network, or when the model is trained on an existing system and later deployed on a new, similar system. New data and business characteristics arise and have to be met.

In any of these cases, the model has to be adapted dynamically to the new environment and settings. In both of the above mentioned cases, the models presented by previous works are not able to adapt properly, thus resulting in poor generalisation of the model.

Logs are constantly evolving due to the fact that developers are modifying source code frequently, including logging statements, thus leading to changing log data. Kabinna et al. [2] found that 20%-45% of logging statements in the studied open source applications change throughout their lifetime. Due to the constant evolution of logging data, the performance of existing anomaly detection approaches is significantly reduced. Existing approaches assume a constancy in logs, which is not realistic in real-world systems, where unstable log events containing changes on events and sequences appear constantly. Even small semantically insignificant changes to known log events would be identified as a completely different log event [3] [1] [4].

Therefore, existing approaches will either fail to work due to their incapacity to cope with unseen log events, or produce incorrect classification results. Additionally, normal log sequences are also likely to change due to new execution paths, processing noise or delays. It is infeasible to continuously update and retrain log-based anomaly detection tools due to the large amount of effort it requires [5]. Therefore, a more general approach is necessary which is robust to the changes mentioned above and can be transferable to new systems.

1.2 Scope

The availability of large amounts of log data have renewed interest in developing one-class deep learning methods to extract general patterns from non-anomalous log data. This work proposes to establish a connection between the latest advances in natural language processing and anomaly detection in log data. The need for using language models to transform log events into numerical vector embeddings arises from the weaknesses of older anomaly detection approaches which are not able to generalise well on new logs and cope with the challenges mentioned above. Difficulties exist in reusing previously obtained knowledge from a given dataset and transferring it to a differently structured dataset. This work provides a means to automatically detect anomalies in logs which are potentially affected by processing noise and changes of log events by updates of the underlying software and to transfer knowledge obtained from a dataset of log sequences to another dataset of logs with a transfer method.

In particular this includes the following points:

1. a model through which numerical log vector representations are obtained from general purpose language models and are utilised in a bi-directional long short-term memory (Bi-LSTM) neural network for anomaly detection,

2. two prediction modes, including a regression-based approach and a classification-based approach are proposed and evaluated,
3. experiments including scenarios to evaluate the robustness of the obtained log vector representations by altering log messages and sequences of log messages,
4. transfer of the method to another dataset by simulating the logs of an upgraded version of the system and fine-tuning of the model.

1.3 Outline

This master's thesis is divided into six chapters.

Chapter 2 presents background knowledge and terms required for the presented concept.

Chapter 3 presents the formal problem definition, the requirements and assumptions, followed by the developed concept. The complete framework is explained in detail, clarifying the necessity of every step in order to assemble the full model.

In **chapter 4**, the experimental set-up, followed by the results of the evaluation are presented. In doing so, a detailed evaluation of the used language models and a comparison between them is conducted.

Chapter 5 gives an overview over the current state of research, and describes three important contributions in detail.

In **chapter 6**, the final conclusion of the work is presented.

2 Background

This chapter summarises the technologies and terminologies that are most important for the proposed solution, and puts them into context.

In section 2.1 a description of the term *cloud computing* is given. In section 2.2 techniques to tackle the problem of anomaly detection in general are presented. In section 2.3 the theoretical foundation of deep neural networks is described, followed by an introduction to LSTMs. 2.4 gives insights on how natural language can be represented using language models, and how they can be useful in the context of anomaly detection in logs. Finally, in section 2.5 the importance and functionality of log parsers are explained.

2.1 Cloud Computing

The term *cloud computing* usually refers to hardware and system software in large data centres which provide a platform for applications delivered as services over the Internet. Due to the possibilities offered by clouds which are available in a pay-as-you-go manner, businesses with new ideas do not require enormous amounts of prefinancing for a hardly projectable amount of hardware and human operators to get their services online. It allows them to dynamically adapt to changing business needs without neither overcommitting hardware for services that do not turn out to be as intensely used as expected, nor undercommitting for a service that excels expectations, thus missing potential revenue due to not being able to cope with the demand [6].

Virtualisation plays a vital role in modern cloud systems, offering the possibility for numerous users and their applications to share infrastructure in parallel, in contrast to conventional hosting, as depicted in figure 2.1. Through virtualisation it is possible to achieve a better degree of utilisation of available hardware, for it allows a hardware server to run multiple software servers at the same time. Modern cloud services providers, like AWS, Microsoft Azure or Google Cloud, offer a vast number of different useful services, which can be provisioned flexibly and rapidly to the user, like allocation of task-specific hardware, different database types, object storage solutions or applications like analysis or management tools, as depicted in figure 2.2. End customers can be individual persons or businesses. While the services offered by public cloud providers are usually full-fledged and can be used by an end customer directly, the services offered by a cloud provider can also be integrated into a private cloud solution, thus resulting in a hybrid cloud,

where workloads can be dynamically shifted between private and public clouds as costs and computing capacity needs change.

Making sure that these large numbers of services and virtual machines are operating correctly is a challenging task for the cloud service providers. It is therefore vital to keep records of program executions in the form of system logs to be able to retrace errors that have occurred.

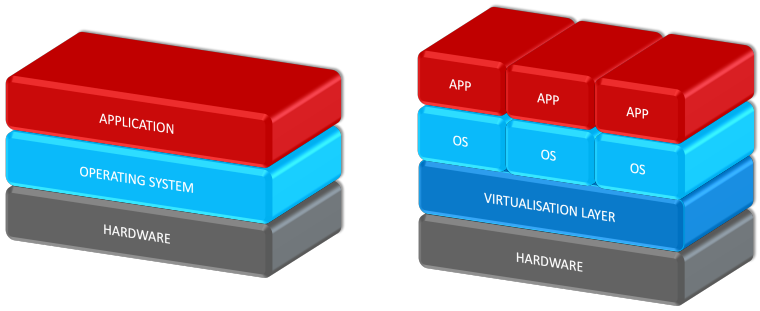


Figure 2.1: Traditional architecture vs. virtualised architecture

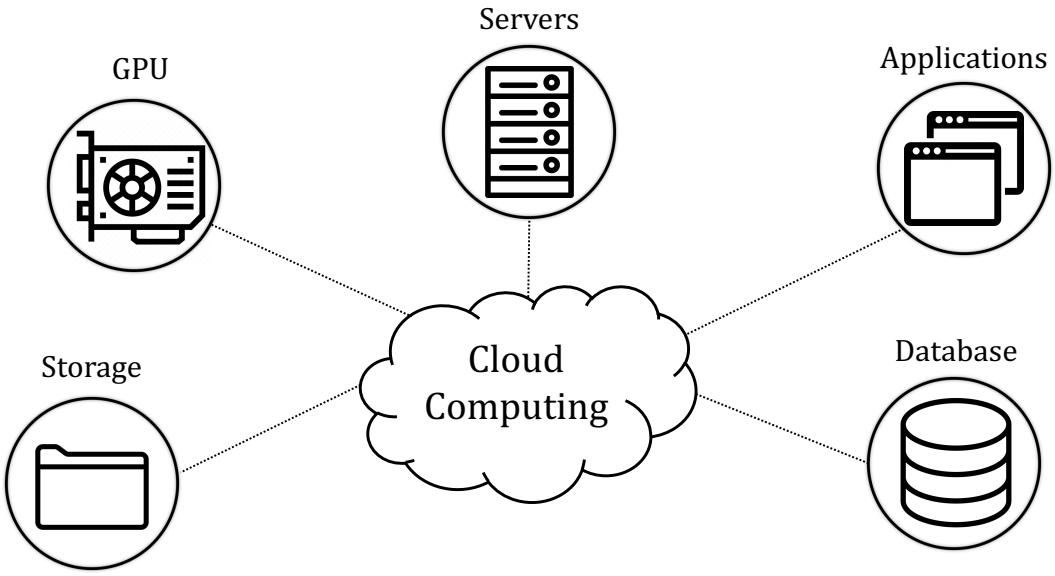


Figure 2.2: An overview showing the various services a cloud provider offers.

2.2 Anomaly Detection

In order to track down failures on the basis of system log data, a mechanism that is able to distinguish normal from anomalous system log patterns needs to be utilised. *Anomaly detection* describes the general problem of finding subsets or patterns in data,

that do not conform to a defined notion. These patterns are often referred to as outliers or anomalies.

Anomalies can arise in datasets for various reasons, like system errors, fraud or malicious activities. It often might appear to be straight-forward to define normal regions, and declare all data laying outside of these regions as anomalies. Unfortunately, finding these normal regions is in fact very difficult, since it might not always be possible to capture the nature of *normal* data in its completeness, due to lacking data or the unsharp border between normal and abnormal. Consider figure 2.4 which illustrates the presence of normal regions and anomalies in a dataset. The datapoints in the regions D_1 and D_2 are considered normal, since the majority of observations lie in these regions. Points whose values differ sufficiently from the values considered to be normal, like the points in regions A_1 and A_2, are considered anomalies. But consider also the points B_1. It is not clear offhand if they should be marked as normal or abnormal. It can also be doubtable, if the borders around D_1 and D_2 are correct. They might not be sufficiently broad, due to the lack of enough normal, i.e. non-anomalous training examples. Additionally to the difficulty of finding correct division of normal and abnormal, normal behaviour can be subject to constant evolution in a dynamic system, thus making previous definitions of *normal* behaviour wrong, obsolete or incomplete. Additionally, it is often hard or impossible to obtain labeled data for the desired domain, thus hindering the training and verification of a model [7].

Due to the difficulties arising from the aforementioned constraints, solutions to the problem of anomaly detection are usually very domain-specific, influenced by the form in which data is available, labeled or unlabeled and the form of anomalies which are to be detected. Solutions presented by researchers feature techniques from various fields, including data mining, statistics and machine learning, which are applied to the domain in question [7]. Figure 2.3 outlines the central steps involved in finding an appropriate anomaly detection technique to a given problem.

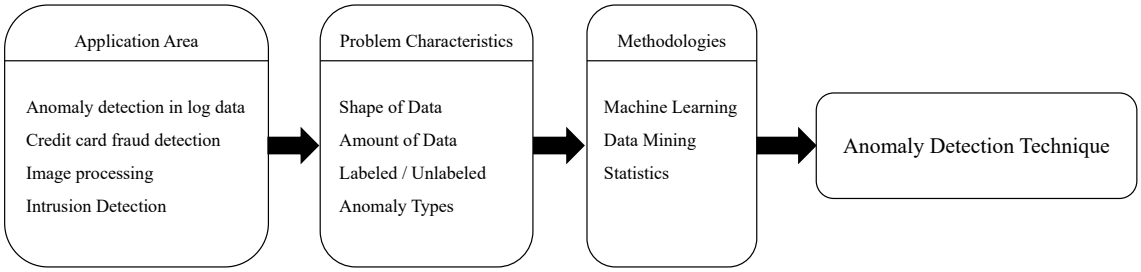


Figure 2.3: Schema of the development of an anomaly Detection technique [7].

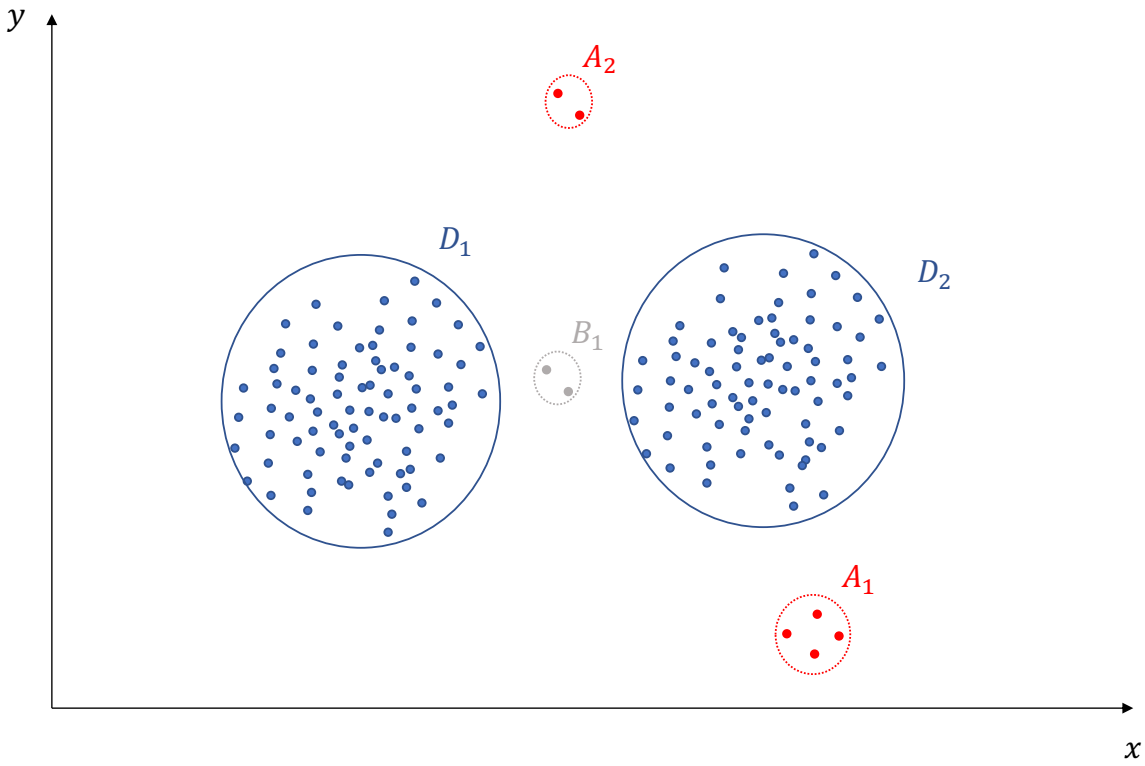


Figure 2.4: Example of anomalies in a dataset.

2.3 Deep Learning

Deep learning is a sub-category of machine learning that is able to achieve remarkable results in the area of anomaly detection by learning to represent data in specific data structures within a neural network. Deep learning can outperform traditional machine learning techniques as the amount of data increases [8]. Section 2.3.1 gives an introduction to neural networks, which are the computational basis of deep learning. Section 2.3.2 describes LSTMs, which is a deep learning technique that is specialised in handling time series data. Section 2.3.3 introduces Bi-LSTMs, an enhancement to standard LSTMs.

2.3.1 Neural Networks

Neural networks are systems that learn to compute a functional relationship between an input and an output by analysing training examples [10]. Figure 2.5 is an illustration of a classical feed-forward neural network. It consists of *neurons* and *weights* connecting the neurons. There are three types of neurons: each input value maps to an *input* neuron depicted as x_n. *Hidden* neurons, depicted as z_i are predictors created by mathematical

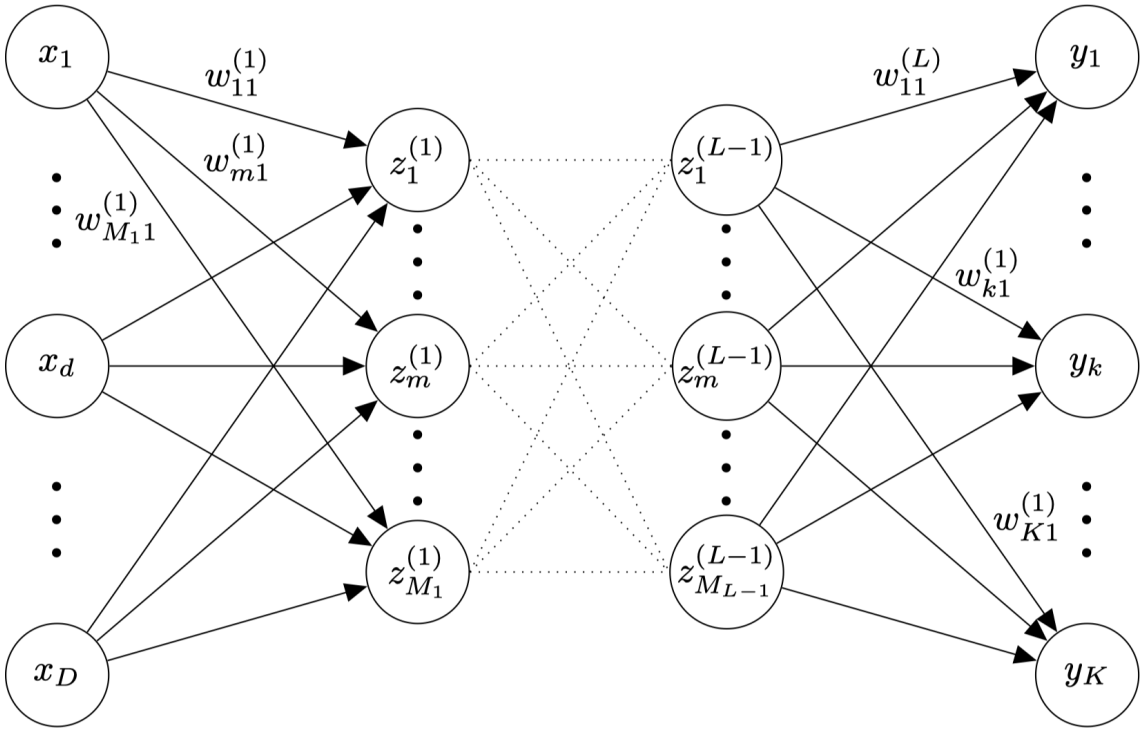


Figure 2.5: An illustration of a standard neural network [9].

functions and the *output* neurons, depicted as y_k, gather given predictions and compute the output [9]. Given data pairs $(x_1, t_1), ..., (x_N, t_N)$ with $x_n \in \mathbb{R}^D$ being input data, mapping to D input neurons, and $t_n \in \mathbb{R}^K$ being target data. Each component x_n is fed to one input neuron. If L is the number of layers, then there are $L-1$ hidden layers. The network's latent variables of the hidden neurons are denoted by $z_m^{(l)}$. When data is being forwarded through the network, with the goal of obtaining a prediction from the network, the following formulas are relevant and describe every step through the network. The result of a complete forward propagation is denoted by equation 2.1:

$$a_j^{(l)} = \sum_{i=1}^{M_{l-1}} w_{ji}^{(l)} z_i^{(l-1)} = \mathbf{w}_j^{(l)^T} \mathbf{z}^{(l-1)} \tag{2.1}$$

On the result of this computation, an *activation function* is applied as in equation 2.2. There are a variety of activation functions, depending on the use case. Activation functions transform the activation of output neurons as in equation 2.1 into an output signal [10]. For *regression* problems, the linear activation function $h(x) = x$ can be used. For *multi-class classification* problems, where t_n is a set of classes, the softmax function $\sigma(a)_j = \frac{e^{a_j}}{\sum_{k=1}^{K} e^{a_k}}$ is used [9].

There are more cases, but regression and multi-class classification are the most relevant

here, as they are used in chapter 3.

$$z_j^{(l)} = h_l(a_j^{(l)}) = h_l\left(\mathbf{w}_j^{(l)^T}\mathbf{z}^{(l-1)}\right) \tag{2.2}$$

If $l = 1$, then equations 2.3 and 2.4 hold:

$$a_j^{(1)} = \mathbf{w}_j^{(1)^T}\mathbf{x} \tag{2.3}$$

$$z_j^{(1)} = h_1\left(\mathbf{w}_j^{(1)^T}\mathbf{x}\right) \tag{2.4}$$

If $l = L$, then every y_k can be obtained in the following way:

$$y_k = h_L(a_k^{(L)}) = h_L\left(\mathbf{w}_k^{(L)^T}\mathbf{z}^{(L-1)}\right) \tag{2.5}$$

The result y_k of the computation of x_k is then compared to the real value t_k using an *error function*. Mean Squared Error $MSE = \frac{1}{n}\sum_{i=1}^{n}(t_k - y_k)^2$ can be used for regression problems, calculating the average squared difference between the estimated and the actual values, and Cross-Entropy $-\sum_{c=1}^{M} y_{o,c}\,\log(p_{o,c})$ for classification, calculating the separate loss for each class label per observation. The obtained values are then fed into the *back propagation algorithm*, updating weights w_j, thus trying to minimise the error.

2.3.2 LSTM networks

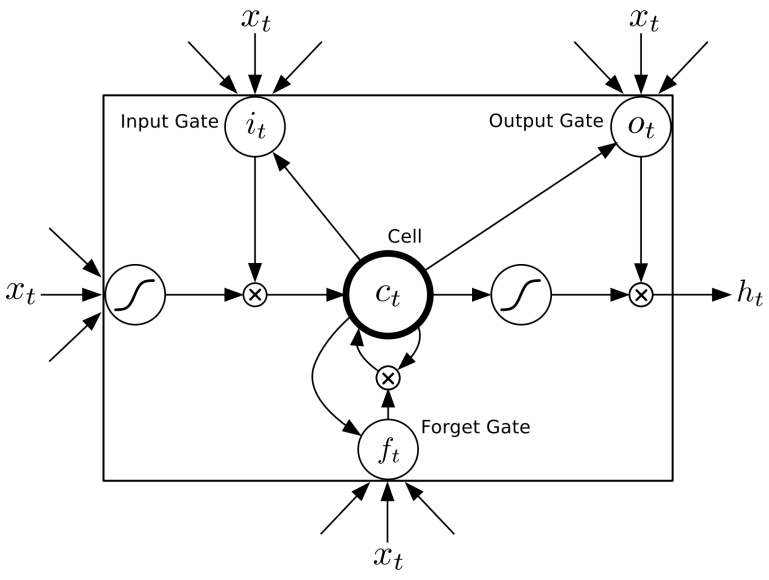


Figure 2.6: A LSTM block [11]

Recurrent neural networks (RNN) with Long Short-Term Memory (LSTM) are a widely-used effective model to tackle learning problems on sequential data. Being a

general model, they do not have be to be adapted to a specific problem like earlier methods, thus allowing them to produce state-of-the-art results for problems like language modeling [12] and anomaly detection [3], amongst others.

The core of the LSTM architecture is a memory cell which maintains its state over time, in combination with gating units, which control the information flow [13], allowing it to remove or add information. The traditional LSTM architecture was first described by [14]. The schematic structure of LSTM blocks is depicted in figure 2.6.

The first step is expressed through equation 2.6. It is a forget gate which considers the previous hidden state input h_{t-1} and the current input x_t, and outputs a vector of numbers between 0 (forget: value should be multiplied by 0) and 1 (keep: multiply value by 1), one for each element from the previous cell state c_{t-1}. Next, equation 2.7 computes which values to update. Equation 2.8 creates a vector of candidate values $\hat{c}_t$ for the new state c_t. In equation 2.9, the previous state is multiplied by f_t, in order to keep or forget elements, adding the new candidate values $\hat{c}_t$ multiplied by the degree updating i_t. As the last two steps, in equation 2.10, the output is computed by applying a sigmoid on the cell state in order to decide which parts to output. Then, in equation 2.11, a tanh is applied on the cell state c_t, which is multiplied by the output of the sigmoid gate o_t [15] [11].

$$f_t = \sigma(W_{xf}x_t + W_{hf}h_{t-1} + W_{cf}c_{t-1} + b_f) \tag{2.6}$$

$$i_t = \sigma(W_{xi}x_t + W_{hi}h_{t-1} + W_{ci}c_{t-1} + b_i) \tag{2.7}$$

$$\hat{c}_t = \tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c) \tag{2.8}$$

$$c_t = f_t c_{t-1} + i_t \hat{c}_t \tag{2.9}$$

$$o_t = \sigma(W_{xo}x_t + W_{ho}h_{t-1} + W_{co}c_t + b_o) \tag{2.10}$$

$$h_t = o_t \tanh(c_t) \tag{2.11}$$

2.3.3 Bidirectional LSTM networks

In a sequence prediction task in which one has access to both past and future input features for a given point in time, a bidirectional LSTM network as depicted in 2.7 can be applied as proposed by Graves et al. [11]. In this way, it is possible to make use of past features via the forward state and future features via the backward states. Forward and backward passes are executed similarly to the mechanism described in 2.3.2.

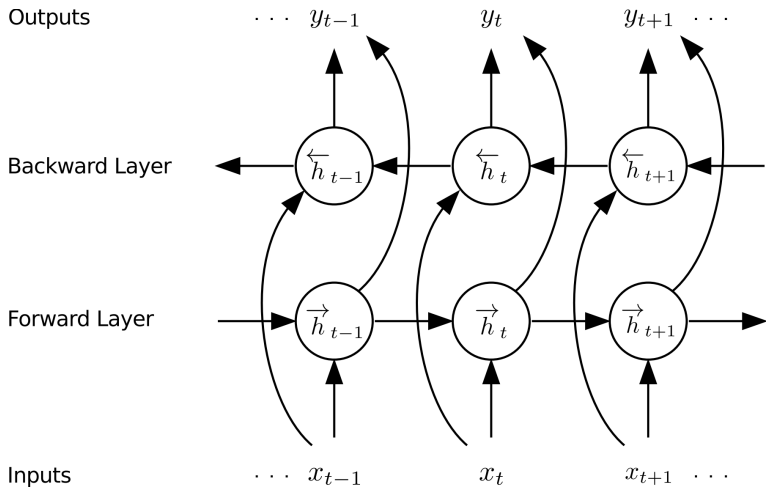


Figure 2.7: A Bi-directional LSTM [11].

2.4 Natural Language Processing

Since raw log data cannot be processed without further ado by a LSTM, finding a suitable representation of system logs is necessary. The fact that log data usually consists partly of natural language suggests to treat them as such. Section 2.4.1 gives a short general introduction to the concept of natural language processing, while section 2.4.2 describes the core idea of transforming natural language into a format which can be used for further analysis. Section 2.4.3 gives an overview of how Bert, a state-of-the-art language model, executes this transformation.

2.4.1 General concept

Natural language processing (NLP) involves the engineering of computational models to solve practical problems in understanding human languages. Having initially relied on processing involving statistics, probability and machine learning, the recent boost in available computational power with GPUs, allowed deep learning to raise the bar for many NLP-tasks. NLP can be broadly divided into two categories, namely *base concepts* which deal with the fundamentals of understanding language, and concrete applications by means of these very concepts, although the border between the two is often fluent. Base concepts include *language modelling*, *morphological processing* (identifying segments within words), *syntactic processing* (how do different words and phrases relate to each other within a sentence) and *semantic processing* (understanding the meaning of words), whereas applications include areas such as text translation, classification of documents, summarisation of texts, extraction of information and many more.

2.4.2 Word embeddings

Language modelling can be viewed as an essential piece of probably any practical application of NLP. Generally speaking, it involves creating a model to predict words given previous words by finding appropriate representations for words through analysing the relations of words within their context. Numeric vectors which represent single words obtained by language model techniques are called *word embeddings* [16]. For example, the word "Olympics" appears often in the context or close to words like "athlete", "running" or "tournament" but rather rarely next to words like "microphysics" or "chicken". These relationships can be translated into a vector that describes how the word "Olympics" is used within a language [17]. Word embeddings can be retreived either by Principle Component Analysis or by using deep neural network models and capturing their internal states.

2.4.3 Bert

The functionality of a language representation model will be outlined on the basis of Bert (**B**idirectional **E**ncoder **R**epresentations from **T**ransformers) by Devlin et al. [18]. The model architecture is a multi-layer bidirectional Transformer encoder. The encoder maps an input sequence of symbol representations to a sequence of continuous representations. The Transformer architecture introduces self-attention and fully connected layers on top of the encoder structure, which has been shown to be superior in quality and is significantly cheaper to train than previously proposed models [19].

Training includes two steps: *pre-training* and *fine-tuning*. Pre-training involves training on unlabelled data over various pre-training tasks. For finetuning, the weights initialised with the parameters obtained from pre-training are recalibrated on labelled data.

For pre-training, they extract sentences from a large unlabelled corpus like English Wikipedia. The obtained sentences are then transformed into tokens, and separated by pre-defined separation symbols, `[CLS]` for the beginning of a sentence, `[SEP]` for the ending of sentences as illustrated in figure 2.8. They then proceed with the first task, namely Masked Language Model (MLM). For this purpose, they mask 15% of words with the special `[MASK]` token, and then predict these tokens. The second task is Next Sentence Prediction (NSP), where sentences `A` and `B`, which are separated with the aforementioned `[SEP]` token, as it can be again seen in figure 2.8, are marked with the label `isNext` if `B` follows `A` or `notNext` if the following sentence is a random sentence, with both cases occurring 50% of the time. After the computationally expensive pre-training is done, taking 4 days of training on 16 cloud TPUs for one language [20], fine-tuning can be done on any downstream NLP task in at most one hour on one cloud TPU, for example the Stanford Question Answering Dataset (SQuAD v1.1) by Rajpurkar et al. [21], a collection of 100k crowd-sourced question/answer pairs, or the

General Language Understanding Evaluation (GLUE) benchmark by Wang et al. [22] which involves various natural language understanding tasks.

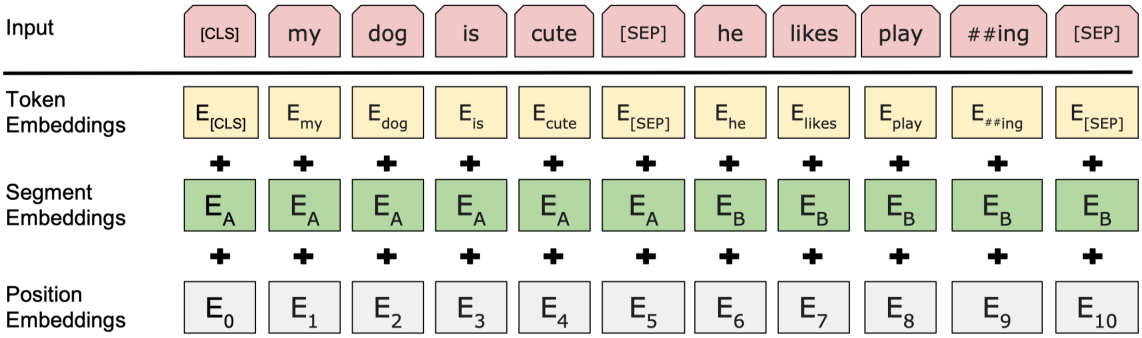


Figure 2.8: BERT input representation. The input embeddings are the sum of the token embeddings, the segmentation embeddings and the position embeddings [18].

2.5 Log Parsing

Due to the unstructured nature of system log data, a log parser has to be utilised on the logs, a crucial step in order to be able to transform them into word embeddings as described in 2.4. Raw log messages consist of a constant and a variable part, with the constant part remaining identical for every occurrence, while the variable part records runtime information and varies among different event occurrences. The goal of log parsing is to separate the *constant* and *variable* part of a raw log message [23][24]. Figure 2.9 shows how logging statements from Java source code are parsed. The logging statements variable parts `block`, `block.getNumBytes()` and `inAddr` are dynamically interpreted at runtime and replaced by their respective values. The resulting log message is then printed with additional customisable values (timestamp, logging level and component) by the respective logging framework. The structured log can then be parsed by the log parser, separating the constant part, which is also called a *template* (`Received block <*> of size <*> from /<*>`), from the variable parts (`blk_-562725280853087685, 67108864` and `10.251.91.84`), replacing the variable parts inside the constant parts with a pre-defined token - "`<*>`" in this example.

Log parsing is usually the first step in order to perform a log analysis task. Log parsing enables enables searching, filtering, grouping and mining of logs. Applications include usage analysis at Twitter [25] or workload modelling [26]. Logs can also be used as data sources for performance modelling [27] where performance improvements of a system can be validated using log data. A very prominent application of log parsing is anomaly detection. Since logs record execution information of a system, they are a valuable data source for identifying abnormal behaviour of a system [24].

There exist offline and online log parsers, offline meaning that it first reads and analyses the whole dataset first before applying the parsing model, while online parsers adjusts the parsing model gradually during the parsing process [28]. Log parsers employ various concepts and techniques in order to parse logs - a few of them are summarised here:

- *Frequent Pattern Mining* involves finding sets of patterns, in this case templates, that appear frequently in a data set. The procedure can be outline as follows: Iterating over the log data several times, while building frequent sets of tokens, followed by grouping log messages in clusters, and then extracting event templates from each of the clusters.
- Log parsing can be viewed as a *clustering* problem. All approaches can be roughly outlined as clustering templates hierarchically based on a defined metric, for example the weighted edit distance between pairwise log messages [24].
- Some proposed methods utilise special heuristics, exploiting the unique characteristics of log messages. IPLoM first identifies frequent words occurring more frequently than a threshold value, then extracts combinations of these words that occur in each line in the data set, marking them as cluster candidates, and finally selecting the candidates the occur more often than a threshold value as clusters [29]. Drain employs a parse tree with fixed depth. It first preprocesses the incoming messages with regular expressions based on domain knowledge, then search a log group for that message, with log groups being leaf nodes. If a suitable group is found, it matches the message to that group, if not, then a new group is created [28].

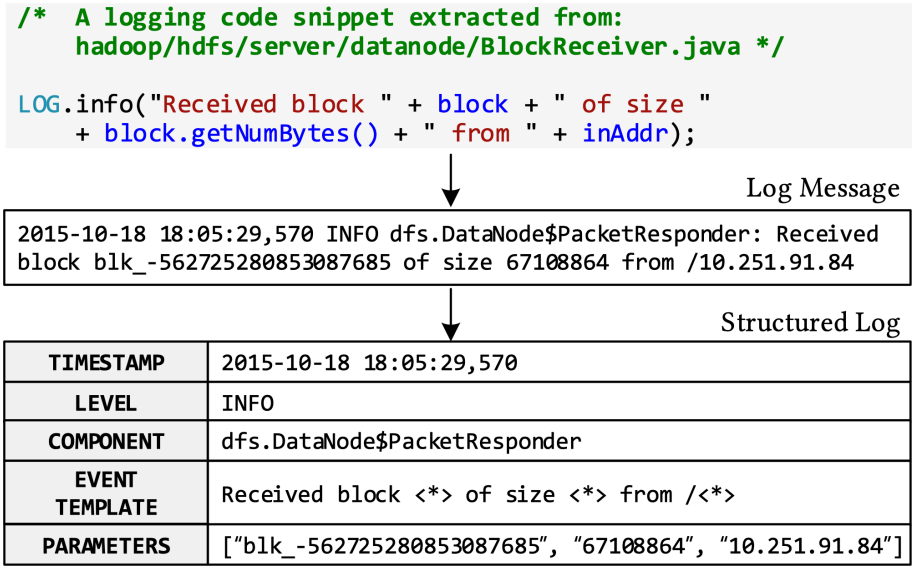


Figure 2.9: Schematic execution of log parsing [24]

3 Concept

Establishing a connection between the latest advances in NLP and anomaly detection in system log data is a recently emerged field in research. The need for using language models to convert log events into word embeddings emerges from the weaknesses of older anomaly detection approaches that are not flexible enough to cope with the complex nature of system log data. There exist difficulties in reusing previously obtained knowledge from training on data from a given dataset and transferring it to a differently structured dataset. Most proposed techniques for solving the problem of anomaly detection in system log data suffer from not being transferable and being non-resilient to changing log data. The objective of this work is to provide a means to automatically detect anomalies in logs that are potentially affected by processing noise and changes of log events by updates of the underlying software, and to transfer knowledge obtained from a dataset of log sequences to another dataset of logs with a *transfer of knowledge* method.

In section 3.1, the general problem statement is defined, and necessary requirements for the model are specified. In the following section 3.2, the overall system architecture with its components is outlined and visualised. In section 3.3 the necessary pre-processing steps for preparing the raw log messages for the anomaly prediction model are described in detail. In section 3.4 the developed prediction approaches are described in detail. There exist two approaches, the regression and the classification approach, both using log event representations obtained by a language model. In section 3.5 the transfer of knowledge mechanism is described.

3.1 Problem Statement and Prerequisites

Logs are print statements inside programs which are defined by a fixed sequence of code statements written by developers. The execution of a program is defined by these statements, and therefore follows a predetermined pattern. Hence, the produced logs must also follow certain patterns, chronological orders, and proportional relationships between number of occurrences of logs with each other.

3.1.1 Formal problem definition

We define a system log L as a sequence of log events or log messages $(m_0, m_1, ..., m_n)$ generated by logging instructions (e.g. *printf()* or *log.trace()*) within the software source

code during the execution of the program. Log messages consist of a constant (log template) and a variable part. Log messages consist of tokens - most tokens are English words, but do also include special characters. Each log message consists of a bounded sequence of tokens, $\mathbf{z_i} = (z_j : w \in \mathbb{Z}, j = 1, 2, ..., |\mathbf{z_i}|)$, where $\mathbb{Z}$ is a set of all tokens, j is the positional index of a token within the log message m_i, and $|\mathbf{z_i}|$ is the total number of tokens in m_i. For different m_i, $|\mathbf{z_i}|$ can vary. Depending on the concrete tokenisation method, t can be a word, word piece, or character. Therefore, tokenisation is defined as a transformation function $\mathcal{T} : m_i \rightarrow \mathbf{t_i}, \forall i$.

We additionally introduce the notion a numerical vector representation (embedding vector). An embedding vector e_i is a d-dimensional real valued vector representation of a log message template t_i. The embedding vector e_i is obtained from a pre-trained language model $M(t_i)$.

Classification: For every distinct log template t_i, we assign a class c_i. Let $g(E_i, \Psi) : |s| \times \mathbb{R}^w \rightarrow \mathbb{R}^k$, be a function computed by a neural network, that given a subsequence of s vectors out of the dataset E, namely $E_i = (e_i, ..., e_{i+s})$, returns a probability distribution $Pr_i[e_{i+s+1}|E_i] = (c_0 : p_0, c_1 : p_1, ..., c_k : p_k)$, describing the probability for each template class from C to appear as the next class at index $i + s + 1$, given E_i. The objective is to learn the parameters Ψ, so that for each fixed length sequence of vectors, the function predicts the correct set of possible subsequent classes. If one of the top q candidate classes matches the actual class, then $\hat{b_i} = 0$ is returned, if none match the actual class, $\hat{b_i} = 1$ is returned.

Regression: Let $h(E_i, \Phi) : |s| \times \mathbb{R}^w \rightarrow \mathbb{R}^w$ be a function computed by a neural network, that based on a sequence of s vectors $E_i = (e_i, ..., e_{i+s})$ predicts the vector d at index $i + s + 1$. The objective in this case is to learn parameters Φ, so that the system predicts the vector following the sequence of vectors of length s. If the distance between the predicted vector d and the actual vector e_{i+s+1} is above or below threshold values, which will be computed by the q-th percentile of all loss values from the original dataset, then $\hat{b}_{i+s+1} = 1$ is returned, if it is inside these thresholds, then $\hat{b}_{i+s+1} = 0$ is returned.

3.1.2 Requirements and Assumptions

1. The model requires word vectors that are computed by a language model that has been pre-trained on a sufficiently large corpus.

2. The model requires *normal*, non-anomalous log data which do not contain anomalies for training.

3. The model is not able to detect anomalies which are only detectable in the variable part of the log messages. The model only considers the templates, i.e. keys of the log message.

3.2 System Overview

In this section, a broad overview of the overall system is presented, with figure 3.1 illustrating the steps necessary for the learning procedure, followed by anomaly prediction. The core concept can be outlined as follows: first, the original log sequences are ordered, then a log parser is used to extract templates t_i from the original log sequences $(m_0, m_1, ..., m_n)$ and then transform the log sequences to template sequences $(t_0, t_1, ..., t_n)$. Afterwards, the template sequences are transformed into log sequence embeddings $(e_0, e_1, ..., e_n)$ by a language representation model M. This procedure is described in detail in section 3.3.

For the training part, sub-sequences of log embeddings $E_i = (e_i, ..., e_{i+s})$ are fed into a Bi-LSTM that learns to predict the next embedding e_{i+s+1}, which is called the regression task, specified in section 3.4.3 and coloured red in figure 3.1. For the classification-based approach, the next template class c_{i+s+1} is predicted, specified in section 3.4.2 and coloured blue in figure 3.1. The results of these predictions are then compared to the true subsequent log embedding or class of the true subsequent log template, in order to train the model to identify non-anomalous log data.

The prediction part involves two steps: first, the template sequences obtained from the original log sequences A are altered by changing the order of the sequences or manipulating the templates, as described in section 3.3.5. As a second step, these manipulated sequences are fed to the model that has been trained on embedding sequences not containing any alterations, thereby obtaining the model's predictions $\hat{b}$.

Transfer of knowledge builds onto this process. After a model has been trained as described on dataset A, pre-processing is conducted the same way on a new dataset B, thus obtaining a sequences of embeddings. For the classification task, the template sequences of dataset B are mapped to the class mappings of dataset A, and then used for prediction. The regression task functions analogously to learning without the transfer mechanism. Transfer of knowledge is described in detail in section 3.5

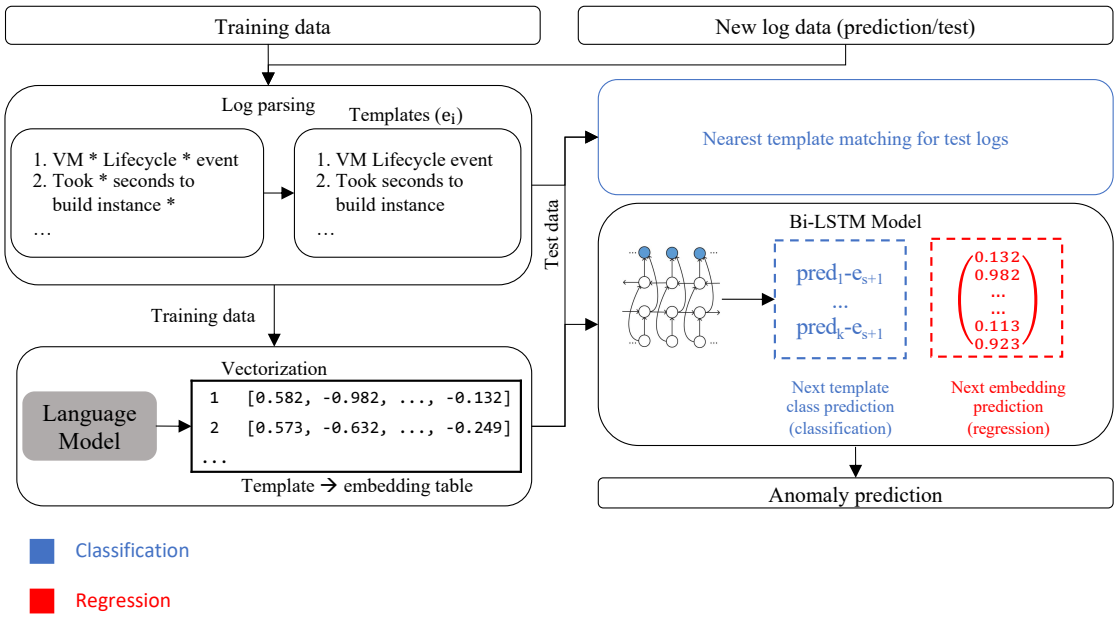


Figure 3.1: Anomaly Detection System

3.3 Pre processing

Log files are usually available in an unordered, raw state, and need to be ordered, parsed and transformed into an appropriate format so that they can be handled as sequences by a Bi-LSTM. The required steps for this purpose are outlined in this section. In the first subsection 3.3.1, the steps required for parsing logs in order to obtain log templates t_i are outlined, followed by the transformation into word embedding vectors e_i in 3.3.3. A diagram illustrating the complete pre-processing pipeline can be see in figure 3.9.

3.3.1 Log Parsing

Log parsing is an important step for automated anomaly detection, as already described in section 2.5, since raw log messages are unstructured data and contain a lot of extra information. For this work, the log parser Drain [28] is utilised. It is characterised by high accuracy and efficiency [28] and achieves the best results compared with other related methods, evaluated in [24]. The detailed result of the execution of a log parser can be see in figure 2.9. There are a few important aspects to note: Not only does the log parsing step extract log templates, but it also extracts other valuable information in a structured way, namely timestamps and, in this example the bulk id. Timestamps are needed, in order to make sure that the logs are in the correct chronological order, since it is possible, that system logs are an aggregation of log output of different sub-routines or different instances, which can happen concurrently, thus producing unordered logs.

Additionally to sorting by timestamps, it can also be required to sort system logs by group ids, instance ids or bulk ids as it can be seen in step 1) in figure 3.9. Matching instance ids are identified, made visible by marking same instance ids yellow. They are then separated in order to be able to observe each self-contained block separately from other blocks. After these sorting procedures, the result of the log parser's transformation $\mathcal{T} : m_i \rightarrow \mathbf{t_i}, \forall i$ from log messages to templates are visible as step 2) in figure 3.9.

3.3.2 Template cleansing

Even though log parsing and the aforementioned ordering steps largely improve the further processability of logs for sequential learning, by making it possible to single out the fundamental semantics of a log event, they are still partly made up of special characters and variable names. The following characters are removed:

1. All non-character tokens such as delimiters, digits, and particularly variable placeholders (`<*>`).
2. All concatenations of words are split, for example `sync_power_state` will be split into the separate words `sync`, `power` and `state`
3. All leading, trailing and repeated whitespace characters are removed.

This part of the pre-processing is marked with step 3) in figure 3.9.

3.3.3 Word vectorisation

After the aforementioned pre-processing steps, the log events are transformed into word embeddings, using a language model that is able to convert words or whole sentences into word or sentence embedding, effectively representing function $M(t_i)$ defined in 3.1.1. The process of transforming a sequences of logs with the API of a language model is visualised in figure 3.2.

Satisfying the following two requirements is essential in the context of providing suitable word embeddings for the anomaly detection task:

- Distinguishability: Word embeddings should capture the difference between log events with differing semantics. For example "`<*> Terminating instance`" and "`<*> Deleting instance files <*>`" are log events with highly different semantics, even though they contain equal (instance) and in the broader sense similar (terminating, deleting) words. This means their cosine distance should be high.
- Tolerance: Word embeddings should capture the similarity between different log events with same or very similar semantics. For example, the log event pair "`<*> Creating image`" and "`<*> Image created successfully`" or "`VM up`" and "`VM started`". This in turn should result in a low cosine distance [5].

In oder to be able to compare the quality of different language model's word embeddings with regards to the task of anomaly detection, it is possible to easily swap the used language model for log event transformation. For this work, the pre-trained language models used for evaluation are Bert, GPT-2 and XL-Transformers.

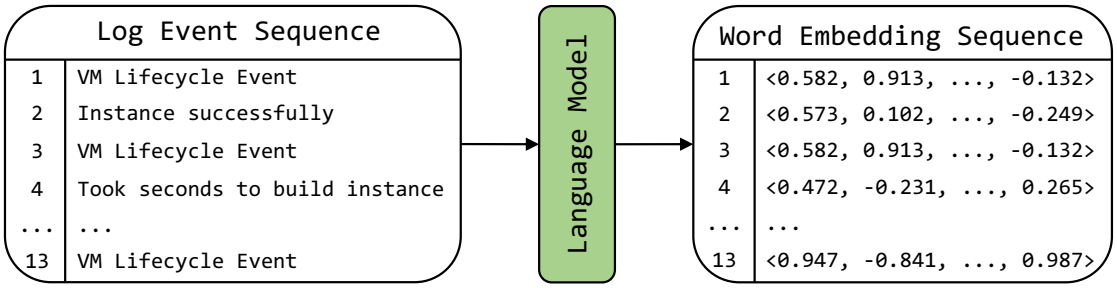


Figure 3.2: Transformation of a log event sequence to a word embeddings sequence.

3.3.4 Finetuning

Word embeddings models are usually provided pre-trained in different formats (e.g. with or without upper case), because training them from scratch is expensive - for Bert, it requires 4 days of training on 16 cloud TPUs for one language [20]. They are trained on large corpuses with unsupervised tasks. In order to make them more useful with regards to the task of anomaly detection, it is reasonable, to adjust the given pre-trained datasets using the log data corpora, by using the finetuning API provided by the language models. For this purpose, the log corpus on which finetuning shall be executed is prepared for the finetuning task of Masked Language Model, as described in subsection 2.4.3.

3.3.5 Log Alteration

By altering log events the evolution and instability of log events is being simulated. Since software is changed by developers, also log statements are subject to constant change. It is therefore necessary to build a flexible model that does not have to be retrained completely after each update of a log producing software. Log alteration is also used to simulate a different dataset B based on a given dataset A, for the evaluation of the model on drastically changed logs that appear like from another dataset, as outlined in 3.5.

Anomalies and alterations can be injected at arbitrary ratios. Various types of alterations are injected into original log data, either on the log event itself as depicted in figure 3.3 or on the sequence of log events as visualised in figure 3.4.

Three types of alterations are injected into the log events: a various amount of words is inserted between the tokens, for example words that appear in the context of logs, like "*deleted*", "*during*", "*for*" or "*time*". Words can also be also deleted or replaced.

It is not expected that the system detects a log event as an anomaly, that has not been changed much, i.e. only one word has been added into a statement (e.g. if "** Took *.* seconds to deallocate network for instance.*" has been changed to "** Took *.* seconds to deallocate network for this instance.*").

Additionally, it is possible to perform changes on the sequence of logs. In the following example, let $M = (m_i : i = 0, 1, 2, ..., n))$ be a sequence of log events:

- events can be *deleted* from the sequence, meaning that if the event at index j is selected for deletion, the resulting sequence is $M_{del} = (m_0, ..., m_{j-1}, m_{j+1}, m_n)$.
- events can be *shuffled*, meaning that if the event index j is selected for shuffling at index k, the resulting sequence is $M_s = (m_0, ..., m_j, m_k, m_{k+1}, ..., m_{j-1}, m_{j+1}, ..., m_n)$
- events can be *duplicated*, meaning that if the event at index j is selected for duplication, the resulting sequence is $M_{dup} = (m_0, ..., m_j, m_j, m_{j+1}, ..., m_n)$

Similar to the insertion of minimal alterations on the log events themselves, it is expected of the system not to detect an anomaly for the deletion, duplication or shuffling of events, since these also can reflect minor changes through new valid execution paths due to version updates and new deployments.

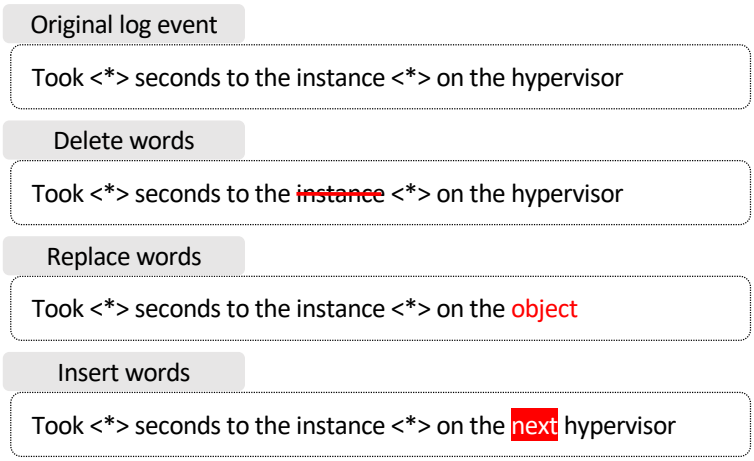


Figure 3.3: Altering log events.

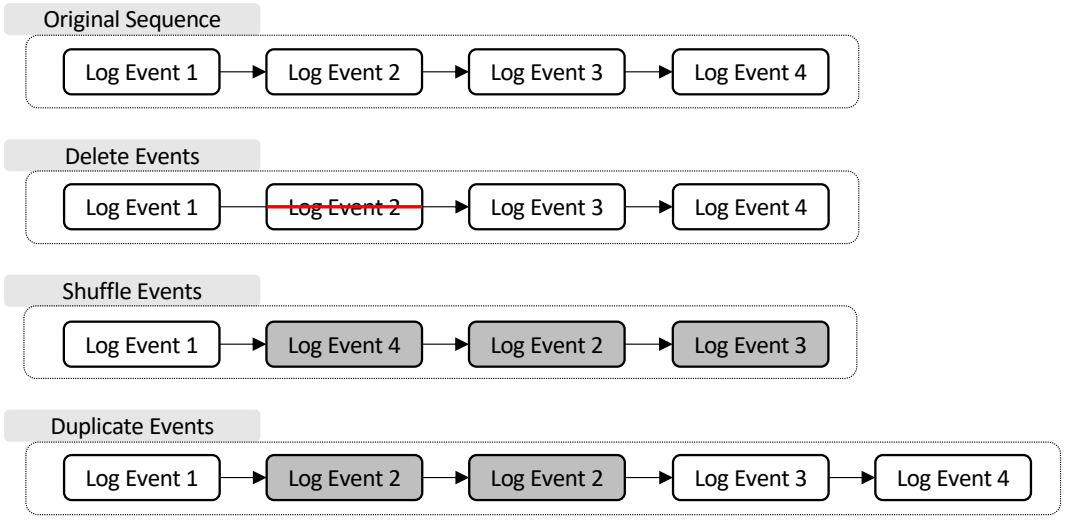


Figure 3.4: Altering log sequences.

3.4 Prediction Model

3.4.1 LSTM Model

Through the aforementioned steps in 3.3, the system log L has been transformed into a sequence of word embeddings $e_i \in \mathbb{R}^w$. In order to learn sequences of logs, a sequence of s log embeddings are concatenated to form an embedding sequence $E_i = (e_i, ..., e_{i+s})$. Taking a sequence of embeddings as input, a *Bi-LSTM* neural network as described in 2.3.3 is utilised to predict the class or embedding at position $i + s + 1$. Figure 3.5 shows the structure of the Bi-LSTM. As a first step a dropout layer is applied to the input sequence which randomly drops out information in order to reduce overfitting and improve generalisation, before feeding it to the forward and backward layers of the Bi-LSTM. This way, more information of the input log sequences can be captured than when only an uni-directional LSTM would be utilised. Then, the outputs of the Bi-LSTM are again fed into a dropout layer, followed by a fully connected layer which reduces the output of the LSTM to the desired dimensions, i.e. $\mathbb{R}^w$ for regression and number of classes c for classification. Finally, an activation function *act* is applied to the last output o_{i+s+1}, *log-softmax* for classification and *linear* for regression, respectively, in order to obtain the model's prediction p_{i+s+1}.

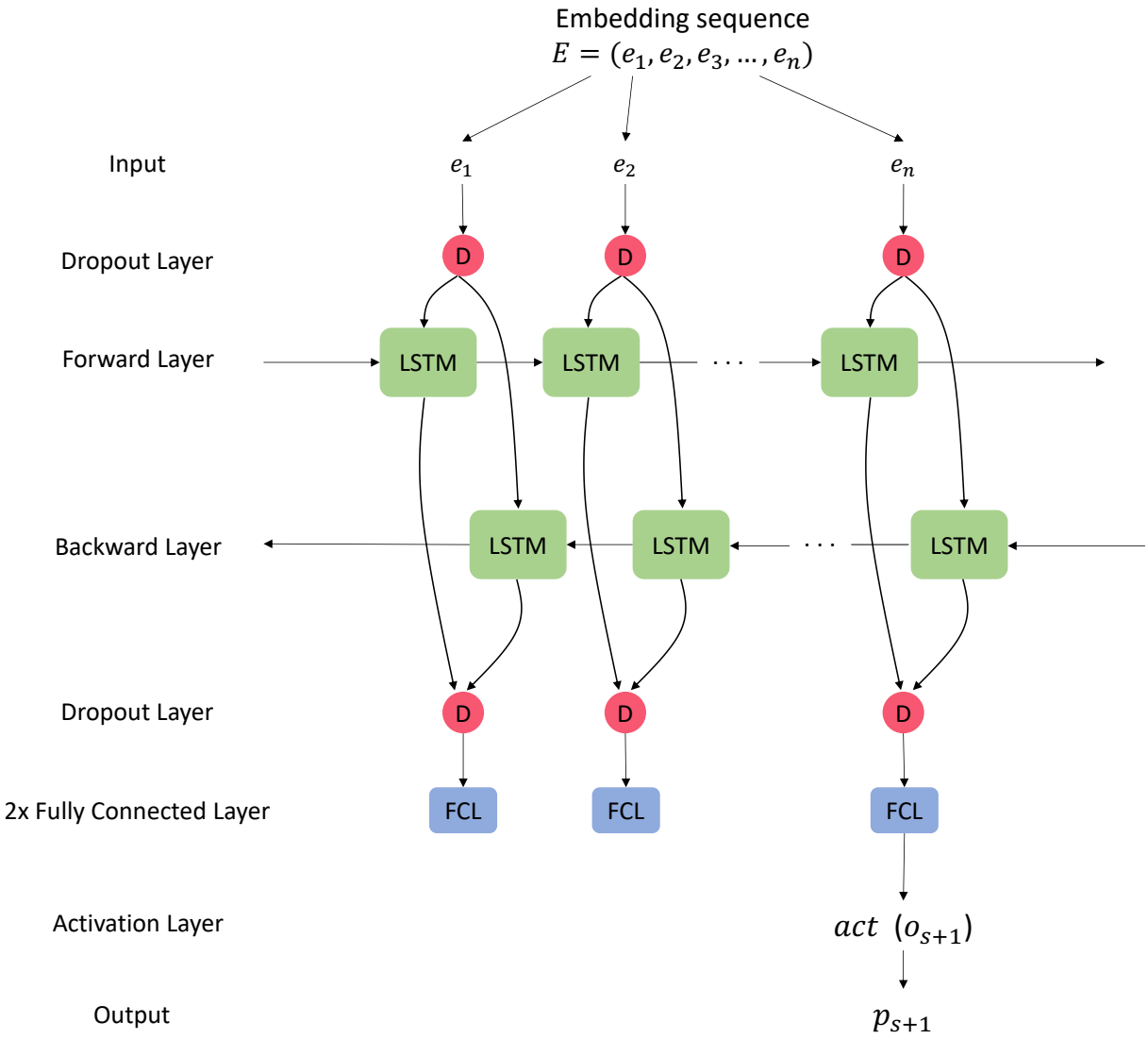


Figure 3.5: Bi-LSTM model

3.4.2 Classification

For the classification approach, the finite set of n unique log event templates is mapped to class indices $c_0, ..., c_n$. Training of the neural network is then performed on original log data that does not contain anomalies.

- The *input* values of the training data have the dimensions n, s and w, with n being the number of unique log events, s being the length of the sequence of word embeddings for which the neural network shall learn the consequent class and w being the dimensionality of the log event embeddings.
- The *target* values of the training data are structured as follows: for every sequence of word embeddings $E_i = (i, ..., i + s)$, there is a corresponding class c_{s+i+1} that stands for the template at position $s + i + 1$. The system is trained to predict that class correctly.

After training has been executed on the *train* dataset, the prediction phase starts, where a *test* dataset containing anomalies and alterations, as described in 3.3.5, can be processed by the neural network. For a new incoming test dataset, the system first has to match every template to the template classes of the train dataset. For every template in the test dataset, the nearest neighbour out of the templates of the train dataset is determined and will get the respective class assigned, as depicted in figure 3.6. This means in particular, that for every unique template, the corresponding word embedding is retrieved, and then every one of the word embeddings of the test dataset is mapped to the word embedding from the train dataset with the lowest cosine distance. Additionally, a threshold has to be found, so that if for a given embedding in the test data, no corresponding embedding with a cosine distance below this threshold is found, that template shall not get a class assigned, this would otherwise lead to a situation where the log event gets mapped to any of the template classes, which is not wanted behaviour.

After the matching process is finished, the system can read in the sequences of log events of the test data. For every sequence of log events E_i of length s, the system returns a probability distribution $Pr[c_{s+i+1}|E_i] = \{c_0 : p_o, c_1 : p_1, ..., c_n : p_n\}$ that denotes the probability of each log template class to appear as the subsequent one. It is possible, that there are multiple candidates for the following template. Let's assume, that the system is attempting to terminate an instance, then the corresponding template to class c_{s+i+1} could be for example '*Instance terminated successfully*' or '*Waiting for instance to terminate*'. The possible template classes c_i are sorted based on their probabilities. A predicted template class is considered normal, resulting in the system's prediction $\hat{b}_{s+i+1} = 0$, if it is among the top q candidates. It is marked as anomalous, $\hat{b}_{s+i+1} = 1$ otherwise.

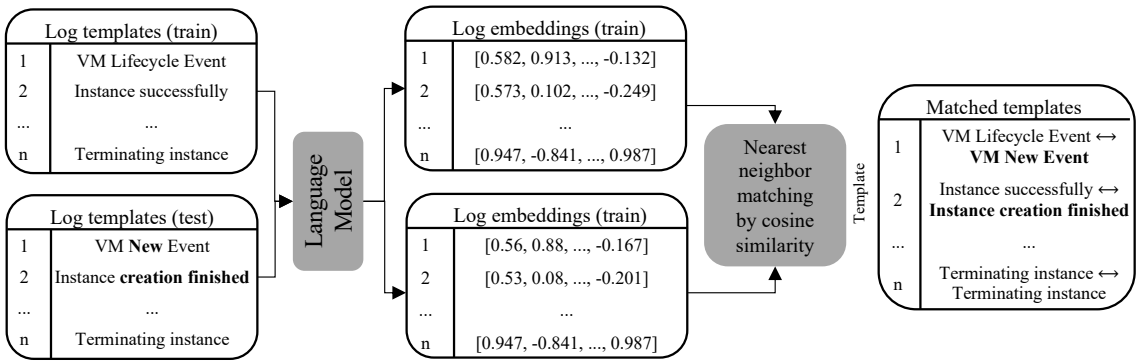


Figure 3.6: Template mapping

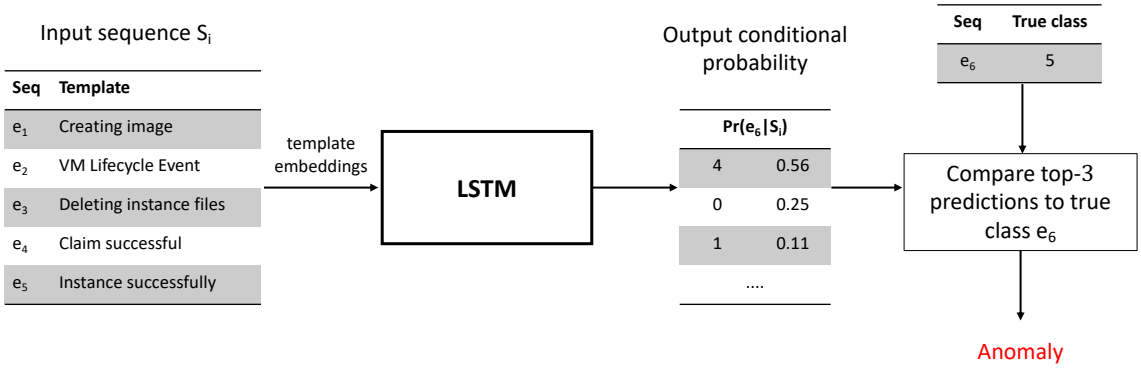


Figure 3.7: Class Prediction example for $g = 3$

3.4.3 Regression

For the regression approach, the neural network is trained to solve a regression task, with the input values of the training data being structured as described in 3.4.2, while the corresponding target value for the sequence of embeddings $E_i = (e_i, ..., e_{i+s})$ is the embedding of the log event at position $i+s+1$, meaning the neural network shall predict the next embedding e_{i+s+1}. After training on the original dataset, the mean squared error loss of every target word vector at position $i + s + 1$ of the corresponding input sequence E_i, and the neural network's predicted word vector, is computed. Afterwards, the *q-th* percentile of the agglomerated loss values of the original dataset is computed. The optimal value q for the following purpose is to be determined by performing a grid-search. For the sequences of the test dataset, the loss values are computed in the same way as for the original dataset. This is depicted in figure 3.8. The system will then mark every log event at position i whose word embedding's loss value is above the calculated *q-th* percentile as an anomaly, with $\hat{b}_i = 1$, otherwise as non anomalous, with $\hat{b}_i = 0$.

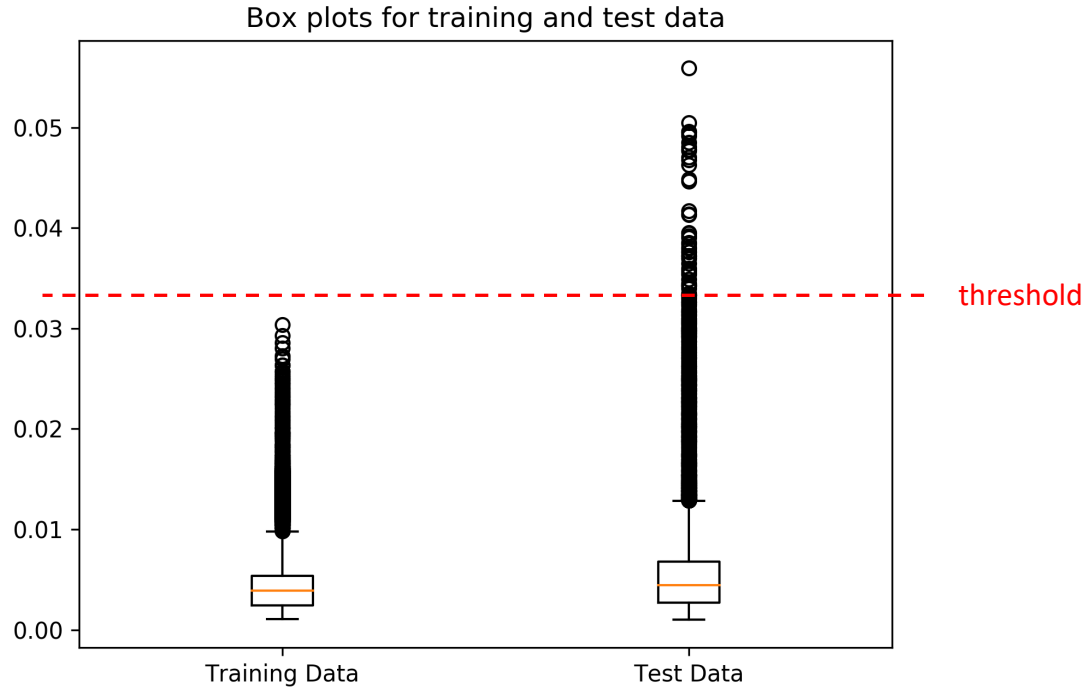


Figure 3.8: Box plots showing the distribution of loss values for training and test data.

3.5 Transfer of Knowledge

As already mentioned in chapter 1, modern cloud systems and their underlying software are subject to constant change. Through continuous re-deployment and updates of services and systems, there is a lack of sufficient data to perform training of a new anomaly detection model. Therefore, using pre-trained general purpose language models for extracting vector representations of logs and re-usage of already pre-initialised weights of a Bi-LSTM model from training on older log version allows transferring the model to new unseen logs. Let dataset A be the training dataset from already known log messages, and dataset B be a log dataset from an updated version of the system. The steps for prediction of unseen logs using the classification approach are outlined in subsection 3.5.1 and for the regression approach in subsection 3.5.2.

3.5.1 Classification

For the classification approach, the model is first trained as depicted in the pre-processing and training part of figure 3.1 on a train dataset A. Then, in order to re-use the trained model, several preliminary steps are executed. First, every log event of a training dataset B is mapped to the nearest neighbour of train dataset A, i.e. the word embedding with the shortest cosine distance, and gets assigned the same class. This is the same procedure as described in 3.4.2 and depicted in 3.6, with the only difference that there does not exist a threshold for assigning a class. Every log event will get a class assigned. Then, few-shot training on the training dataset B will be executed, in order to adjust the model to the new dataset B. As a final step, with the adjusted model on training dataset B, the prediction phase on a test dataset B can be executed, completely analogous to the description in 3.4.2 and as depicted in 3.1.

3.5.2 Regression

For transfer of knowledge using the regression-based approach, the model is first trained as depicted in the pre-processing and training part of figure 3.1 on a train dataset A. Then, the same weights of the LSTM that have been learned from this training are re-loaded, and few-shot training on a different train dataset B is executed. The model is then ready to make predictions on anomalies on a test dataset B using the approach described in 3.4.3.

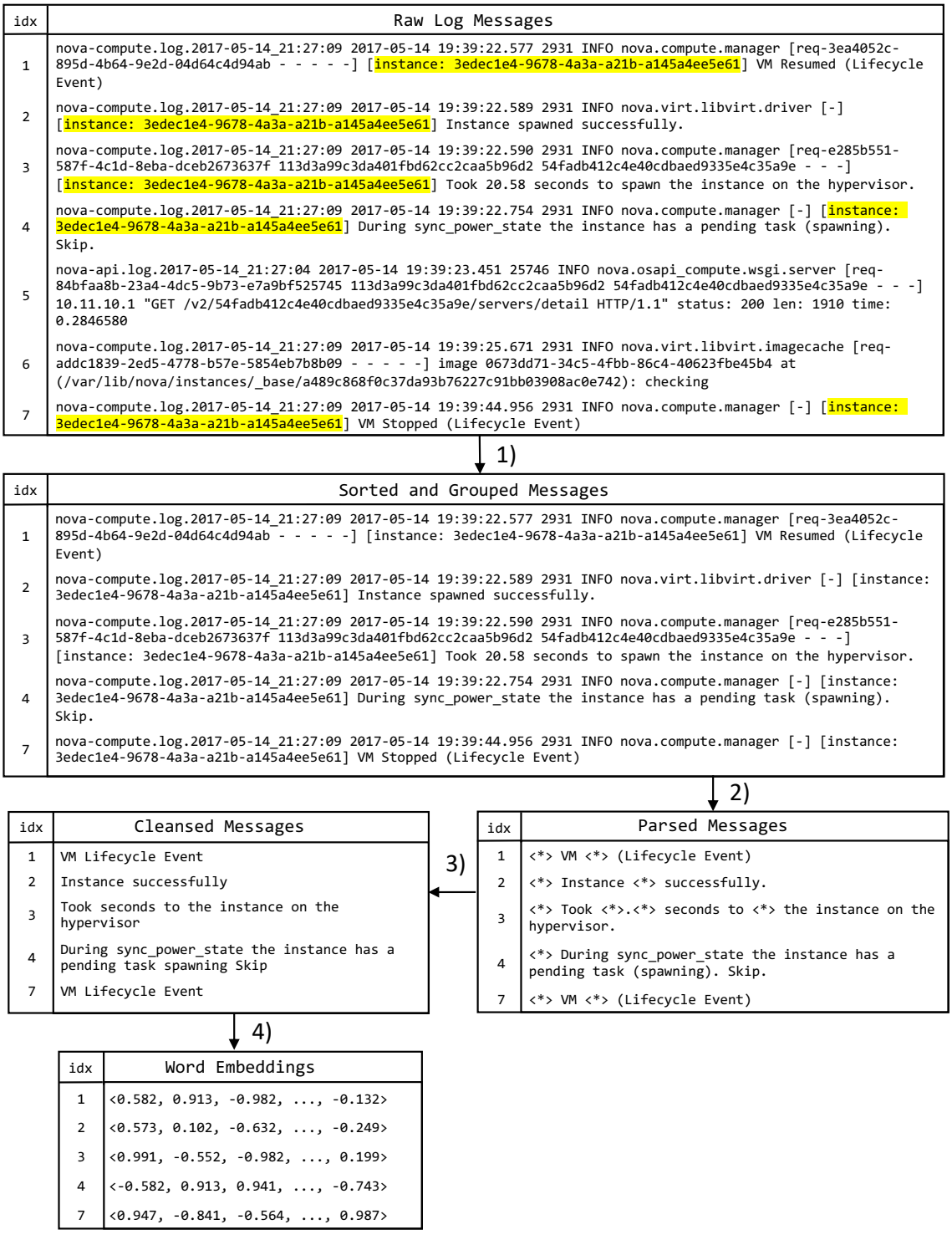


Figure 3.9: One full iteration of the pre-processing pipeline.

4 Results

This chapter presents the results of various experiments of the proposed concept including evaluation of how effective the representations from different language models are for anomaly detection on 1) already seen log data during training and 2) unstable log data that changes due to log updates. In particular, the language models Bert, XL-Transformers and GPT-2, as described in 3.3.3, are compared. Additionally, the regression-based method, described in 3.4.3, and the classification method, described in 3.4.2, are compared, showing advantages or disadvantages among each other.

Section 4.1 lists details about the used hardware and technologies, followed by section 4.2, in which the results of the evaluation are presented. Finally, section 4.3 presents the results of the evaluation.

4.1 Experimental Setup

In this section, the used hardware, libraries and the dimensions of the word embeddings for every language model and the used log dataset are described.

4.1.1 Hardware and Library Specifications

The machine that was used to conduct the experiments has the following specifications:

- Intel(R) Core(TM) i5-9600K CPU @ 3.70GHz
- 128 GB RAM
- 2 x NVIDIA GeForce RTX 2080 Ti 11GB RAM
- Ubuntu 18.04.3 LTS

The most relevant libraries that were used are:

- Python 3.6.7
- Numpy 1.14.5
- PyTorch 1.4.0
- Transformers 2.3.0

- Tensorflow 2.1.0

The word embedding vector of the used language models have the following dimensions:

- Bert: 768
- GPT-2: 768
- XL-Transformers: 1024

4.1.2 Anomaly Detection on one Dataset

For evaluating the model the OpenStack datasets "normal log dataset 2" (referenced to as original dataset from now on) containing 137k log lines and the "log dataset having performance anomalies" containing 18k log lines (referenced to as test dataset from now on), provided by Loghub, are utilised [30]. The original dataset stems from logs that were recorded by running multiple OpenStack instances for 20 hours and 13 minutes on CloudLab [31], and 2 hours and 17 minutes for the test dataset.

The test dataset contains anomalies which are only detectable by inspecting the parameters of the log events, as described in 2.9, i.e. the variable part of the log event which is not visible in the template after parsing. Therefore, a total of 5% of anomalies in relation to the total number of lines of the test dataset are injected into the test dataset, yielding a labeled test dataset for evaluation of the trained model. This is necessary due to the assumptions specified in 3.1.2. Two types of anomalies are injected separately:

1. An anomaly that is *semantically different* from the other log events. In this case the log event "*Timeout for executing the request*" is injected at a ratio of 5%. In order to assess the quality of the used word embeddings further, log alterations are additionally injected into the dataset containing anomalies, in order to simulate the instability of logs as described in 3.3.5.

2. Anomalies that are *semantically similar* to the other log events. For 5% of the log lines, 50% of the words are replaced by other words (e.g. "time", "during", "causing", "replace" and "crashed"). For example, "*Took seconds to build instance*" will be changed to "*Took replace during build causing*".

Table 4.1 gives an overview of the amount of log lines per dataset. The number of *raw lines* reflects the number of log lines in the dataset's initial state, while the number of *lines ordered* specifies the number of log lines after the log has been ordered by instance id, as described in 3.3.1 and depicted in 3.9.

Dataset	#lines raw	#lines ordered	Anomalies
Train	137k	33k	0%
Test	18k	5k	5%

Table 4.1: Test and train dataset.

4.1.3 Transfer of Knowledge

In order to examine the ability of the model to transfer the knowledge obtained from one dataset to a different one, the same OpenStack log dataset utilised in 4.1.2 is used as a basis. For this experiment, all occurrences of the templates which are visible in figure 4.1 on the left side are replaced by the templates visible on the right side. Additionally, all alterations presented in section 3.3.5 - line shuffling, duplication and deletion; and word insertion, removal and replacement - are injected into the dataset at ratios of 5%, 10% and 15% each in dataset A. This heavy modifications simulate a different version of the dataset. Then, the model is trained on this dataset A for 60 epochs. Dataset B, the unaltered version of dataset A is then used to conduct few-shot training on the model, followed by anomaly detection on a dataset that has the same characteristics as dataset B, i.e. no alterations.

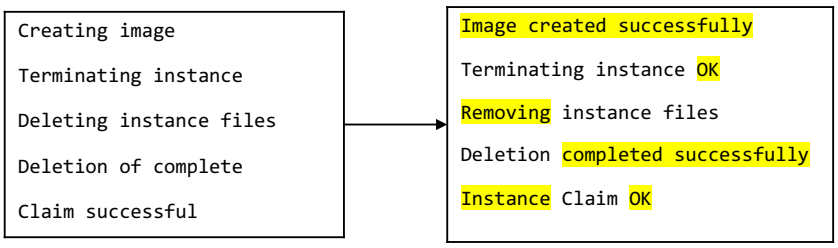


Figure 4.1: Changes on templates of the original dataset.

4.2 Evaluation

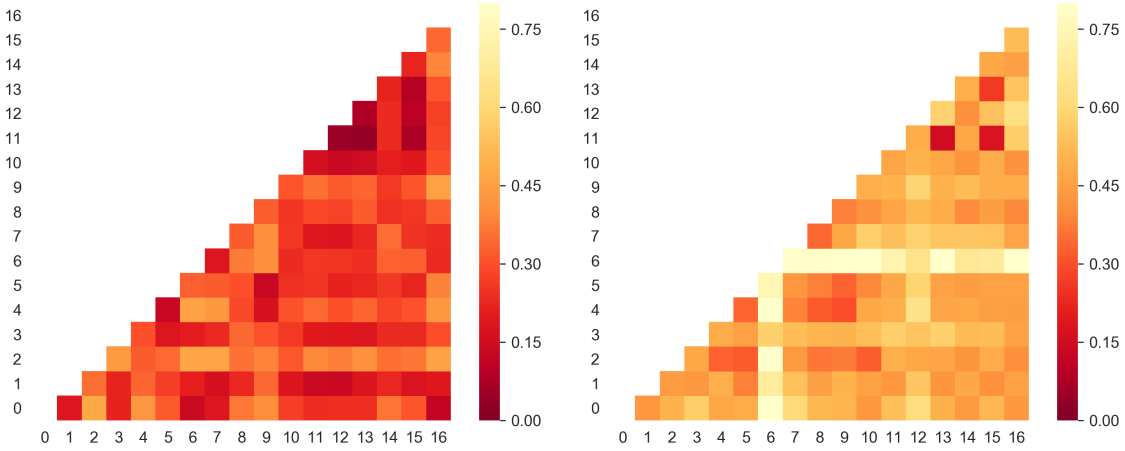
In this section the results of the evaluation between the two anomaly detection approaches, the regression-based approach and the classification-based approach are compared using three different language models: namely Bert, XL-Transformers and GPT-2. The pre-processing steps string cleansing and finetuning, as described in 3.3, are investigated with regards to their potential on benefiting the quality of the word embeddings. Subsequently, the results of the evaluation using one dataset and the transfer of knowledge approach are presented.

4.2.1 String cleansing

String cleansing, as described in 3.3.2, can potentially improve the distinguishability between templates drastically in the used dataset. For Bert a heatmap that shows the pairwise cosine distances between templates before cleansing is depicted in figure 4.2a. The corresponding distances after cleansing in figure 4.2b. The corresponding templates for the indices on the x and y axes before and after cleansing can be found in table 4.3 and table 4.4, respectively. The cosine distances before and after cleansing for GPT-2 can be found in figure 4.3a and figure 4.3b, and for XL-Transformer in figure 4.4a and figure 4.4b, respectively. The average pairwise cosine distances between templates before and after cleansing can be seen in table 4.2. The average cosine distance between templates increases by 96% for GPT-2, yet the initial value is already very low with only 0.0027. We can see how the average cosine distance between templates increases for Bert by 84%, from 0.2718 to 0.5004 and for XL-Transformers from 0.2511 to 0.7001. While Bert has a slightly higher average pairwise cosine distance before cleansing, XL-Transformers highly benefits from cleansing demonstrated by an increase of 178%. This underlines the importance of this step in order to meet the requirements postulated in 3.3.3, depending on the utilised language models.

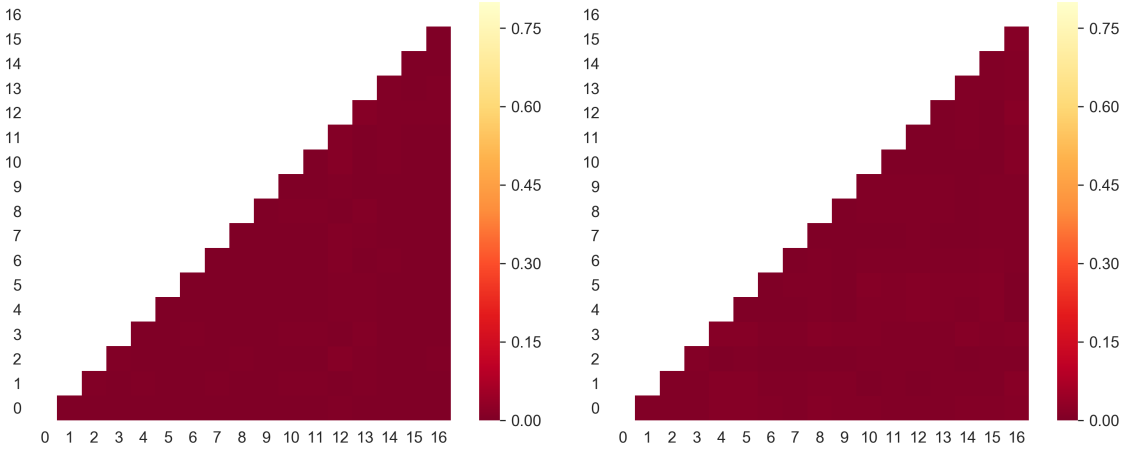
Model	Before cleansing	After cleansing
XL	0.2511	0.7001
Bert	0.2718	0.5004
GPT-2	0.0027	0.0053

Table 4.2: Average pairwise template cosine distances.



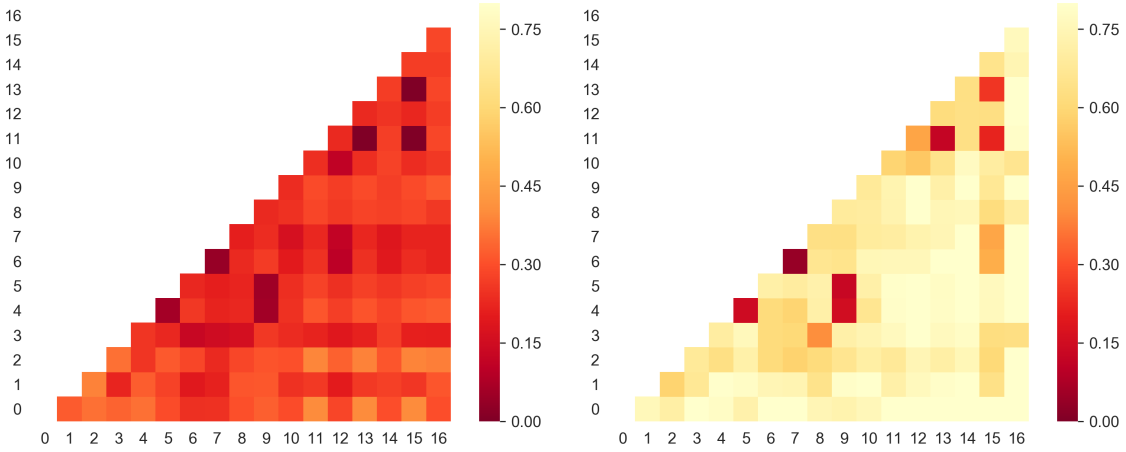
(a) Before cleansing. (b) After cleansing.

Figure 4.2: Bert pairwise template cosine distance.



(a) Before cleansing. (b) After cleansing.

Figure 4.3: GPT-2 pairwise template cosine distance.



(a) Before cleansing.

(b) After cleansing.

Figure 4.4: XL-Transformers pairwise template cosine distance.

Index	Template
0	⟨∗⟩ Creating image
1	⟨∗⟩ VM ⟨∗⟩ (Lifecycle Event)
2	⟨∗⟩ During sync_power_state the instance has a pending task (spawning). Skip.
3	⟨∗⟩ Instance ⟨∗⟩ successfully.
4	⟨∗⟩ Took ⟨∗⟩.⟨∗⟩ seconds to ⟨∗⟩ the instance on the hypervisor.
5	⟨∗⟩ Took ⟨∗⟩.⟨∗⟩ seconds to build instance.
6	⟨∗⟩ Terminating instance
7	⟨∗⟩ Deleting instance files ⟨∗⟩
8	⟨∗⟩ Deletion of ⟨∗⟩ complete
9	⟨∗⟩ Took ⟨∗⟩.⟨∗⟩ seconds to deallocate network for instance.
10	⟨∗⟩ Attempting claim: memory ⟨∗⟩ MB, disk ⟨∗⟩ GB, vcpus ⟨∗⟩ CPU
11	⟨∗⟩ Total memory: ⟨∗⟩ MB, used: ⟨∗⟩.⟨∗⟩ MB
12	⟨∗⟩ memory limit: ⟨∗⟩.⟨∗⟩ MB, free: ⟨∗⟩.⟨∗⟩ MB
13	⟨∗⟩ Total disk: ⟨∗⟩ GB, used: ⟨∗⟩.⟨∗⟩ GB
14	⟨∗⟩ ⟨∗⟩ limit not specified, defaulting to unlimited
15	⟨∗⟩ Total vcpu: ⟨∗⟩ VCPU, used: ⟨∗⟩.⟨∗⟩ VCPU
16	⟨∗⟩ Claim successful

Table 4.3: Templates before cleansing.

Index	Template
0	Creating image
1	VM Lifecycle Event
2	During sync power state the instance has a pending task spawning Skip
3	Instance successfully
4	Took seconds to the instance on the hypervisor
5	Took seconds to build instance
6	Terminating instance
7	Deleting instance files
8	Deletion of complete
9	Took seconds to deallocate network for instance
10	Attempting claim memory MB disk GB vcpus CPU
11	Total memory MB used MB
12	memory limit MB free MB
13	Total disk GB used GB
14	limit not specified defaulting to unlimited
15	Total vcpu VCPU used VCPU
16	Claim successful

Table 4.4: Templates after cleansing.

4.2.2 Finetuning

As described in 3.3.4, finetuning can potentially help producing word embeddings that are more adequate for solving certain natural language processing tasks. As described in 3.3.3, a high cosine distance between semantically different templates is required. The dataset that consists of the templates in table 4.4 has been chosen for finetuning. Since the pre-trained language models Bert, GPT-2 an XL-Transformers have been trained on a large corpus, finetuning would also need to be executed on a sufficiently large corpus. Since this is not the case, the results of finetuning for a maximum of four epochs, as suggested by the Bert authors [18], does not yield the desired results. As it can be seen in figure 4.6, it was not possible to increase the cosine distances between templates on the task of Masked LM (as described in 2.4.3), compared to the cosine distances depicted in figure 4.2b. The average distance between templates dropped from 0.4449 to 0.3016. The fact that the loss on the evaluation part of the dataset is not decreasing adequately, as shown in figure 4.5, shows that through learning on such a small corpus it is not possible to generalise well enough, which makes finetuning on the default learning task not useful in this case. Since the Huggingface Transformers library does not offer finetuning interfaces for GPT-2 and XL-Transformers on the same task as for Bert, they are not further investigated for finetuning.

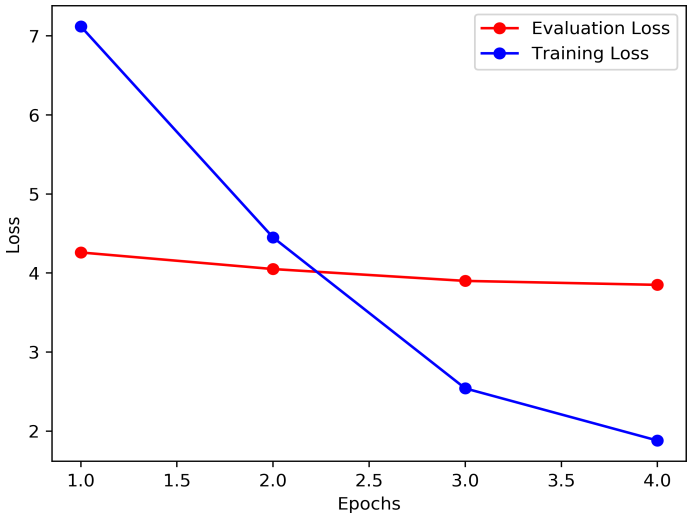


Figure 4.5: Training and evaluation loss for finetuning on masked LM.

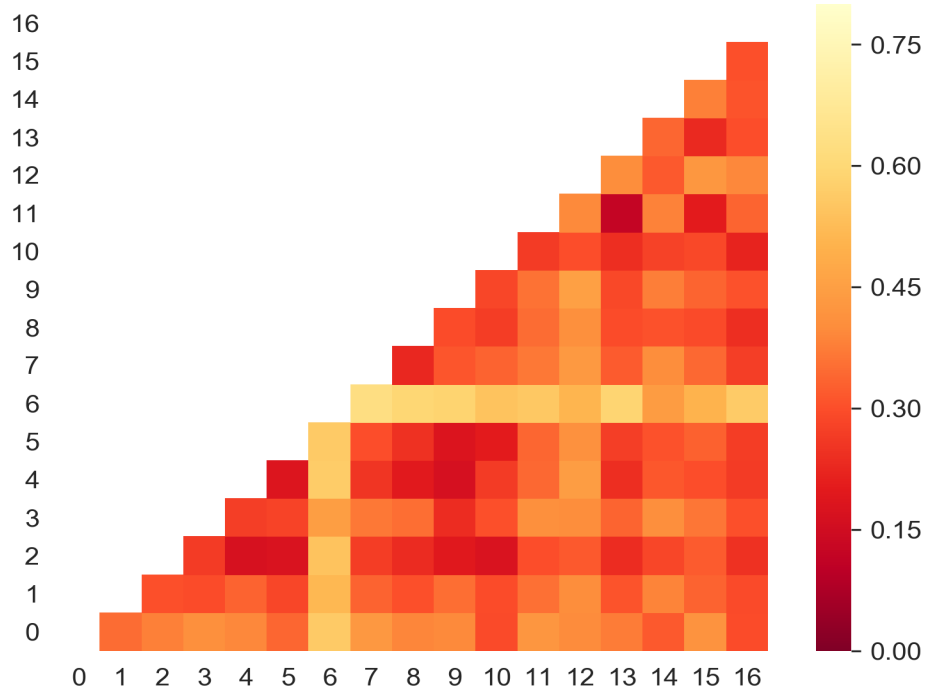


Figure 4.6: Cosine distance between templates after cleansing and finetuning

4.2.3 Hyperparameters

In order to find well-suited parameters for the LSTM model, it was initially applied to the problem using a minimal configuration. With the help of a grid-search and running simulations with different configurations, the following hyperparameters have proven to yield the most satisfying results:

- 512 hidden units for the bidirectional LSTM, with one layer;
- two fully connected layers with 512 units $\rightarrow$ 256 units and 256 units $\rightarrow$ output size;
- dropout of 0.1 between every layer;
- input sequence length of 7;
- 60 epochs of training.

4.2.4 Regression

In this subsection the results of the regression-based approach, including various alterations using the three different language models, are presented. In order to evaluate

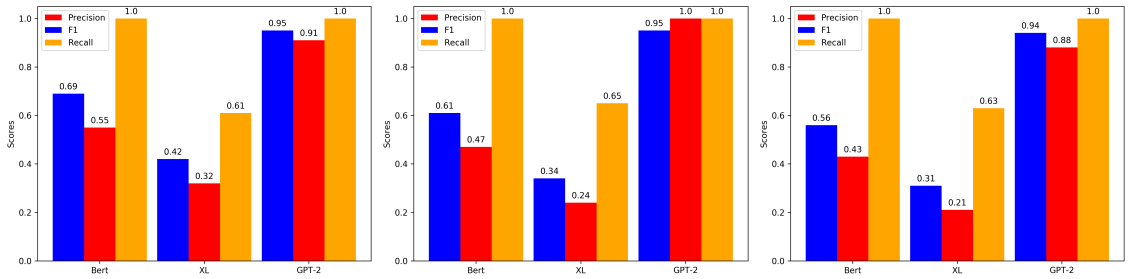
the robustness of the language models to the evolution and instability of log events, alterations as described in 3.3.5 are injected at different ratios. The impact on detecting semantically different anomalies after alterations on the sequence of logs are injected, i.e. deleting, shuffling and duplicating events are summarised in figure 4.7. Alterations are not injected all at the same time, but independently from each other. The results in the figures are average values of all experiments with alterations on the log sequences. Results broken down by each alteration can be found in the appendix .1. From 5% to 15% alterations, both Bert and GPT-2 show recall values of 1.0, while XL-Transformers is only able to detect 61% to 65% of all anomalies. With regards to precision, GPT-2 achieves values from 0.91 for 5% alterations to 1.0 for 10% alterations to 0.88 for 15% alterations, while Bert declines from 0.55 at 5% alterations to 0.43 with 15% alterations. XL-Transformers in turn returns low values of 0.32 for 5% alterations to 0.21 for 15% alterations. Moreover, for F1-score GPT-2 achieves very good results with about 0.95 for all alteration ratios, while Bert declines from 0.69 for 5% alterations to 0.56 for 15% alterations. XL-Transformers achieves a F1-score of 0.42 for 5% alterations, 0.34 for 10% alterations and 0.31 for 15% alterations. It is evident that GPT-2 performs better than Bert and XL-Transformers in all metrics in this category, except for recall, where Bert is performing equally well. Bert can be located as second place in this category, while XL-Transformers delivers the least promising results in comparison to Bert and GPT-2.

In addition to the alterations on the log sequences, alterations on the log events themselves, i.e. inserting, removing and replacing words are also of interest, with subsequent injection of semantically different anomalies. The results of this experiment can be seen in figure 4.8. Again, the alterations are not injected all at once, but independently – the figure shows averaged results. Exactly as for the previous results, we can see that both Bert and GPT-2 achieve a recall value of 1.0 for all alteration ratios, while XL-Transformers achieves results of around 0.63 for all alteration ratios. GPT-2 performs equally good on both F1-score and precision as in the previous experiments, while Bert achieves F1-scores which are between 8 and 10 percentage points better, and for precision improvements of around 7 percentage points. XL-Transformers shows even higher improvements than Bert, with an average improvement of 13 percentage points for F1-score and an average improvement of 17 percentage points for precision, with a slight degradation in recall of 3 percentage points on average. Again, GPT-2 shows the best results overall, followed by Bert and XL-Transformers.

The results for injecting semantically similar anomalies can be see in figure 4.9. It is clearly visible that this experiment shows a different picture than the previous ones. GPT-2 achieves less qualitative results than before, while Bert and especially XL-Transformers show partly better results than they did with the the injection of semantically different anomalies. Bert achieves a F1-score of 0.71, XL-Transformers 0.6 and GPT-2 0.09. Also with regards to precision the results are slightly better for Bert

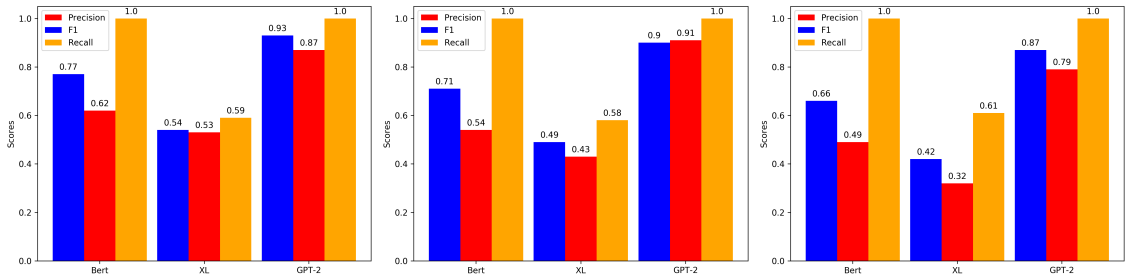
and XL-Transformers - Bert achieving 0.61, XL-Transformers 0.6, but GPT-2 only 0.13. XL-Transformers shows improvements for recall with 0.75, while Bert degrades to 0.86 and GPT-2 degrades to 0.07. This experiment, complementary to the previous ones, where a semantically different anomaly line was injected and GPT-2 proved to be robust, shows that GPT-2 is not able to detect semantically similar anomalies well. Bert is most useful for the task of anomaly detection using the regression approach overall.

Another aspect that is evaluated is the impact of the input sequence length, i.e. the number of concatenated log events, for which the next log event shall be predicted on the ability of the model to detect anomalies. The results of this evaluation can be seen in 4.10, where for 15% of the log lines one random word is inserted as alteration, and 5% semantically different anomalies are injected. Bert profits from sequence lengths longer or equal to 6, whereas the quality of results for GPT-2 and XL-Transformers are not as affected from the input sequence length, but are generally lower for XL-Transformers.



(a) 5% alteration (b) 10% alteration (c) 15% alteration

Figure 4.7: Altering the sequences of logs at different ratios, inject semantically different anomalies, using regression.



(a) 5% alteration (b) 10% alteration (c) 15% alteration

Figure 4.8: Altering log lines at different ratios, inject semantically different anomalies, using regression.

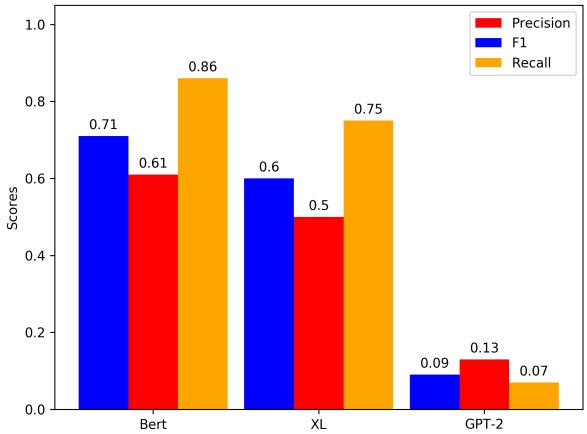


Figure 4.9: Inject semantically similar anomalies, using regression.

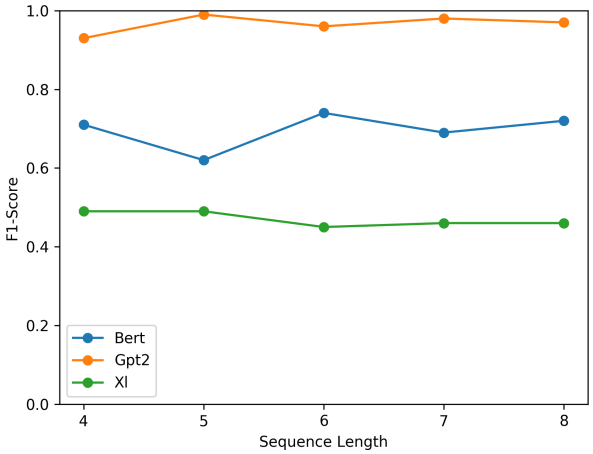


Figure 4.10: F1-Score for varying input sequence lengths, for 15% of the log lines insert one word as alteration, inject semantically different anomalies, using regression.

4.2.5 Classification

In this subsection, the results of the classification-based approach are presented, under the exact same conditions as in 4.2.4. The ability of the system to detect semantically different anomalies, despite alterations on the sequences of logs, i.e. deleting, shuffling and duplicating events, can be seen in 4.11. For the classification-based approach a different picture becomes apparent than for the regression-based approach, where GPT-2 showed better results for the injection of semantically different anomalies. For classification though, Bert and XL-Transformers show recall values of 1.0 throughout, where GPT-2 only reaches recall values of around 0.7. Bert achieves a F1-scores of 0.67 for 5% alterations, 0.59 for 10% alterations and 0.54 for 15% alterations, while XL-Transformers achieves 0.51 for 5%, 0.45 for 10% and 0.41 for 15% alterations, and GPT-2 achieves 0.44 for 5% alterations, 0.39 for 10% alterations, and 0.36 for 15% alterations. For precision, Bert achieves 0.37 for 15% alterations, XL-Transformers 0.25 and GPT-2 0.24. Bert shows the best F1-score and precision throughout, followed by XL-Transformers and GPT-2.

The impacts of alterations on the logs themselves, i.e. inserting, removing and replacing words are summarised in figure 4.12, where semantically different anomalies are injected. As mentioned above, only averaged results on the individual injections are presented. Results broken down by each alteration can be found in the appendix .2. Here it becomes evident again, similarly to the results for the regression-based approach, that both Bert and XL-Transformers perform better for the detection of semantically different anomalies with alterations on the logs themselves than alterations on the sequences of logs, while GPT-2 performs slightly worse when these two categories are compared. Once again, as for the alterations on the log sequences, Bert and XL-Transformers both achieve a recall value of 1.0, while GPT-2 achieves a recall value of around 0.7 for all alteration ratios. Bert achieves a F1-score of 0.77 for 5% alterations, 0.7 for 10% alterations and 0.67 for 15 % alterations. XL-Transformers achieves a F1-score of 0.58 for 5% alterations, 0.55 for 10% alterations and 0.53 for 15% alterations. GPT-2 achieves F1-scores of 0.53 for 5% alterations, 0.46 for 10% alterations, and 0.43 for 15% alterations. For precision, Bert achieves a value of 0.63 for 5% alterations, 0.53 for 10% alterations and 0.5 for 15% alterations. XL-Transformers achieves a precision of 0.41 for 5% alterations, 0.38 for 10% alterations and 0.36 for 15% alterations. GPT-2 achieves 0.43 for 5% alterations, 0.34 for 10% alterations and 0.31 for 15% alterations.

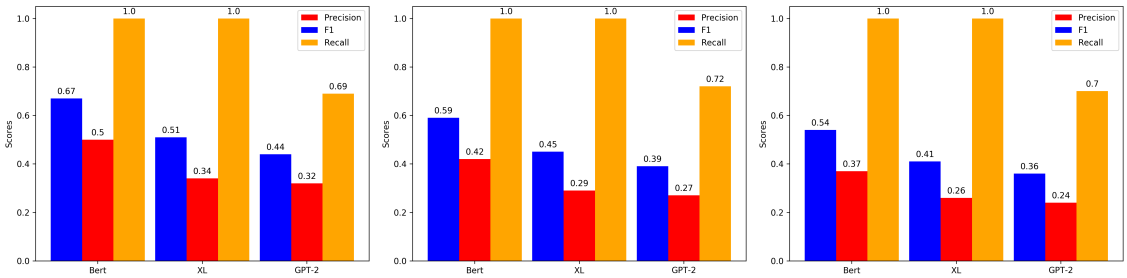
Overall, it can be stated that Bert performs best for detecting semantically different results using the classification approach, followed by XL-Transformers and GPT-2.

The impact of the input sequence length on the F1-scores can be seen in 4.13, where for 15% of the log lines one word is inserted to simulate unstable logs, and 5% semantically different anomalies are injected. Bert again seems to profit slightly from sequence lengths longer than 6, whereas the quality of results for XL-Transformers and GPT-2 seem to

have a tendency to degrade in prediction quality for longer input sequence lengths.

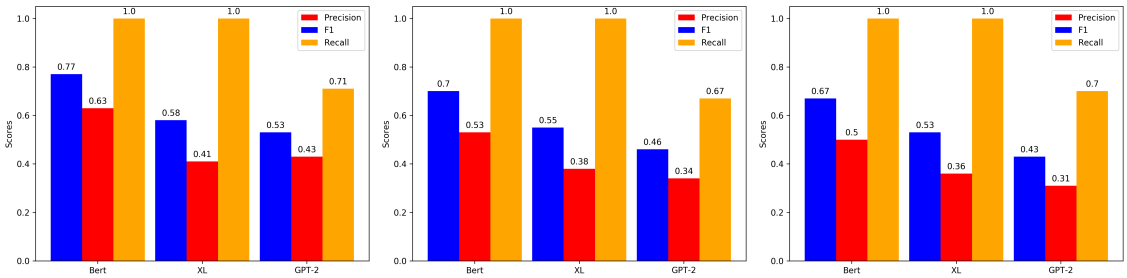
The results for injecting semantically similar anomalies can be seen in figure 4.14. In comparison to the previous results, better F1-scores and precision are achieved by Bert, XL-Transformers and GPT-2. The results for recall stay essentially the same. While Bert is able to achieve a F1-score of 0.94, with 0.9 precision and recall of 1.0, XL-Transformers achieves a F1-score of 0.81, with 0.68 precision and recall of 1.0 and GPT-2 returns a F1-score of 0.71, precision of 0.73 and recall of 0.68, showing that Bert is able to detect semantically similar anomalies best, when using the classification approach.

Similarly to the results using regression, the classification approach for detecting semantically similar anomalies seems most fit for Bert and the least for GPT-2, yet GPT-2 is able to achieve much more useful results using the classification approach when detecting semantically similar anomalies, than using the regression approach, where it is the strongest detecting semantically different anomalies.



(a) 5% alteration (b) 10% alteration (c) 15% alteration

Figure 4.11: Altering the order of log sequences at different ratios, inject semantically different anomalies, using classification.



(a) 5% alteration (b) 10% alteration (c) 15% alteration

Figure 4.12: Altering log events at different ratios, inject semantically different anomalies, using classification.

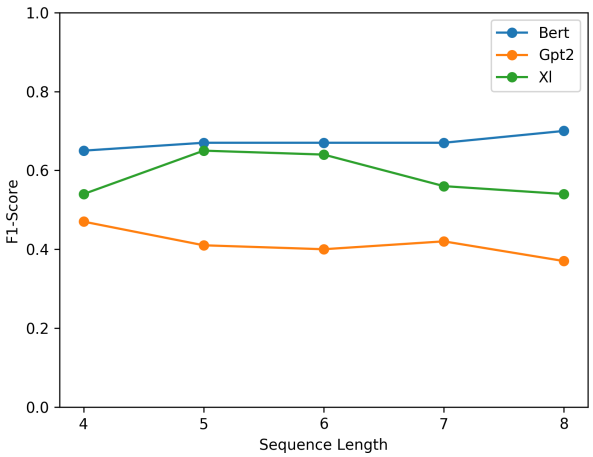


Figure 4.13: F1-Score for varying input sequence lengths, for 15% of the log lines insert one word as alteration, inject semantically different anomalies, using classification.

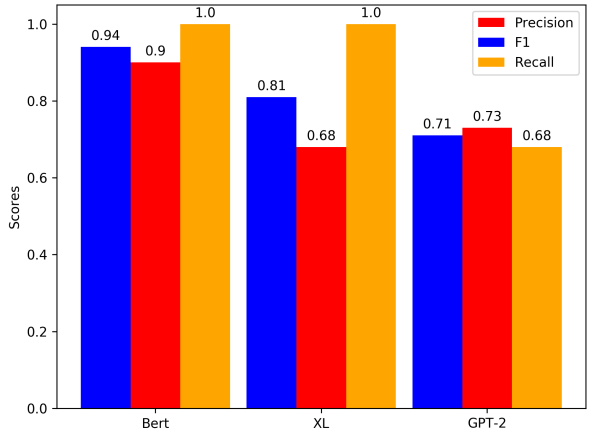


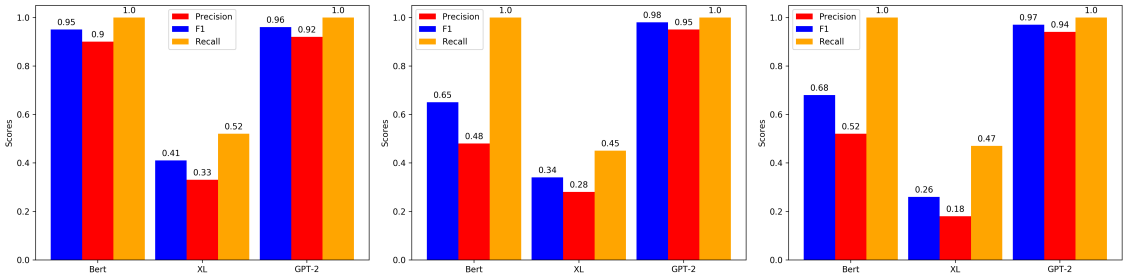
Figure 4.14: Inject semantically similar anomalies, using classification.

4.2.6 Transfer of Knowledge Using Regression

As described in 4.1.3, the alterations that were injected separately in the experiments on one dataset in the last section, are now injected all at once, in order to simulate a different dataset A. Figure 4.15 shows the results for alterations on 5%, 10% and 15% of the log lines of all alterations each at the same time and injection of semantically different anomalies after 60 epochs of training on the train dataset A and 5 epochs of training on the train dataset B. Similar to the previous experiments using regression, both Bert and GPT-2 achieve perfect recall values of 1.0, while XL-Transformers only achieves around 0.49 for increasing injection ratios. With regards to F1-score and precision, GPT-2 performs far better than Bert and XL-Transformers, yet Bert achieves a good F1-score of 0.95 and precision of 0.9 for 5% alteration ratios, but degrades substantially for 10% and 15% to around 0.67 in F1-score and around 0.5 in precision. XL-Transformers shows an F1-score of 0.41 for 5% alterations, 0.34 for 10% alterations and 0.26 for 15% alterations, and precision of 0.33 for 5% alterations, 0.28 for 10% alterations and 0.18 for 15% alterations. GPT-2 shows very stable results, even with increasing ratio of injection. With a F1-score of 0.96 for 5% alterations, 0.98 for 10% alterations and 0.97 for 15% alterations and precision of 0.92 for 5% alterations, 0.95 for 10% alterations and 0.94 for 15% alterations and recall of 1.0, is ahead of the other two language models in every metric, except for recall, where Bert performs equally well. Bert performs second-best, while XL-Transformers achieves the least useful results.

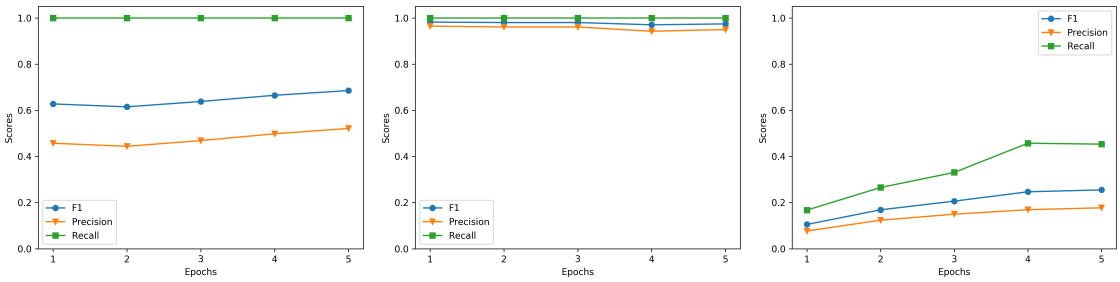
Figure 4.16 depicts the development of the metrics of detecting anomalies for every additional epoch of training on dataset B. It is clearly visible, that XL-Transformers improves the most per epoch, with an increase of the F1-score from 0.1 to 0.26 and recall from 0.17 to 0.47. Bert has a smaller increase per training epoch, with F1-score increasing from 0.62 to 0.68, while recall already starts at 1.0. The results of GPT-2 do not change significantly much per epoch for every metric, but start at a very high level already for every metric clearly above 0.9, corresponding to the findings already made on GPT-2 using regression in 4.2.4.

The results for injecting 5% semantically similar anomalies for transfer of knowledge with 15% alterations using regression can be seen in figure 4.17. Similar results as in 4.2.4 are yet again achieved, where GPT-2 proves to be robust in the previous experiment, it is not able to distinguish the semantically similar log lines from the normal ones well. Bert and XL-Transformers are able to achieve better results, with Bert achieving a F1-score of 0.63, 0.58 and recall of 0.7, XL-Transformers achieving a F1-score of 0.55 precision of 0.45 and recall of 0.7, GPT-2 is only able to achieve a F1-score of 0.08, precision of 0.23 and recall of 0.05. This proves again that Bert is able to execute the task of anomaly detection for transfer of knowledge using regression best overall, when both types of anomaly injections are considered.



(a) 5% alteration (b) 10% alteration (c) 15% alteration

Figure 4.15: Transfer of knowledge with different ratios of alteration, 5% semantically different anomalies, using regression.



(a) Bert (b) GPT-2 (c) XL

Figure 4.16: Improvement of metrics for transfer of knowledge per additional learning epoch, with 15% alterations and injection of semantically different anomalies, using regression.

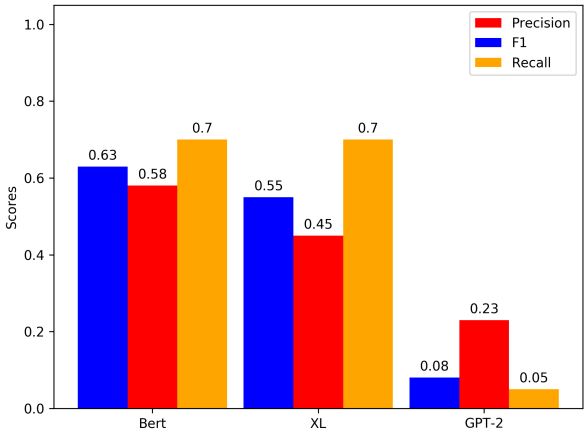


Figure 4.17: Transfer of knowledge. 15% alteration, injection of semantically similar anomalies, using regression.

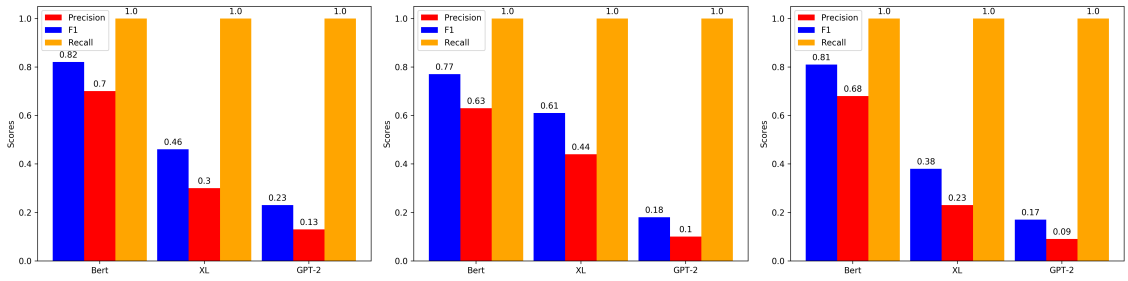
4.2.7 Transfer of Knowledge Using Classification

For transfer of knowledge using the classification-based approach, the same experiments as described in 4.2.6 were conducted. Figure 4.18 shows the results of the transfer of knowledge experiment with injection of semantically different anomalies. Interesting observations include the stable results achieved using Bert which are little sensitive to increasing alteration ratios. Bert achieves a F1-score of 0.82 for 5% alterations, 0.77 for 10% alterations and 0.81 for 15% alterations, and precision of 0.7 for 5% alterations, 0.63 for 10% alterations and 0.68 for 15% alterations. For XL-Transformers, the F1-score is 0.46 for 5% alterations, 0.61 for 10% alterations and 0.38 for 15% alterations, while precision goes from 0.3 for 5% alterations to 0.44 for 10% alterations to 0.23 for 15% alterations. GPT-2 shows overall less useful results than for the regression approach and the injection of semantically different anomalies, with a F1-score of 0.23 for 5% alterations, 0.18 for 10% alterations and 0.17 for 15% alterations, precision of 0.13 for 5% alterations, 0.1 for 10% alterations and 0.09 for 15% alterations, and a recall of 1.0 for all injection ratios.

The results show that Bert has clear advantages over GPT-2 and XL-Transformers for the classification-based approach. It is noticeable, that all language models achieve a recall value of 1.0. This is due to the high cosine distance of the injected semantically different anomaly log event.

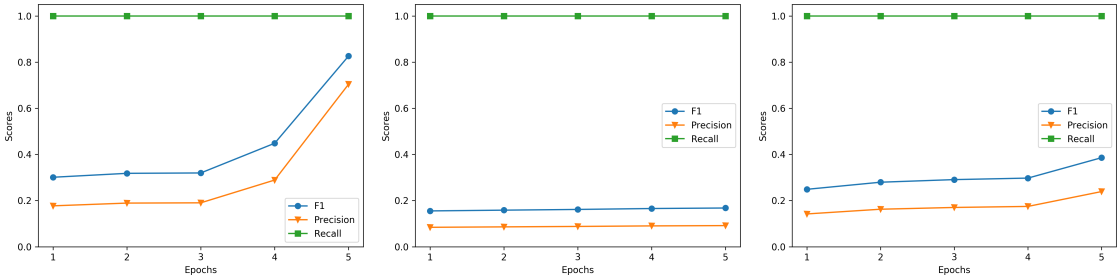
Figure 4.19 depicts the development of the metrics of detecting semantically different anomalies for every additional epoch of training on dataset B. After every epoch of training on the train dataset B, the labelled test dataset B is fed into the model, and metrics are collected. Bert improves the most per epoch, with an initial F1-score of 0.3 for one epoch of training to 0.81 after 5 epochs, and precision of 0.18 after one epoch to 0.68 after 5 epochs. GPT-2 shows no significant changes, while XL-Transformers improves its F1-score from 0.24 after one epoch to 0.38 after 5 epochs and precision from 0.17 after one epoch to 0.23 after 5 epochs. All language models start with a recall value of 1.0 after one epoch which stays the same after 5 epochs.

The results for injecting 5% semantically similar anomalies for transfer of knowledge with 15% alterations using classification can be seen in figure 4.20. In comparison to the usage of the regression-based approach for transfer of knowledge and injection of semantically similar anomalies as explained in 4.2.6 and depicted in 4.17, all three language models achieve better results, especially GPT-2. All three language models also achieve a recall value of 1.0. Bert achieves a F1-score of 0.75 and a precision of 0.61, XL-Transformers achieves a F1-score of 0.69 and a precision of 0.53, GPT-2 achieves a F1-score of 0.43 and precision of 0.27. Bert is obviously the most fit for the task of anomaly detection using the classification-based approach, for injection of semantically different and semantically similar anomalies.



(a) 5% alteration (b) 10% alteration (c) 15% alteration

Figure 4.18: Transfer of knowledge with different ratios of alteration, 5% semantically different anomalies, using classification.



(a) Bert (b) GPT-2 (c) XL

Figure 4.19: Improvement of metrics for transfer of knowledge per additional learning epoch, with 15% alterations and injection of semantically different anomalies, using classification.

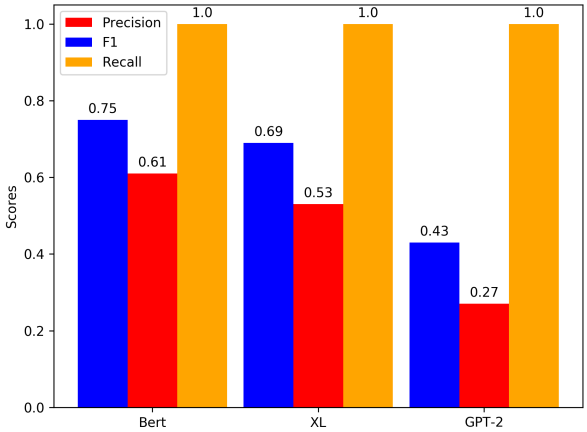


Figure 4.20: Transfer of knowledge. 15% alteration, injection of semantically similar anomalies, using regression.

4.3 Discussion of Results

By evaluating three different language models, it has been shown that the used word embeddings can highly influence the quality of the results for anomaly detection, with different word embeddings having strengths and weaknesses in different categories.

While GPT-2 shows strengths in the regression-based approach with injection of semantically different anomalies – achieving a F1-score of up to 0.94, and being far better overall than Bert and XL-Transformers – it shows clear disadvantages when semantically similar anomalies are injected, where Bert is the best of the three language models, with a F1-score of 0.71. GPT-2 shows very stable results also for the transfer of knowledge approach, using regression for injection of semantically different anomalies, with a F1-score of up to 0.97, while Bert and XL-Transformers degrade substantially when the ratio of alteration is increased. Yet again GPT-2 drops significantly for the injection of semantically similar anomalies, with a F1-score of 0.08, while Bert is achieving the best results with a F1-score of 0.63.

For the classification-based approach, the results show a different picture in general. While both Bert and XL-Transformers are able to achieve a recall score of 1.0 for injection of semantically different and similar anomalies, GPT-2only achieves around 0.7. The results do not differ as substantially for GPT-2 between semantically different (F1-score up to 0.43 for 15% alterations) and semantically similar injections (F1-score of 0.71) as they do in comparison to the regression-based approach. Bert is able to achieve an F1-score of up to 0.67 for 15% alterations and injection of semantically different anomalies, XL-Transformers reaches up to 0.53. For semantically similar anomalies, Bert achieves a F1-score of 0.94 and XL-Transformers 0.81. Bert is clearly ahead in this category. For the transfer of knowledge using classification, and injection of a semantically different anomaly, all are able to achieve full recall scores, due to the semantic nature of the anomaly. Bert is clearly the best in this category. Also for the semantically similar injection, Bert is the most fit language model.

Overall it can be stated that Bert shows the most steady and best results overall, while GPT-2 is highly dependent on the semantic nature of the injected anomaly, showing very good results for the injection of semantically different anomalies, and very unreliable results for the injection of semantically similar anomalies using the regression approach. XL-Transformers, on the other hand, profits from the regression-based approach.

The results prove the usability of the presented model. All three language models are generally suitable for anomaly detection.

5 Related Work

There has been a large amount of research and development of new approaches for anomaly detection in logs. Approaches can be characterised by the following categories: supervised learning models, unsupervised learning models, statistical models, classical machine learning models and, finally, deep learning models. Numerous supervised learning methods were applied to solve the problem of log anomaly detection. Liang et al. [32] and Yuan et al. [33] trained a SVM classifier to detect errors. Farshchi et al. [34] adopted a regression-based method, to find correlations between an operation's logs and the operation activity's effect on cloud resources. Chen et al. [35] presented a decision tree learning approach, to diagnose failures in large Internet sites. However, these methods have two limitations: They rely on system-specific labeled log data for training and do not provide a general method, to cope with ever-changing log data.

Additionally, unsupervised learning methods have been proposed. Xu et al. [36] use the Principal Component Analysis method, to construct a log count matrix, grouping log events to sessions with the session id which is available for every log event. Lin et al. [37] and [38] both proposed approaches that cluster logs.

The recent remarkable advances of deep learning depict new promising paths for anomaly detection in logs. While LSTMs generally have been put to use in detecting anomalies in time series [39], they have been used in anomaly detection in logs: Du et al. [3] presented DeepLog which is described in detail in section 5.1. Zhang et al. [4] used a LSTM similarly. Even though these approaches yield good results, they are not able to cope with changing log data, since log events have to be transformed into fixed indices.

There are studies that have applied NLP techniques and consider log events as natural language. Bertero et al. [40] used Google's word2vec algorithm to obtain word embeddings, exploiting the obtained feature space, using standard classifiers, like SVM and Random Forest, to detect anomalies. Zhang et al.[4], additionally to using the LSTM model for time series prediction, apply TF-IDF weight and consider each log event as a word. Brown et al. [41] use combine attention based models together with word word embeddings. These approaches do not take into account the contextual information in log sequences.

Very recently, LogRobust, which is described in detail in section 5.3 and LogAnomaly, described in section 5.2, use pre-trained word embeddings, using an attention-based Bi-LSTM model to learn log sequences.

5.1 DeepLog: Anomaly Detection and Diagnosis from System Logs through Deep Learning

Du et al. proposed DeepLog [3], a prominent example of a model that treats system logs as natural language sequences. An overview of the model is depicted in figure 5.1. The first step of the proposed model is exactly as in the majority of the works in the area: Logs are first pre-processed with a log parser, separating the constant from the variable part. Log templates are mapped to log keys k_i, which are indices between 0 to the number of different templates. For each log entry e_i, the elapsed time between e_i and e_{i-1} are stored in $\vec{v}_{e_i}$, together with parameter values in a parameter value vector. An LSTM is then trained on the sequences k_i to learn normal system execution paths. Given a window size of $h = 3$, a sequence of log keys $\{k_5, k_{11}, k_2, k_{14}, k_{15}\}$ would result in the *input sequence* and *output sequence* for training of $\{k_5, k_{11}, k_2 \rightarrow k_{14}\}$ and $\{k_{11}, k_2, k_{14} \rightarrow k_{15}\}$. Given these sequences of keys, the system is trained to maximise the probability of having $k_i \in K$ as the next log key value. It is thereby effectively learning the probability distribution $Pr(m_t = k_i | m_{t-h}, ..., m_{t-1})$. Given a log key sequence of length h, it outputs a probability distribution of all possible log key values. A log key value is treated as normal, if it is among the top g candidates. For the detection stage, new log key entries e_t are parsed into a log key m_t and parameter value vector. Subsequently, the trained model is used to check if the incoming log key is normal, by sending $w = \{m_{t-h}, ..., m_{t-1}\}$ as an input. Additionally, the parameter value vector is checked. If the log entry is labeled as being abnormal, the model provides semantic information for users to manually diagnose the anomaly. In order to adapt to changing patterns and log entries, the user has the possibility to mark a detected anomaly as a false positive, thus updating the model.

For verification of the model, the authors deployed an OpenStack experiment to fabricate their own log data. They produce over 1 million log entries, with 7% being abnormal. The model is able to achieve a precision of 0.96, recall of 1.0 and F1-score of 0.98.

Even though the model can be adjusted after the training phase on a particular dataset is already completed, by manually reporting false positives, thus, enabling the system to adapt to changing or new sequences of logs, it is not able to dynamically adapt to changes on the log events themselves. If log data changes on the log event level, a re-training of the model is necessary.

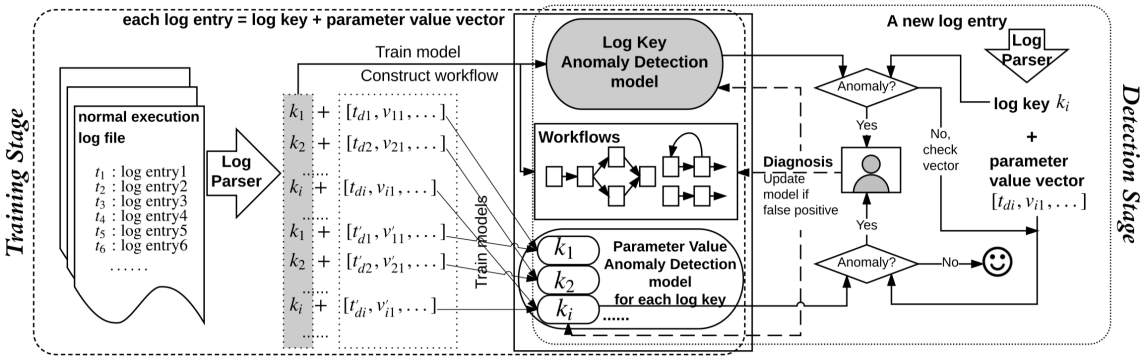


Figure 5.1: DeepLog model overview [11]

5.2 LogAnomaly: Unsupervised Detection of Sequential and Quantitative Anomalies in Unstructured Logs

LogAnomaly, the approach proposed by Meng et al. [1], models a log stream as a natural language sequence. Log event sequences are first parsed into template sequences using FT-Tree. These sequences are then transformed into embedding sequences using a novel word representation method, template2Vec, which is inspired by word2Vec [42]. Figure 5.4 shows the steps included for template2Vec in the offline learning stage: Step (1) shows the inclusion of common synonyms and antonyms in the English language extracted from WordNet [43]. Additionally, log-data-specific synonyms and antonyms can be added. In step (2), dLCE [44] is used as an embedding model, to generate word vectors that represent the words in templates, which are then transformed (3) into template vectors [1].

After the sequences of log events $S = (s_1, s_2, ..., s_m)$ have been transformed into template vector sequences $V = (v_{s_1}, v_{s_2}, ..., v_{s_m})$, an Attention-based Bi-LSTM is applied to learn sequences of normal logs. The sequence for detection is a sliding window of the w most recent template vectors, meaning that for a template vector sequence $V_j = v_{(s_j)}, v_{(s_{j+1})}, ..., v_{(s_{j+w-1})}$, the LSTM learns to predict the template vector v_{j+w}. In addition to sequential patterns, LogAnomaly considers quantitative patterns in sequences of templates. For example, opening files should also be closed, so the number of logs indicating that a file was opened should be equal to the number of logs indicating that a file was closed. If a log would break a certain invariant, it can be assumed that an anomaly has occurred during execution. For log messages $s_i \in S_j$, with S_j being a subsequence of S, the count vector of the log sequence $s_{i-w+1}, s_{i-w+2}, ..., s_i$ is calculated, denoted as $C_i = (c_i(v_1), c_i(v_2), ..., c_i(v_n))$, with $c_i(v_k)$ being the number of occurrences of v_k in the template vector sequence $v_{i-w+1}, v_{i-w+2}, ..., v_i$. C_j, $C_{j+1}, ..., C_{j+w-1}$ are then

inputs for the LSTM. This process is laid out in figure 5.2 [1].

In order to deal with new log templates in an online fashion, if an arriving log cannot be matched to an existing template, FT-Tree is utilised to extract a template from the new log and its template vector is calculated. Subsequently, the template will be matched on an existing one, based on the similarity among template vectors [1].

For verification of the model, the authors employed a manually labeled BGL dataset, containing around 4.7 million logs and a HDFS dataset with around 11 million logs, both with manual labels on anomalous logs or blocks of logs. For the BGL dataset they achieve a precision of 0.97, recall of 0.94 and F1-score of 0.96. For the HDFS dataset, they achieve a precision of 0.96, recall of 0.94 and F1-score of 0.95 [1].

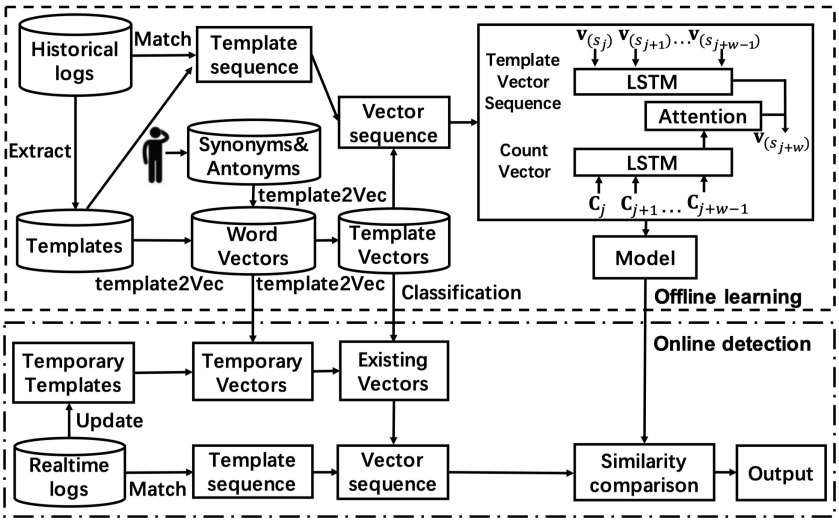


Figure 5.2: LogAnomaly model overview [1].

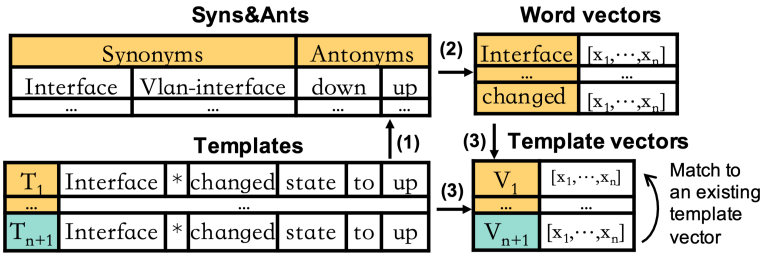


Figure 5.3: Example of Template2vec [1].

5.3 Robust Log-Based Anomaly Detection on Unstable Log Data

Zhang et al. proposed *LogRobust* [5], emphasising the necessity of anomaly detection on log data being sufficiently robust to the unstable nature of log data, which is twofold: On the one hand, they are *evolving*, meaning that frequently modifying source code naturally leads to logging statements getting altered. On the other hand, during collection, retrieval and pre-processing of log data, *processing noise* is introduced into the original log data. For example, in large-scale systems, many logs are produced by separate distributed components, potentially leading to missing, duplicated or disordered logs due to network errors or limited system throughput.

In order to capture the semantics of log events, LogRobust treats sequences of log events like sequences of natural language sentences, exactly as LogAnomaly, described in section 5.2. After adopting Drain to parse log data and extract log templates from it, they apply FastText [45] in order to replace words with corresponding vectors with dimension $d = 300$, thus transforming a log-event sentence S into a word vector list $L = (v_1, v_2, ..., v_N)$, where $v_i \in \mathbb{R}^d$ and N are the number of tokens in a log-event sentence. Next, all N word vectors that represent a log event are aggregated into a fixed-dimension vector. For this purpose, TF-IDF is applied to measure the importance of words in sentences. For example, if a word appears frequently in a log event, it means that this word is potentially highly representative for this log event, thus increasing its *Term Frequency* (TF) value. On the contrary, if a word appears in many log events, it diminishes the distinguishability of log events, leading to a low *Inverse Document Frequency* (IDF) value. In addition, for each word, its TF-IDF weight is calculated by TF-IDF. Finally, the template vector $V \in \mathbb{R}^d$ is obtained by summing up the weighted word vectors.

In a later stage, the log sequences are split into subsequences and are then used as input for an Attention-based Bi-LSTM. Hidden states at time step t of the forward (f) and backward (b) pass are concatenated as $h_t = \text{concat}(h_t^f, h_t^b)$. Log data noise can be reduced with the help of the attention layer, which can automatically learn the importance α_t of a log event. α_t can be computed by using the weight W_t^α of the attention layer at time step t as follows: $\alpha_t = \tanh(W_t^\alpha \cdot h_t)$. The sum of all hidden states produces the prediction output: $pred = \text{softmax}\ (W \cdot (\sum_{t=0}^{t=T} \alpha_t \cdot h_t))$.

For evaluation, the authors use a HDFS dataset [36] which contains around 24 million log messages with about 2.9% labelled anomalies and is commonly used for benchmarking. They simulate processing noise by applying various small changes to the order of log sequences (unstable sequences) and the evolution of log events, by randomly removing or adding words from log events (unstable events). The resulting datasets can be seen in table 5.1. For an injection ratio of 5% on *NewTesting1*, LogRobust has a precision of

1.00, recall of 0.91 and an F1-Score of 0.95. Increasing the injection ratio in 5% steps decreases the F1-Score by roughly 2 percentage points, while the baseline methods degrade more. The results are similar on *NewTesting2*. On the unmodified HDFS dataset, precision is 0.98, recall 1.00 and F1-Score 0.99.

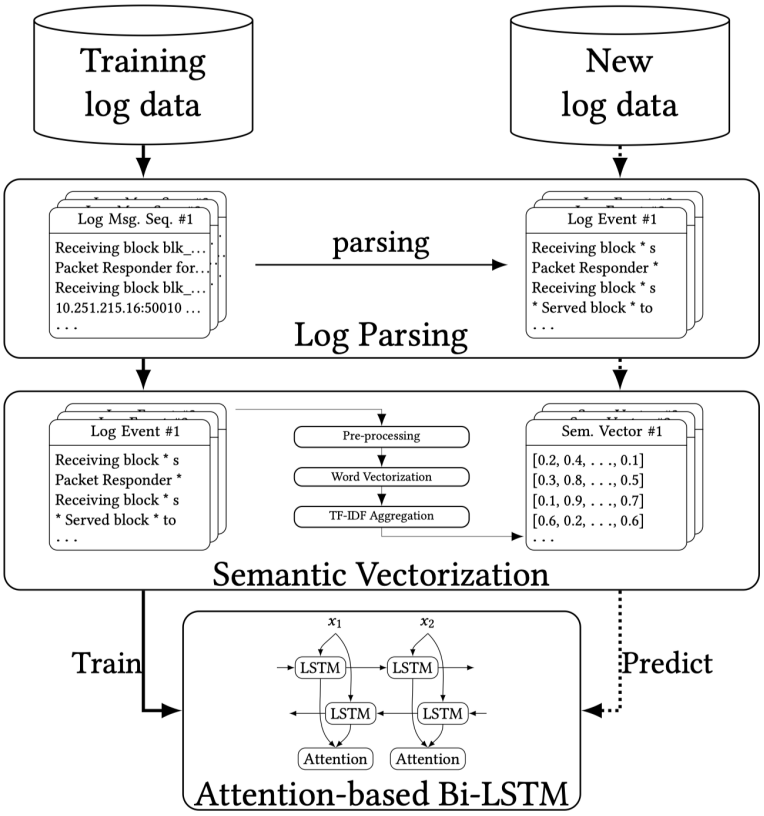


Figure 5.4: Overview of LogRobust [5].

Set	Unstable event	Unstable seq.	Normal	Anomaly	Total
Training	No	No	6,000	6,000	12,000
NewTesting1	Yes	No	50,000	1,000	51,000
NewTesting2	No	Yes	50,000	1,000	51,000

Table 5.1: Manipulated HDFS dataset

6 Conclusion

6.1 Summary

This work addresses the problem of log anomaly detection in large-scale distributed systems such as the cloud, as an essential part of a utility for improving the reliability and the security of a complex system. The generalisation problem for anomaly detection on unseen logs has been addressed by introducing a language model agnostic framework which is able to use pre-trained language models for obtaining log vector representations as a log representation technique. An Bi-LSTM neural network is then used as a method for anomaly detection. With this, the log message semantics and the sequential nature of the log messages are captured. It is shown, that the proposed model is more robust to alteration in the log messages – scenarios frequently occurring in practice due to software updates, deployment of new services or systems. The results show that the framework can achieve high F1-scores and precision, using the state-of-the-art language models. Furthermore, we show that not every representation is equally useful for anomaly detection as some of the language models fail to generate log representations that can be separated by a learned decision boundary.

The proposed approach opens new potential for anomaly detection not just from log data, but from other sources that have the notion of a distributed representation of an event e.g., distributed tracing data. The proposed method will hopefully motivate further research in the direction of development of pre-trained language models on logs. This would enhance the log repersentations, and thus, improve the performance of the anomaly detection methods.

6.2 Outlook

In order to have more means to evaluate different language models, next to the regression and classification approach, a binary classification approach could be implemented.

The word embeddings are taken from the used language models as they are. Fine-tuning on the log corpora is only conducted using the default tasks that have been used to train the language model on large corpora. The corpora on log data are likely to be too small. Even though most log events are sentences in English, they use very reduced idioms. A deeper investigation on how to train the language model specifically

on the task of anomaly detection could potentially be useful in order to obtain better results. Integrating the fine-tuning step into the model's pipeline, can potentially improve the quality of the word embeddings. This can be especially useful for the transfer of knowledge task.

A very important step in order to make the proposed model even more robust, resilient and most importantly able to transfer knowledge on new datasets, evaluation into implementing an attention mechanism, as described by Vaswani et al. [19], that fits the use-case correctly and enhances it. The mentioned papers in 5 have proven that using an attention mechanism in combination with a LSTM can highly improve the results obtained from the model.

Bibliography

[1] W. Meng, Y. Liu, Y. Zhu, S. Zhang, D. Pei, Y. Liu, Y. Chen, R. Zhang, S. Tao, P. Sun, *et al.*, "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*, vol. 7, pp. 4739–4745, 2019.

[2] S. Kabinna, C.-P. Bezemer, W. Shang, M. D. Syer, and A. E. Hassan, "Examining the stability of logging statements," *Empirical Software Engineering*, vol. 23, no. 1, pp. 290–333, 2018.

[3] M. Du, F. Li, G. Zheng, and V. Srikumar, "Deeplog: Anomaly detection and diagnosis from system logs through deep learning," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1285–1298, 2017.

[4] K. Zhang, J. Xu, M. R. Min, G. Jiang, K. Pelechrinis, and H. Zhang, "Automated it system failure prediction: A deep learning approach," in *2016 IEEE International Conference on Big Data (Big Data)*, pp. 1291–1300, IEEE, 2016.

[5] X. Zhang, Y. Xu, Q. Lin, B. Qiao, H. Zhang, Y. Dang, C. Xie, X. Yang, Q. Cheng, Z. Li, *et al.*, "Robust log-based anomaly detection on unstable log data," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 807–817, 2019.

[6] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, *et al.*, "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[7] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM computing surveys (CSUR)*, vol. 41, no. 3, pp. 1–58, 2009.

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

[9] A. Faul, *A Concise Introduction to Machine Learning*. Chapman and Hall/CRC, 2019.

[10] P. Sibi, S. A. Jones, and P. Siddarth, "Analysis of different activation functions using back propagation neural networks," *Journal of Theoretical and Applied Information Technology*, vol. 47, no. 3, pp. 1264–1268, 2013.

[11] A. Graves, A.-r. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *2013 IEEE international conference on acoustics, speech and signal processing*, pp. 6645–6649, IEEE, 2013.

[12] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer, "Deep contextualized word representations," *arXiv preprint arXiv:1802.05365*, 2018.

[13] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber, "Lstm: A search space odyssey," *IEEE transactions on neural networks and learning systems*, vol. 28, no. 10, pp. 2222–2232, 2016.

[14] A. Graves and J. Schmidhuber, "Framewise phoneme classification with bidirectional lstm and other neural network architectures," *Neural networks*, vol. 18, no. 5-6, pp. 602–610, 2005.

[15] "Understanding lstm networks." `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`. Accessed: 2020-06-10.

[16] D. W. Otter, J. R. Medina, and J. K. Kalita, "A survey of the usages of deep learning for natural language processing," *IEEE Transactions on Neural Networks and Learning Systems*, 2020.

[17] "Mit technology review: King - man + woman = queen: The marvelous mathematics of computational linguistics." `https://www.technologyreview.com/2015/09/17/166211/king-man-woman-queen-the-marvelous-mathematics-of-computational-linguistics/`. Accessed: 2020-06-11.

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

[19] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, pp. 5998–6008, 2017.

[20] "Bert." `https://github.com/google-research/bert`. Accessed: 2020-05-17.

[21] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang, "Squad: 100,000+ questions for machine comprehension of text," *arXiv preprint arXiv:1606.05250*, 2016.

[22] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman, "Glue: A multi-task benchmark and analysis platform for natural language understanding," *arXiv preprint arXiv:1804.07461*, 2018.

[23] P. He, J. Zhu, S. He, J. Li, and M. R. Lyu, "Towards automated log parsing for large-scale log data analysis," *IEEE Transactions on Dependable and Secure Computing*, vol. 15, no. 6, pp. 931–944, 2017.

[24] J. Zhu, S. He, J. Liu, P. He, Q. Xie, Z. Zheng, and M. R. Lyu, "Tools and benchmarks for automated log parsing," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pp. 121–130, IEEE, 2019.

[25] G. Lee, J. Lin, C. Liu, A. Lorek, and D. Ryaboy, "The unified logging infrastructure for data analytics at twitter," *arXiv preprint arXiv:1208.4171*, 2012.

[26] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier, "Using magpie for request extraction and workload modelling.," in *OSDI*, vol. 4, pp. 18–18, 2004.

[27] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch, "The mystery machine: End-to-end performance analysis of large-scale internet services," in *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pp. 217–231, 2014.

[28] 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)*, pp. 33–40, IEEE, 2017.

[29] A. A. Makanju, A. N. Zincir-Heywood, and E. E. Milios, "Clustering event logs using iterative partitioning," in *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 1255–1264, 2009.

[30] "University of utah, datasets for the deeplog paper.." `https://zenodo.org/record/3227177/files/OpenStack.tar.gz?download=1`. Accessed: 2020-08-17.

[31] "Cloudlab. flexible, scientific infrastructure for research on the future of cloud computing." `https://web.archive.org/web/20190306043920/https://cloudlab.us/`. Accessed: 2020-06-11.

[32] Y. Liang, Y. Zhang, H. Xiong, and R. Sahoo, "Failure prediction in ibm bluegene/l event logs," in *Seventh IEEE International Conference on Data Mining (ICDM 2007)*, pp. 583–588, IEEE, 2007.

[33] C. Yuan, N. Lao, J.-R. Wen, J. Li, Z. Zhang, Y.-M. Wang, and W.-Y. Ma, "Automated known problem diagnosis with event traces," *ACM SIGOPS Operating Systems Review*, vol. 40, no. 4, pp. 375–388, 2006.

[34] M. Farshchi, J.-G. Schneider, I. Weber, and J. Grundy, "Anomaly detection of cloud application operations using log and cloud metric correlation analysis," ISSRE, 2015.

[35] M. Chen, A. X. Zheng, J. Lloyd, M. I. Jordan, and E. Brewer, "Failure diagnosis using decision trees," in *International Conference on Autonomic Computing, 2004. Proceedings.*, pp. 36–43, IEEE, 2004.

[36] 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*, pp. 117–132, 2009.

[37] Q. Lin, H. Zhang, J.-G. Lou, Y. Zhang, and X. Chen, "Log clustering based problem identification for online service systems," in *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pp. 102–111, IEEE, 2016.

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

[39] P. Malhotra, L. Vig, G. Shroff, and P. Agarwal, "Long short term memory networks for anomaly detection in time series," in *Proceedings*, vol. 89, Presses universitaires de Louvain, 2015.

[40] C. Bertero, M. Roy, C. Sauvanaud, and G. Trédan, "Experience report: Log mining using natural language processing and application to anomaly detection," in *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 351–360, IEEE, 2017.

[41] A. Brown, A. Tuor, B. Hutchinson, and N. Nichols, "Recurrent neural network attention mechanisms for interpretable system log anomaly detection," in *Proceedings of the First Workshop on Machine Learning for Computing Systems*, pp. 1–8, 2018.

[42] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in neural information processing systems*, pp. 3111–3119, 2013.

[43] G. A. Miller, "Wordnet: a lexical database for english," *Communications of the ACM*, vol. 38, no. 11, pp. 39–41, 1995.

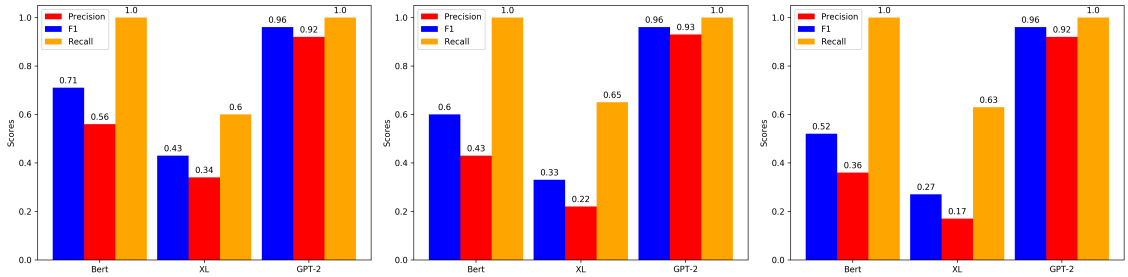
[44] K. A. Nguyen, S. S. i. Walde, and N. T. Vu, "Integrating distributional lexical contrast into word embeddings for antonym-synonym distinction," *arXiv preprint arXiv:1605.07766*, 2016.

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

Annex

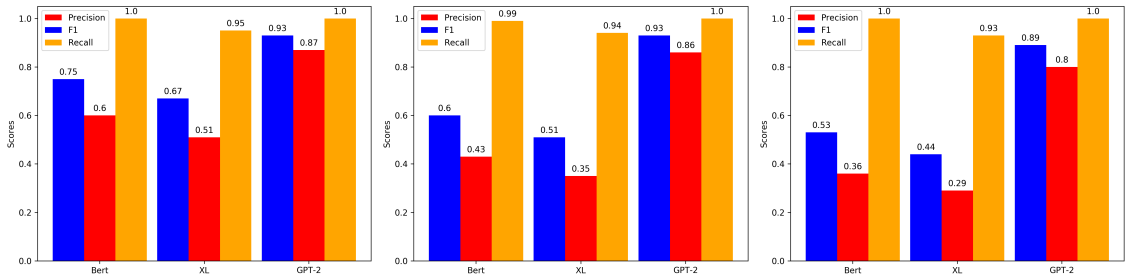
.1 Regression

.1.1 Sequence-based injections



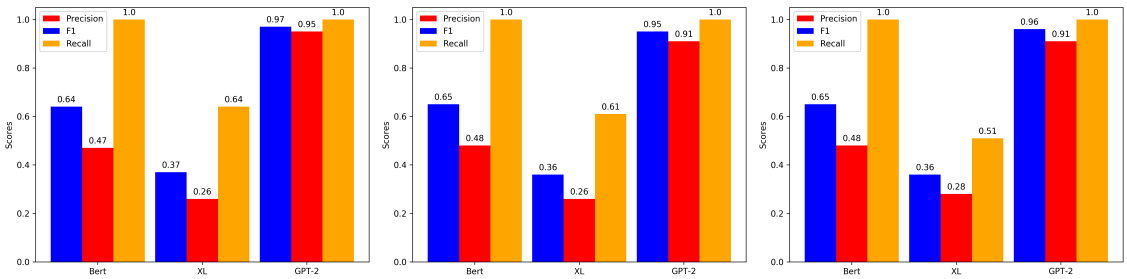
(a) 5% alteration (b) 10% alteration (c) 15% alteration

Figure .1: Delete lines at different ratios using, regression.



(a) 5% alteration (b) 10% alteration (c) 15% alteration

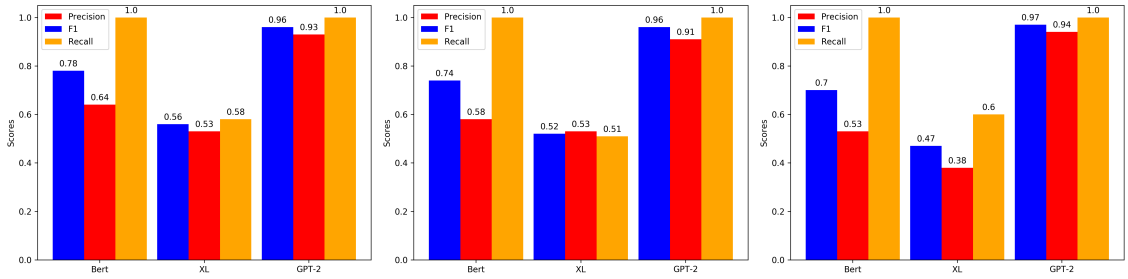
Figure .2: Duplicate lines at different ratios, using regression.



(a) 5% alteration (b) 10% alteration (c) 15% alteration

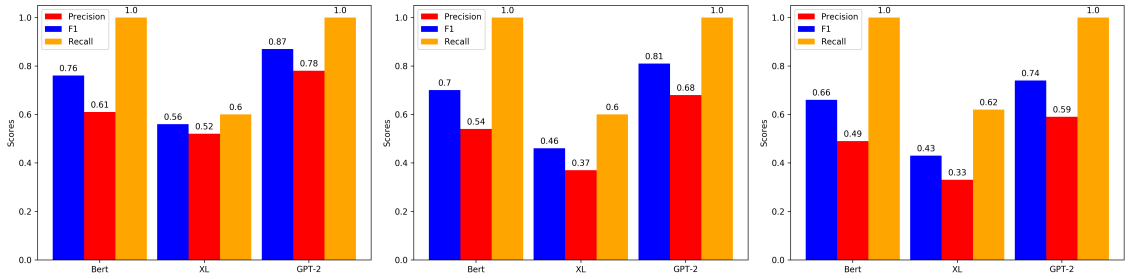
Figure .3: Shuffle lines at different ratios, using regression.

.1.2 Injections on log events



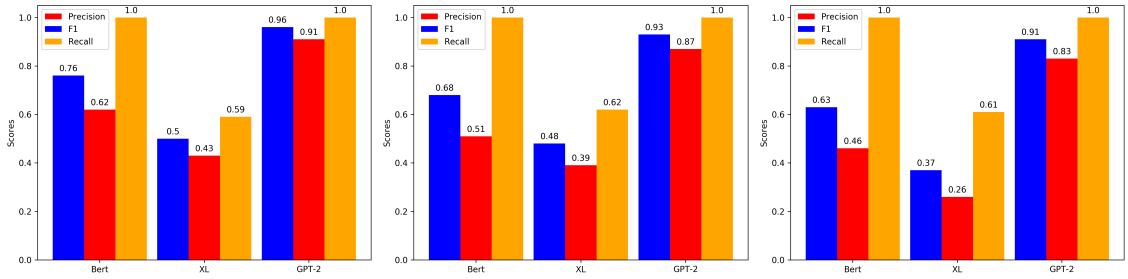
(a) 5% alteration (b) 10% alteration (c) 15% alteration

Figure .4: Insert words at different ratios, using regression.



(a) 5% alteration (b) 10% alteration (c) 15% alteration

Figure .5: Remove words at different ratios, using regression.

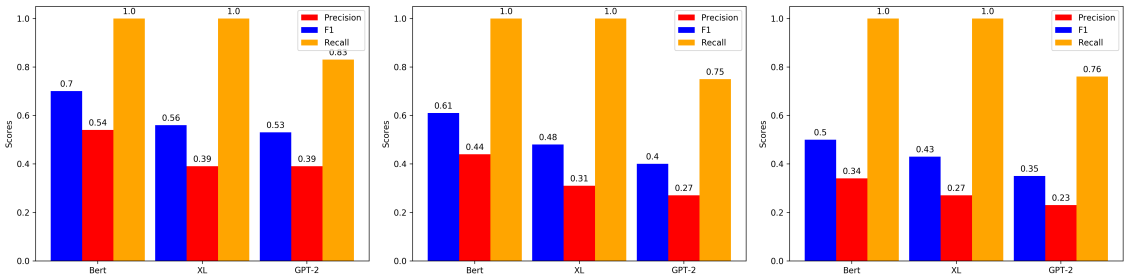


(a) 5% alteration (b) 10% alteration (c) 15% alteration

Figure .6: Replace words at different ratios, using regression.

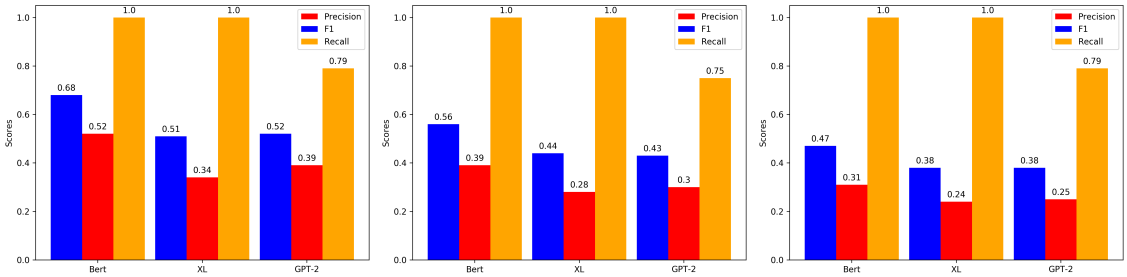
.2 Classification

.2.1 Sequence-based injections



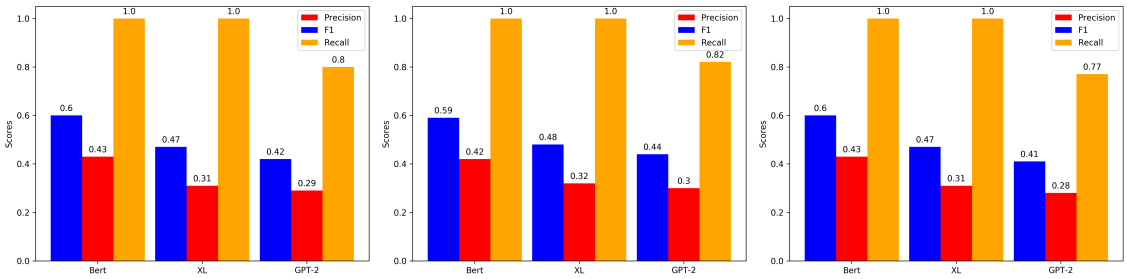
(a) 5% alteration (b) 10% alteration (c) 15% alteration

Figure .7: Delete lines at different ratios using, regression.



(a) 5% alteration (b) 10% alteration (c) 15% alteration

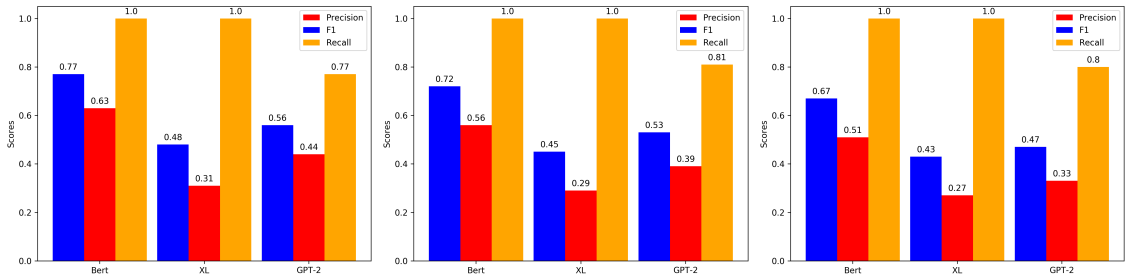
Figure .8: Duplicate lines at different ratios, using regression.



(a) 5% alteration (b) 10% alteration (c) 15% alteration

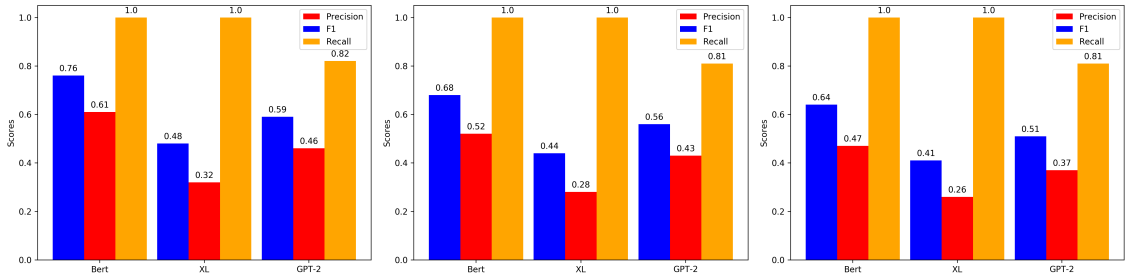
Figure .9: Shuffle lines at different ratios, using regression.

.2.2 Injections on log events



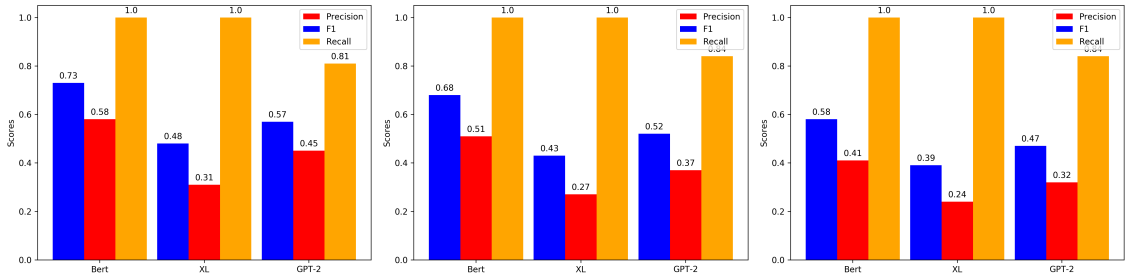
(a) 5% alteration (b) 10% alteration (c) 15% alteration

Figure .10: Insert words at different ratios, using regression.



(a) 5% alteration (b) 10% alteration (c) 15% alteration

Figure .11: Remove words at different ratios, using regression.



(a) 5% alteration (b) 10% alteration (c) 15% alteration

Figure .12: Replace words at different ratios, using regression.