Technische Universität Berlin

Department of Telecommunication Systems
Distributed and Operating Systems

Fakultät IV
Franklinstrasse 28-29
10587 Berlin

Master Thesis

Anomaly Detection in Infrastructure-agnostic Cloud Environments

Harald Ott

Matriculation Number: 367884

August 9, 2020

Supervised by
Prof. Dr. Odej Kao

Assistant Supervisor
Sasho Nedelkoski and Alexander Acker

Hereby I declare that I wrote this thesis myself with the help of no more than the mentioned literature and auxiliary means.

Berlin, August 9, 2020

. . . .

(Signature [your name])

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 detection of anomalies in system logs is crucial in order to ensure the reliability and security of a cloud system. While recent studies focused on inflexible solutions which are non-portable, due to their inability to learn the semantic meaning of logs, this work proposes a model, which uses state-of-the-art language models in order to capture the linguistic nature of system log data.

Zusammenfassung

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, with the complete outage of the paying system of an E-Commerce shop for example, potentially resulting 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 actions that have been performed, but can also indicate the state of a system at a given point in time. 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 explicitly as errors, 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]. Traditional anomaly detection based on standard mining technologies cannot cope with the complex nature of anomalies in modern systems [2], since they are constantly evolving, which would require constant adjustments. Therefore, a more general approach is desirable.

1.2 Scope

The scope of this work is to find an appropriate way to represent dynamically changing log events using language models, and to assess their quality with regards to their ability of being utilised for solving the problem of anomaly detection. Therefore, an approach using an auto encoder to learn fixed length representations of log events that have been transformed into word embeddings with GloVe, is presented as baseline. Next, an approach to utilising word embeddings obtained from Bert, GPT-2 and XL-Transformers as input for a LSTM to learn error-free log sequences in a both supervised and unsupervised manner. For this purpose, two different ways of mapping the input of multiple log events to a target space are evaluated, namely classification and regression.

1.3 Outline

This master's thesis is separated into 7 chapters.

Chapter 2 presents background knowledge and terms required in order to understand the presented concept.

Chapter 3 presents the formal problem definition, the requirements and assumptions, followed by the developed concept. The complete pipeline is explained in detail, clarifying the necessity of every step in order to assemble the full model. Moreover, constraints of the model and suggestions for possible improvements and further refinement of the model are given.

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 model and a comparison between them is conducted.

Chapter 5 describes related work, pointing out similarities and differences to the presented concept.

In **chapter 6**, the final conclusion 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 2.3 the theoretical foundation of deep neural networks is described, followed by an introduction to LSTMs. In section 2.2 techniques to tackle the problem of anomaly detection in general are presented. 2.4 gives insights on how natural language can be represented using language models. Finally, in section 2.5.

2.1 Cloud Computing

The term *cloud computing* usually refers to hardware and systems software in large data centres that provide a platform for applications delivered as services over the Internet. Due to the possibilities offered by clouds that are available in a pay-as-you-go manner, businesses with new ideas do not require enormous amounts of prefinancing in 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 [3].

Virtualisation plays a vital role in modern clouds, 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 Amazon AWS or Microsoft Azure offer a vast number of different useful services, that 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.

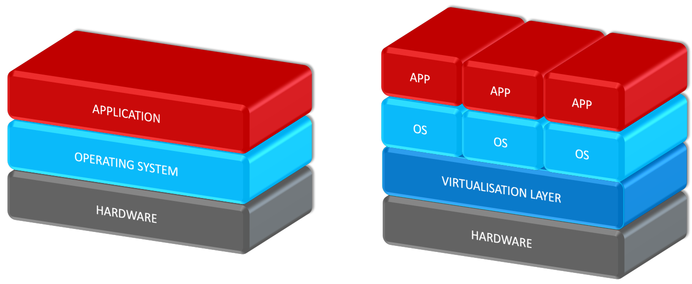


Figure 2.1: Traditional architeture vs. virtualised architecture

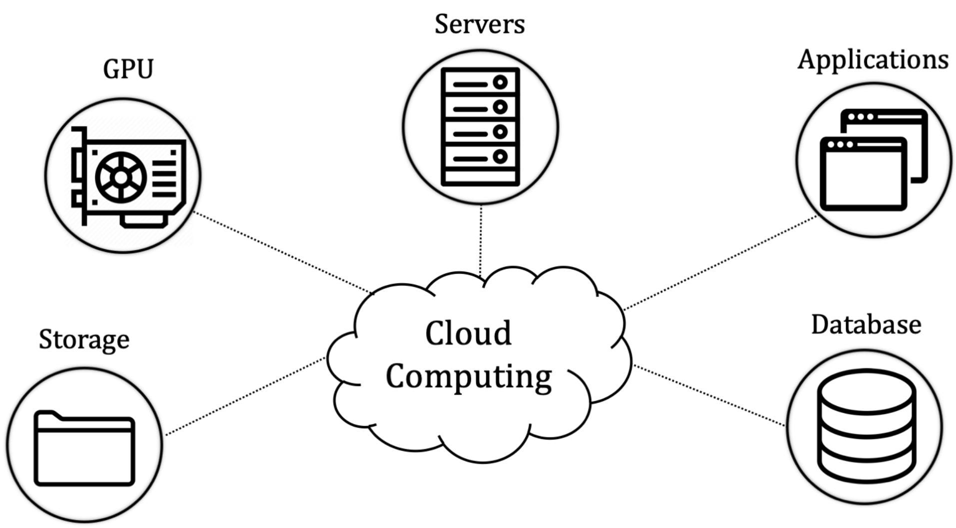


Figure 2.2: Cloud computing services

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 of 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 that are sufficiently far away, like the points in regions A_1 and A_2 are considered anomalies. But consider also the points B_1. Should they be marked as normal or abnormal? Are the borders around D_1 and D_2 correct, or are they not sufficiently broad, due to the lack of enough normal 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 [4].

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 [4]. Figure 2.3 outlines the central steps involved in finding an appropriate anomaly detection technique to a given problem.

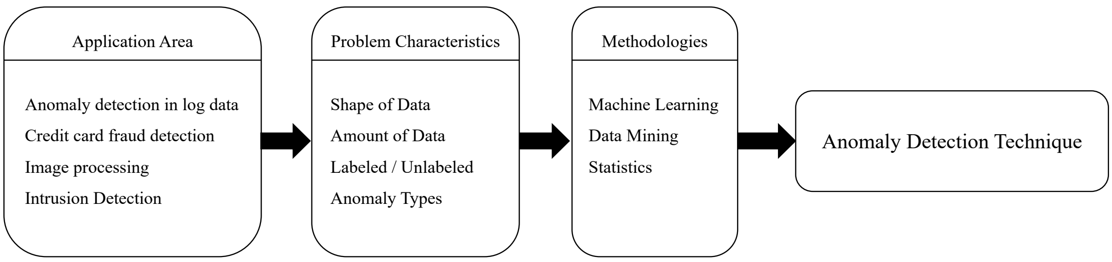


Figure 2.3: Schema of the development of an anomaly Detection technique [4].

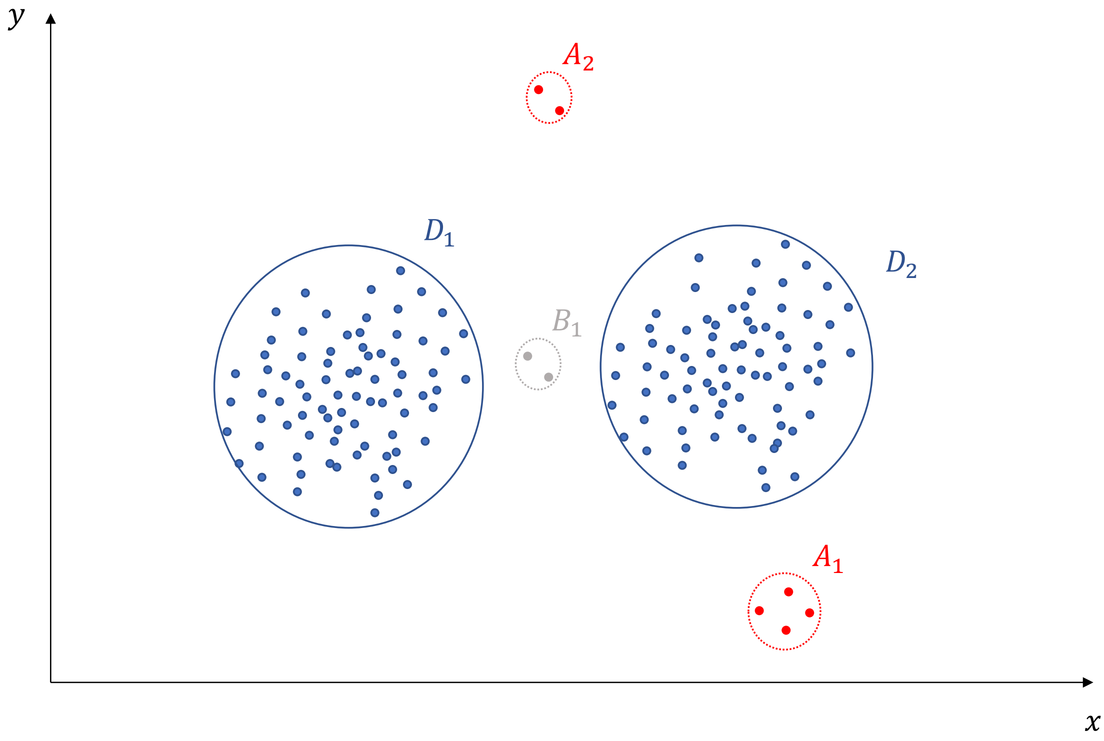


Figure 2.4: Example of anomalies in a dataset.

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 data structures within a neural network. Deep learning can outperform traditional machine learning techniques as the amount of data increases [5]. Section 2.3.1 gives an introduction to neural network, 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 [7]. Figure 2.5 is an illustration of a classical feed-forward neural network. It consist 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 functions and the *output* neurons, depicted as y_k gather given predictions and compute

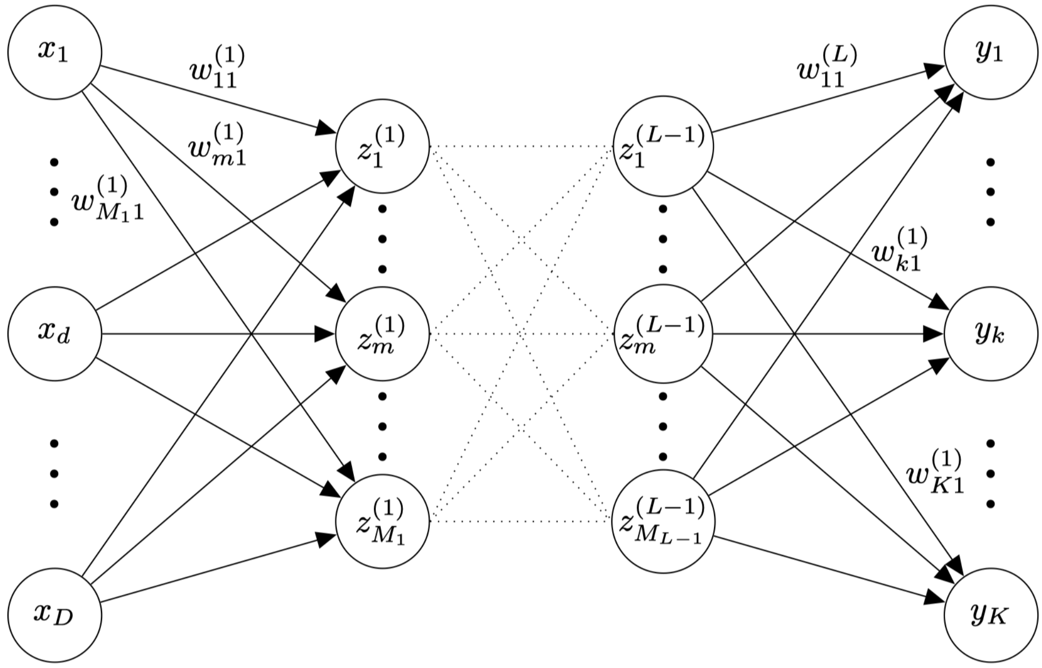


Figure 2.5: A neural network [6].

the output [6]. 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 activation 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 2.2. There exists a variety of activation functions, depending on the use case. Activation functions transform the activation of output neurons as in 2.1 into an output signal [7]. 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 [6].

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

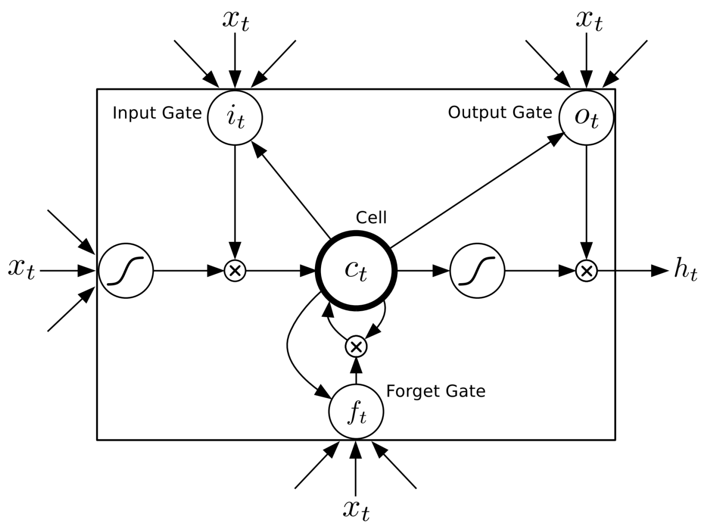


Figure 2.6: A LSTM block [8]

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 [9], anomaly detection [2] amongst others.

The core of the LSTM architecture is memory cell which maintains its state over time, in combination with gating units, which control the information flow [10], allowing it to remove or add information. The traditional LSTM architecture was first described by [11]. 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 with 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, 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 [12] [8].

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

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

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

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

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

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

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. [8]. In this way, it is possible to make us of past features via the forward state and future features via the backward states. Forward and backward passes are executes similarly to

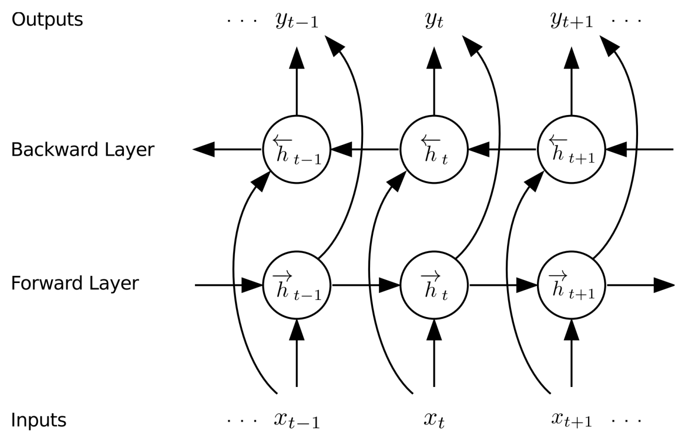


Figure 2.7: A Bi-directional LSTM [8].

The forward and backward passes over the unfolded network over time are carried out in a similar way to regular network forward and backward passes, except that we need to unfold the hidden states for all time steps. We also need a special treatment at the beginning and the end of the data points. In our implementation, we do forward and backward for whole sentences and we only need to reset the hidden states to 0 at the begging of each sentence.

2.4 Natural Language Processing

Since raw log data cannot be processed without further ado by a LSTM, 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 that 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* (find segments within words), *syntactic processing* (how 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* [13]. 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 [14]. 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. [15]. 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 [16].

Training includes two steps: *pre-training* and *fine-tuning*. Pre-training involves training on unlabelled data over various pre-training tasks. Finetuning then uses the weights initialised with the parameters obtained from pre-training, re-calibrating them using 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` 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 [17], 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. [18], a collection of 100k crowd-sourced question/answer pairs, or the General Language Understanding Evaluation (GLUE) benchmark by Wang et al. [19], which involves various natural language understanding tasks.

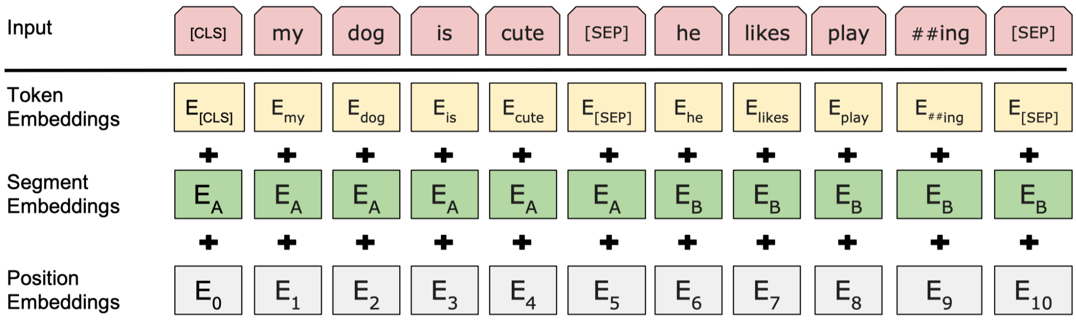


Figure 2.8: Bert

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 [20][21]. 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 is then produced 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 [22] or workload modelling [23]. Logs can also be used as data sources for performance modelling [24] 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 [21].

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 [25]. 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 [21].
- 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 [26]. 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 [25].

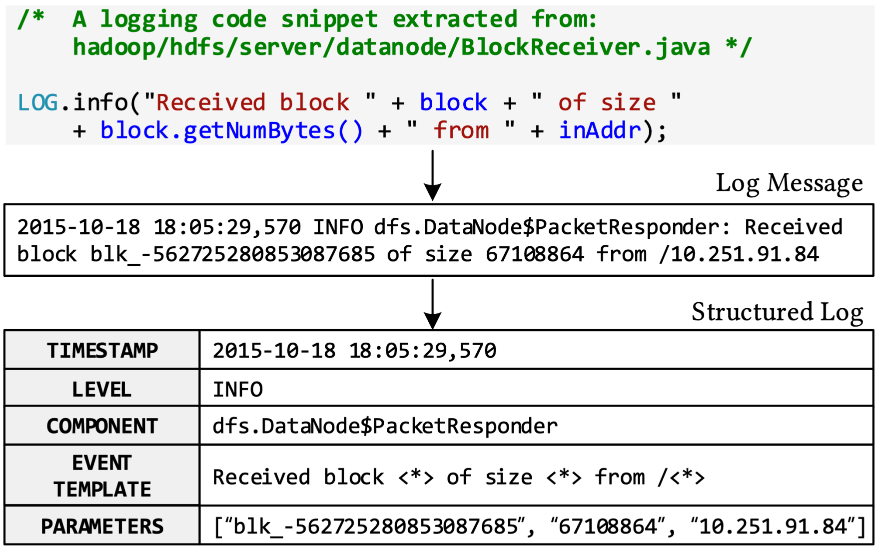


Figure 2.9: Schematic execution of log parsing [21]

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 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 (1) regression and the (2) classification approach, both using log event representations obtained by a language model. In section 3.5 the transfer learning mechanism is described. In the final section **??** possible improvements of the proposed approach are presented.

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. These logs must be first brought into an appropriate form, in .

3.1.1 Formal problem definition

A log is a sequence of ASCII characters, which is denoted by the set $\mathcal{A}$ that form unstructured messages $M = (m_0, m_1, ..., m_n)$ with every character of $m_i \in \mathcal{A}$. Log messages consist of tokens - most tokens are English words, but do also include special characters. The number of tokens from which a log message can consist of, varies. Let $g : \mathcal{A}^i \to \mathbb{R}^w$ be a function that takes a variable length of i tokens that make up a log message and maps them to a fixed length vector of dimension w. This is the function which represents the computation of embeddings. $E = (e_0, e_1, ..., e_n)$ is the sequence of vectors based on M, parsed and transformed into embedding representations. Let $\mathcal{B} = (b_0, b_1, ..., b_n)$ be the dataset where every $b_i \in 0, 1$ denotes if the log event represented by e_i is anomalous (1) or not (0). $\hat{b_i}$ denotes the system's prediction for b_i.

Classification: For every distinct log template, we assign $C = \{c_0, c_1, ..., c_k\}$ to be the set of distinct class keys. Let $g : |s| \times \mathbb{R}^w \to \mathbb{N},\ g(E_i, \Psi) = \mathbb{R}^k$ 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 subsequent class. If one of the top q candidate classes matches the actual class, then $\hat{b_i} = 0$ is returned, if it does not match the actual class, $\hat{b_i} = 1$ is returned.

Regression: Let $h : |s| \times \mathbb{R}^w \to \mathbb{R}^w,\ h(e_i, s, \Phi) = d$ be a function computed by a neural network, that based on a sequence of s vectors $E = (e_j, ..., e_{j+s})$ predicts the vector d at index $j + s + 1$. 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_{j+s+1} is above or below certain threshold values, which will be computed by the q-th percentile of all loss values from the original dataset, then $\hat{b_i} = 1$ is returned, if it is inside these thresholds, then $\hat{b_i} = 0$ 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* log data, which do not contain anomalies for training.

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

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 classification. The core concept can be outlined as follows: first, original log sequences are prepared, then a log parser is used to extract templates from the original log sequences and then transform the log sequences to template sequences. Afterwards, the template sequences are transformed into log sequence embeddings by a language representation model. This procedure is described in detail in section 3.3.

For the training part, the log sequence embeddings are fed into a LSTM, which learns to predict the next embedding, which is called the regression task, specified in section 3.4.3 and coloured red in figure 3.1 or the next sequence class, which is called the classification task, specified in 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 "normal" 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 which has been trained on embedding sequences not containing any alterations, thereby obtaining the model's predictions if a log line is anomalous or not.

Transfer learning 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 template sequences and embedding sequences. 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 transfer. Transfer learning is described in detail in section 3.5

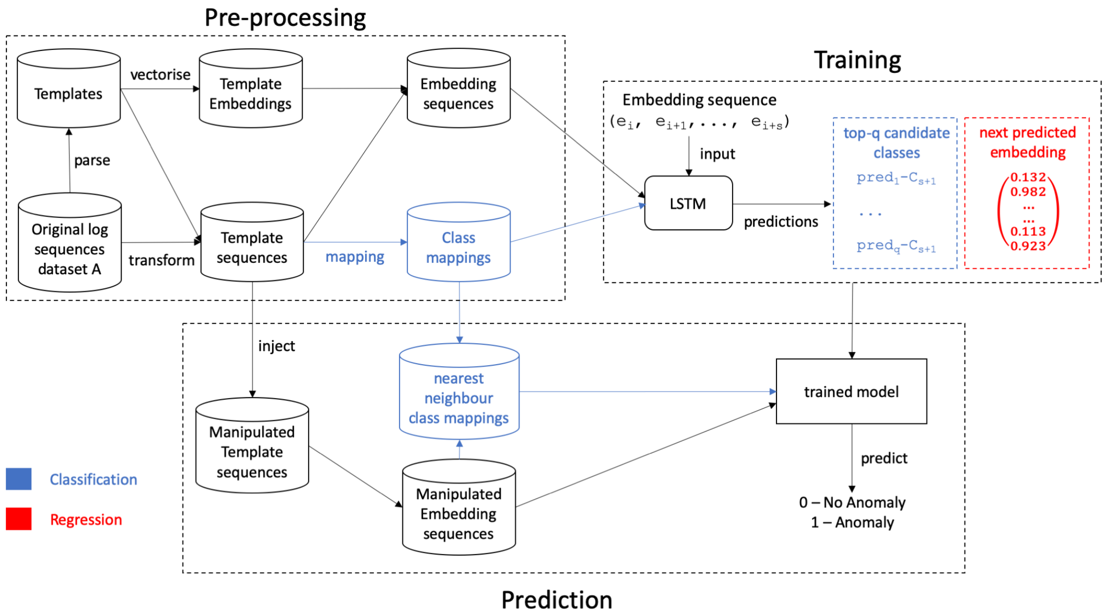


Figure 3.1: Anomaly Detection System

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 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 are outlined, followed by the transformation into word embedding vectors in 3.3.3. A diagram illustrating the complete pre-processing pipeline can be see in figure 3.10.

3.3.1 Log Parsing

Log parsing is an important step for automated log analysis, as already described in section 2.5, since raw log messages are unstructured data and contain a lot of extra information. The result of the execution of a log parser can be see in figure 2.9. There are a few important aspects to note here: Not only does the log parsing step extract log templates, 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 in the example figure 2.9 in order to be able to observe each self-contained block separately from other blocks.

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.

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 g 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 suited 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`" is changed to "`<*> Image created successfully`" or "`VM up`" is changed to "`VM started`". This in turn should result in a low cosine distance [27].

In oder to be able to compare the quality 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.

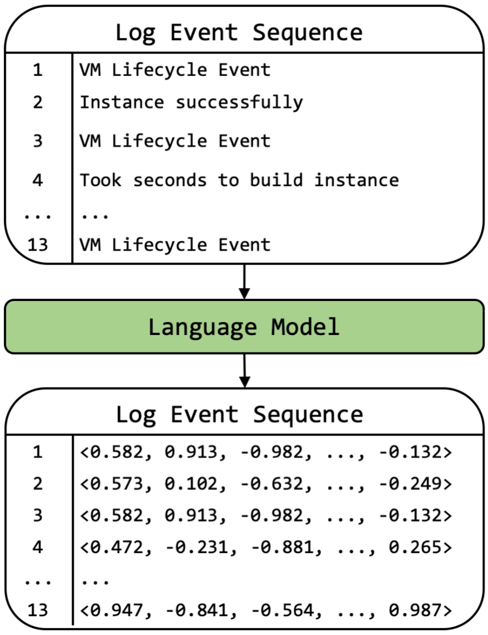


Figure 3.2: Transformation of a log event sequence to a word embeddings sequence.

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 [17]. 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.

3.3.5 Log Alteration

By altering log events, the evolution of log events is being simulated. Since software is changed by developers, also the log statements are subject to constant change. It is desirable to build a flexible model that does not have to be retrained 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 applying transfer learning, as outlined in 3.5.

Injection and alteration is done in a programmatically controllable manner. Various types of alterations are injected into original log data, either on the log event itself as in figure 3.3 or on the sequence of log events as 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"*, but for testing purposes, also words that are random in the context . Words can be also deleted. Finally, words can be replaced by new words. All of these alterations can be injected multiple times into the same log event. It is desirable that the system does not detect a log event as an anomaly, that has not been changed much, i.e. only one or two words have been added into a statement (e.g. if *"* Took *.* seconds to deallocate network for instance."* has been changed to *"* Took time *.* 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 $M_{dup} = (m_0, ..., m_j, m_j, m_{j+1}, ..., m_n)$
- new events can be *inserted* meaning that if the event m_{new} is inserted at index j, the resulting sequence is $M_{ins} = (m_0, ..., m_{new}, m_j, m_{j+1}, ..., m_n)$
- log sequences can be *inverted*, the resulting sequence being $M_{inv} = (m_i : i = n, n-1, ..., 2, 1, 0)$

Just like for the insertion of alterations on the log events themselves, it is desirable of the system not to detect an anomaly for the deletion, duplication or shuffling of events, but for the insertion of anomalies into an event series, and the inversion of log events. These inserted anomaly events can be any type of log events that the system has not seen before.

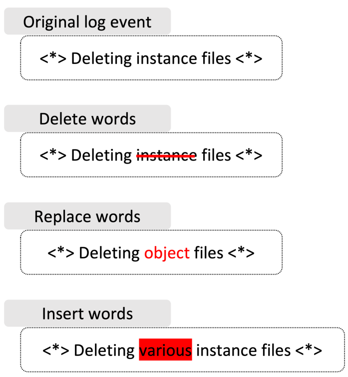


Figure 3.3: Altering log events.

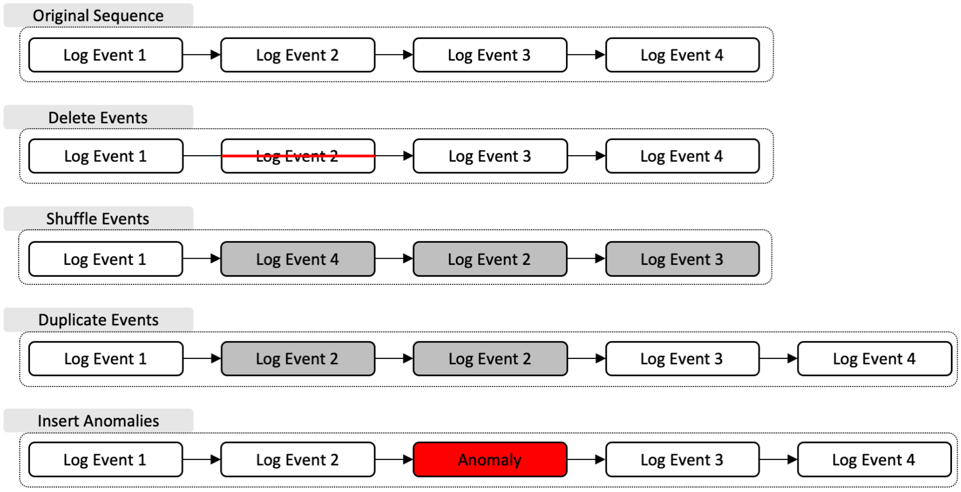


Figure 3.4: Altering log sequences.

3.4 Prediction Model

3.4.1 LSTM Model

Through the aforementioned steps in 3.3, all log events are transformed into 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 $E = (e_i, ..., e_{i+s})$. 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 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 into the desired dimensions, i.e. $\mathbb{R}^w$ for regression and number of classes c for classification. Finally, an activation function f 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}.

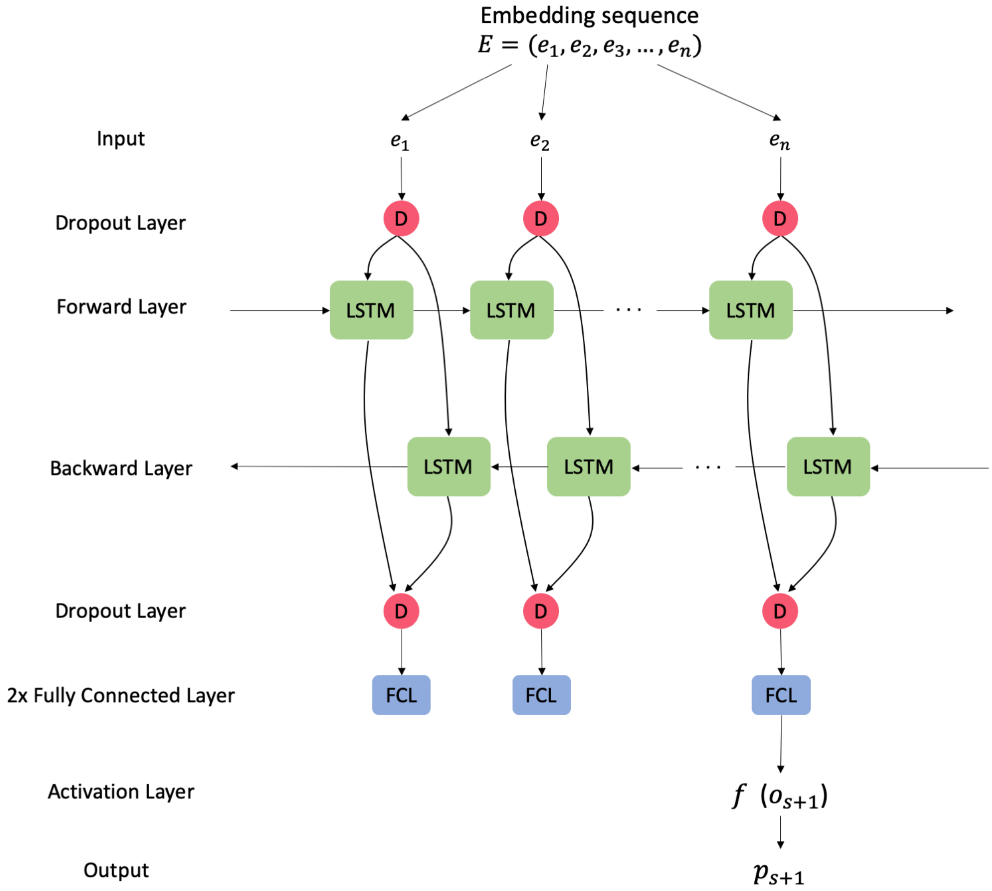


Figure 3.5: Bi-LSTM model

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 subsequence of word embeddings $S_i = (i, ..., i + s)$, 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 word embedding in the test dataset, no corresponding word 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 *"System restarted"* gets mapped to any of the train dataset's template classes, which is not desirable behaviour.

After the matching process is finished, the system can reading the sequences of log events of the test dataset. 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 g candidates. It is marked as anomalous, $\hat{b}_{s+i+1} = 1$ otherwise.

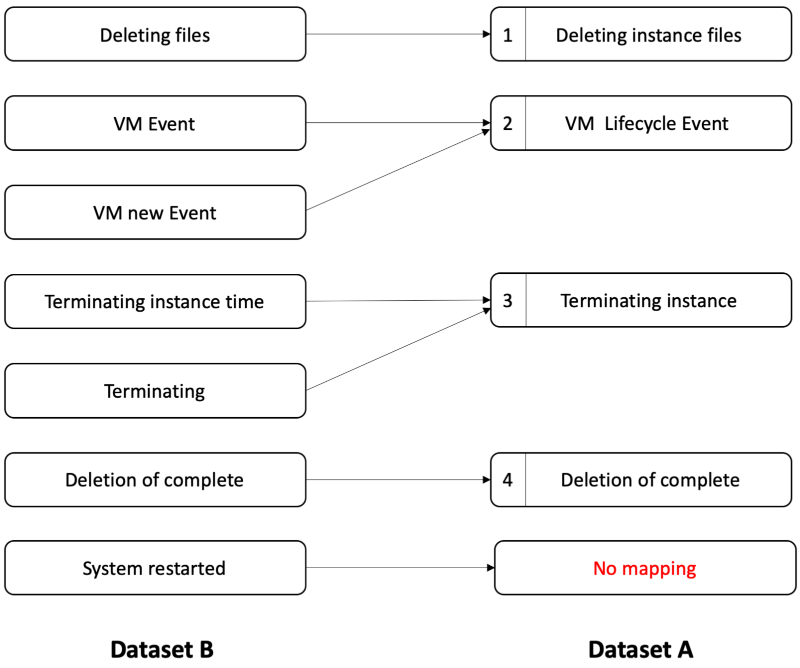


Figure 3.6: Template mapping

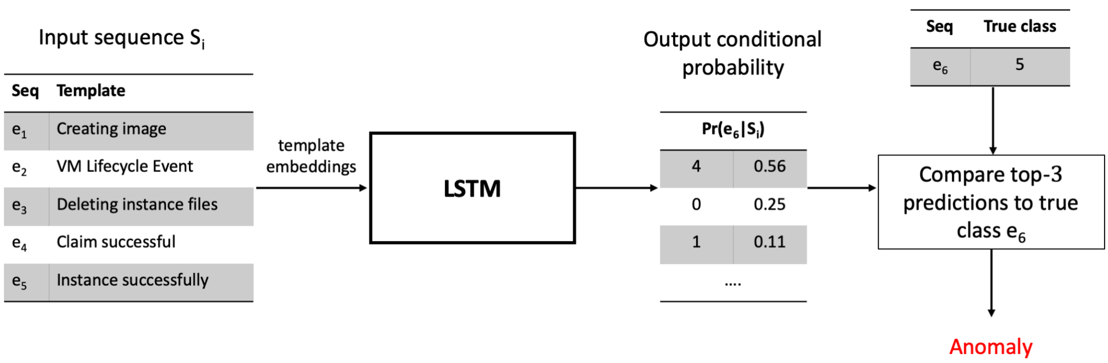


Figure 3.7: Class Prediction example for $g = 3$

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 of lenght s $E_i = (e_i, ..., e_{i+s})$

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 $\hat{b}_i = 0$.

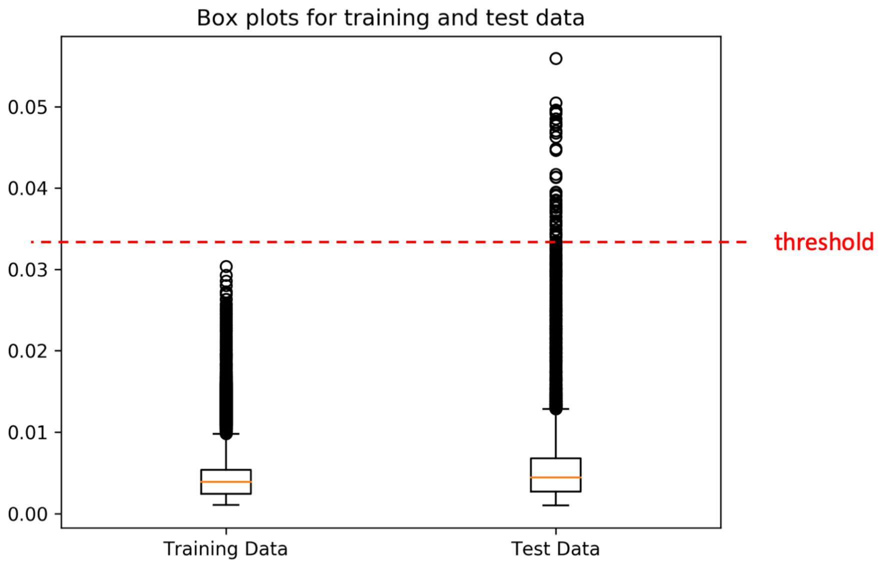


Figure 3.8: Box plots showing the distribution of loss values for training and test data.

3.5 Transfer Learning

Building upon the steps described in the previous subsections, allows to develop an infrastructure-agnostic anomaly detection model. With the help of the described techniques, frequent re-training of models can be avoided and models trained on one dataset can be ported to be used with different datasets. The way transfer learning is executed differs from the classification approach, which is further outlined in 3.5.1 and the regression approach, explained in 3.5.2.

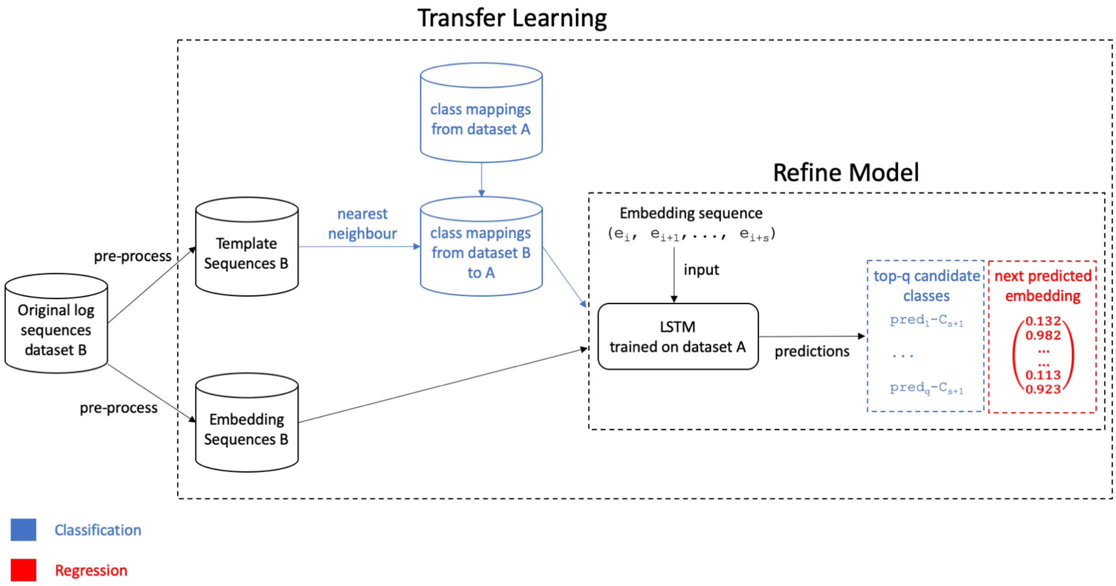


Figure 3.9: Transfer Learning System

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 learning 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.

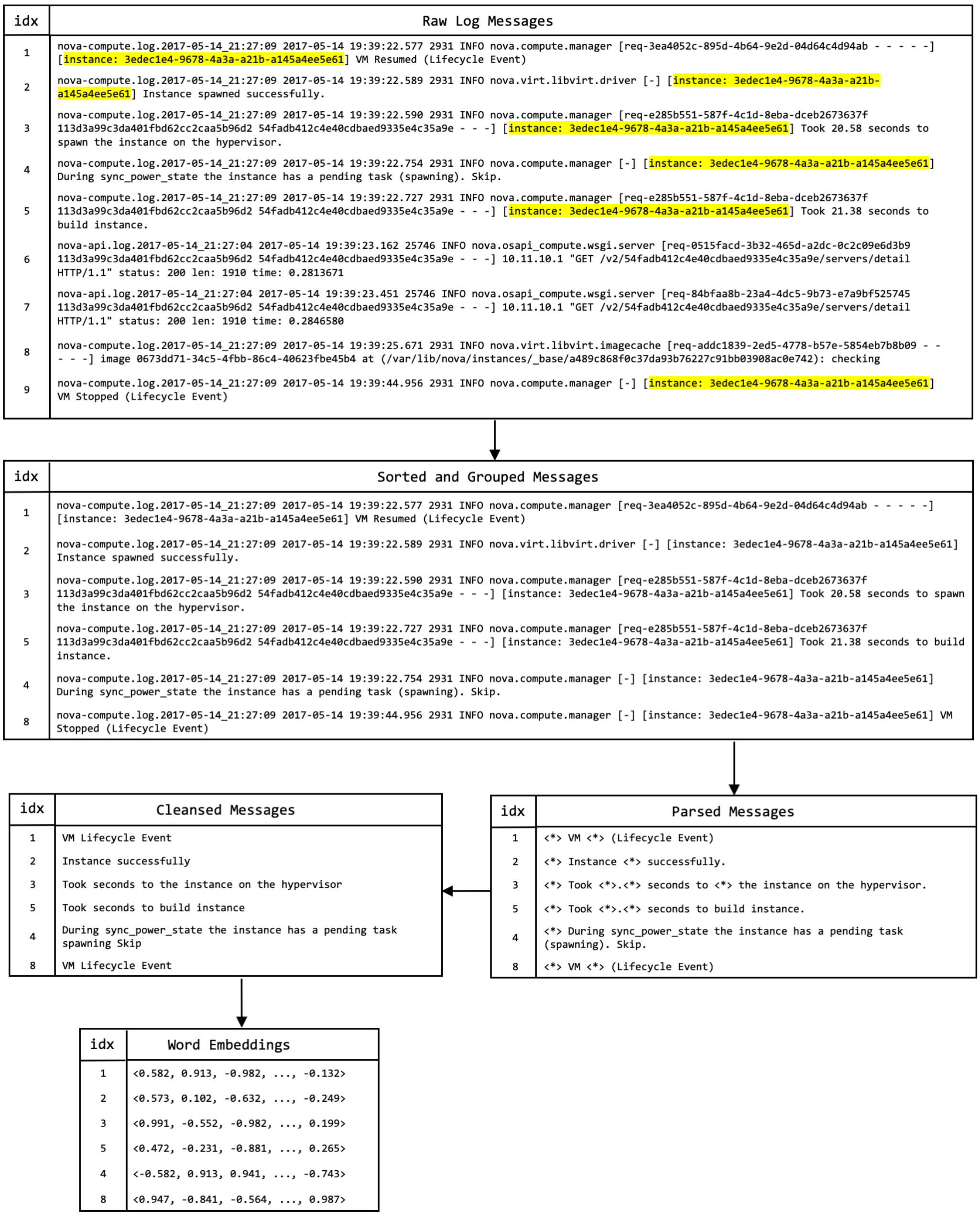


Figure 3.10: One full iteration of the pre-processing pipeline.

4 Results

In the following chapter, the achieved results of the model are presented. The described approaches are evaluated with regards to the common metrics precision, recall, and F1-Score. Different hyperparameter-setups are discussed and visualised in graphs. In particular, the language models Bert, XL-Transformers and GPT-2 as described in 3.3.3 are compared. Additionally, the regression-based method as described in 3.4.3 and the classification method as described in 3.4.2 are compared, showing advantages or disadvantages among each other.

In section 4.1 details about the used hardware and technologies are listed, followed by section 4.2 in which the results of the evaluation are presented. Finally, in section 4.3 the results of the evaluation are discussed.

4.1 Experimental Setup

In this section, the used hardware, libraries, 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 the DeepLog authors at the University of Utah are used [28]. The original dataset stems from logs from running multiple OpenStack instances for 20 hours and 13 minutes for the original dataset and 2 hours and 17 minutes for the test dataset on CloudLab [29]. Since 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, 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. 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 In order to assess the quality of the used word embeddings, log alterations are additionally injected into the dataset containing anomalies, in order to simulate the instability of logs as described in 3.3.5.

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 Learning

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 as utilised in 4.1.2 is used as a basis. First, on the basis of the templates which are presented in table 4.4, the templates which are all altered throughout the dataset can be seen in 4.1, in order to simulate a different dataset. Additionally, various of the alterations discussed in 3.3.5 are injected all at once. For the following experiments, line shuffling, duplication and deletion, and word insertion, removal and replacement are injected into the dataset at various ratios in dataset A. The model is then trained on this dataset A for 60 epochs. As 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 B, i.e. no alterations.

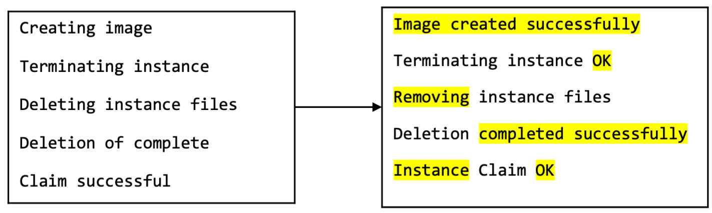


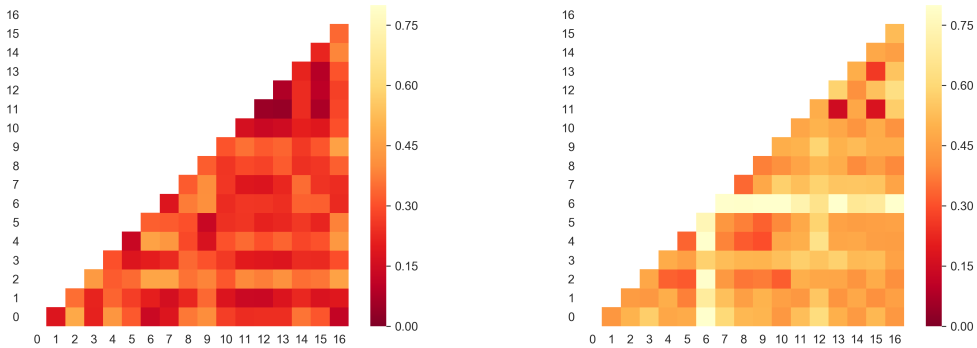
Figure 4.1: Changes on templates of the original dataset.

4.2 Evaluation

In this section, the results of the evaluation between the two anomaly detection approaches, namely 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 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 learning approach are presented.

4.2.1 String cleansing

String cleansing, as described in 3.3.2, can potentially improve distinguishability between templates drastically in the used dataset. For Bert, the pairwise cosine distances before cleansing are depicted in figure 4.2a, after cleansing in figure 4.2b. The corresponding templates to 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



(a) Before cleansing.

(b) After cleansing.

Figure 4.2: Bert pairwise template cosine distance.

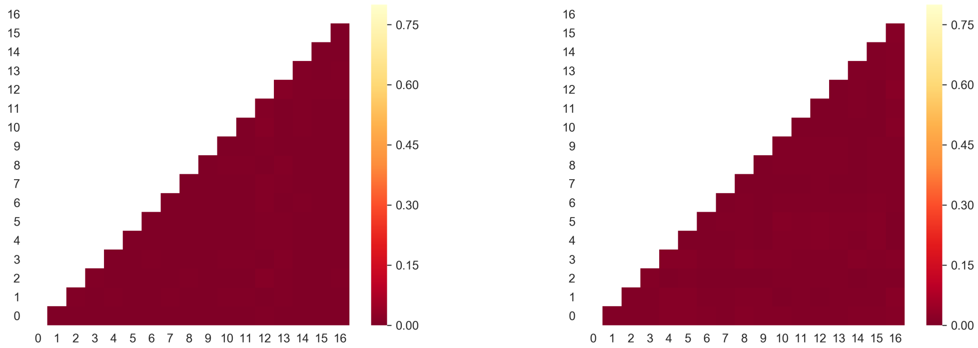
and figure 4.4b, respectively. The average pairwise cosine distances between templates before and after cleansing can be seen in table 4.2. Note that the cosine distances between templates appear to be relatively low for GPT-2 before and after cleansing. Even though cleansing results in a two-fold increase in the distances, the initial value is already very low. We can see how the distinguishability between templates increases for Bert and even more impressively for XL-Transformers. While Bert has a slightly higher average pairwise cosine distance before cleansing, XL-Transformers highly benefits from cleansing with an average value of 0.62. This underlines the importance of this step in order to meet the requirements postulated in 3.3.3, depending on the deployed language models.

Model	Before cleansing	After cleansing
XL	0.2232	0.6223
Bert	0.2416	0.4449
GPT-2	0.0024	0.0047

Table 4.2: Average pairwise template cosine distances.

4.2.2 Finetuning

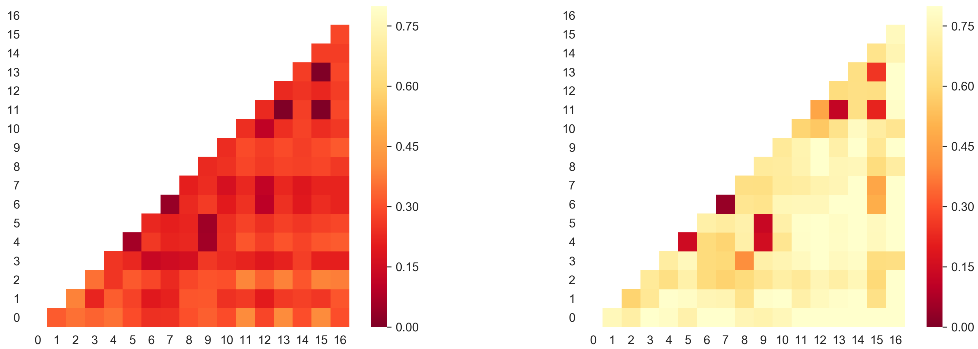
As described in 3.3.4, finetuning can potentially help to produce word embeddings that are more adequate for solving a certain task, it is thus desirable to produce word embeddings that help solving the task of anomaly detection better. As described in 3.3.3, a high cosine distance between semantically different templates is required. The dataset



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

Table 4.3: Templates before cleansing

that consists of the templates in table 4.4 has been chosen for finetuning. Since the pre-trained language models at hand (namely *bert-base-uncased* and *gpt2*) 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 [15], 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 to 0.3016 from 0.4449. The fact that the loss on the evaluation part of the dataset is not decreasing adequately, as shown in figure 4.5, shows that training on such a small corpus is not able 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 out of the box finetuning interfaces for GPT-2 and XL-Transformers on the same task as for Bert, they are not further investigated for finetuning.

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

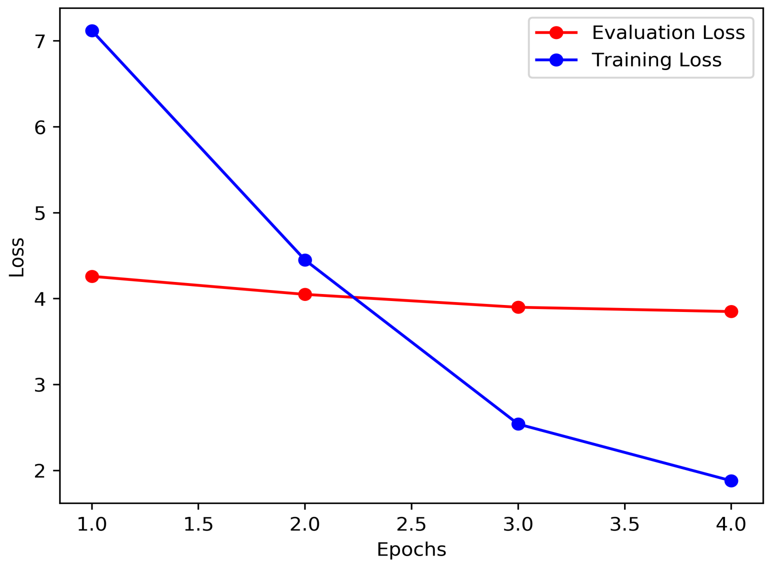


Figure 4.5: Training and evaluation loss for finetuning on masked LM.

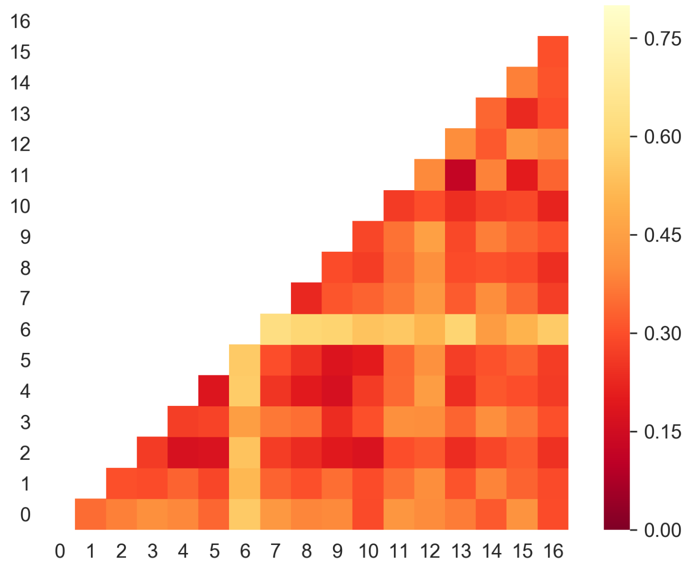


Figure 4.6: Cosine distance between templates after cleansing and finetuning

4.2.3 Hyperparameters

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

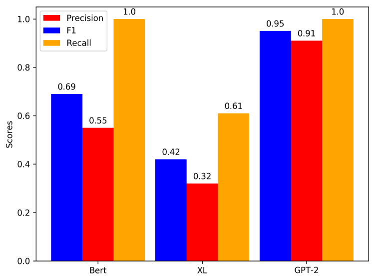
- 512 hidden units for the bidirectional LSTM, with one layer
- two fully connected layers with 512 units → 256 units → output size
- dropout of 0.1 between every layer
- input sequence length of 7
- 60 Epochs of training

4.2.4 Regression-based approach using one dataset

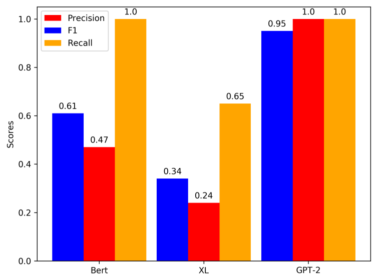
In this subsection, the results of the regression-based approach, with various alterations using the different language models, are presented. In order to evaluate the robustness of the language models to the evolution of log events, alterations as described in 3.3.5 are injected at different ratios. The impact on detecting injected 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. It is evident, that GPT-2 performs better than Bert and XL-Transformers on both the F1-score and precision.

In contrast to the alterations on the log sequences, alterations on the log events themselves, i.e. inserting, removing and replacing words are also of interest. 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.

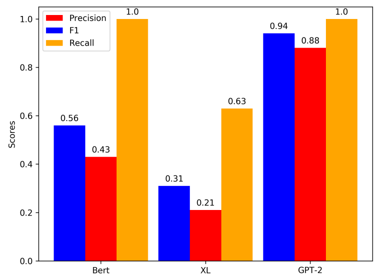
Another aspect that is evaluated, is the impact of the input sequence length, i.e. the number of concatenated log events for which the next log event shall be predicted, on the ability of the model to detect anomalies. The results of this evaluation can be seen in 4.9. Bert and GPT-2 seem to profit slightly from sequence lengths longer than 6 or 7, whereas the quality of results for XL-Transformers are not as affected from the input sequence length, but are generally lower.



(a) 5% alteration

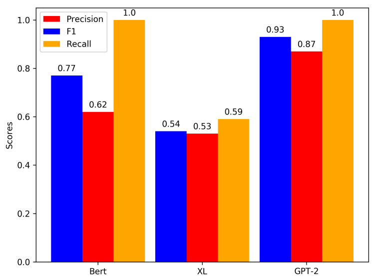


(b) 10% alteration

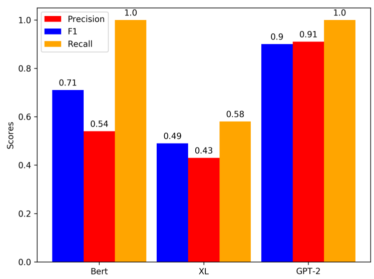


(c) 15% alteration

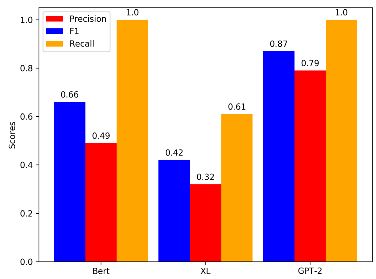
Figure 4.7: Altering the sequences of logs at different ratios, using regression.



(a) 5% alteration



(b) 10% alteration



(c) 15% alteration

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

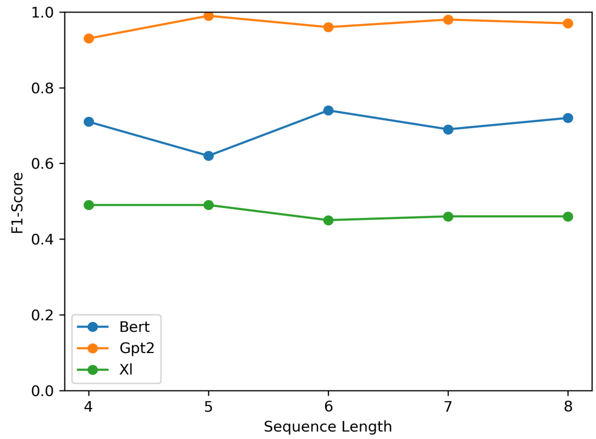


Figure 4.9: F1-Score for varying input sequence lengths, 15% word insertion alterations, 5% anomalies injected, using regression.

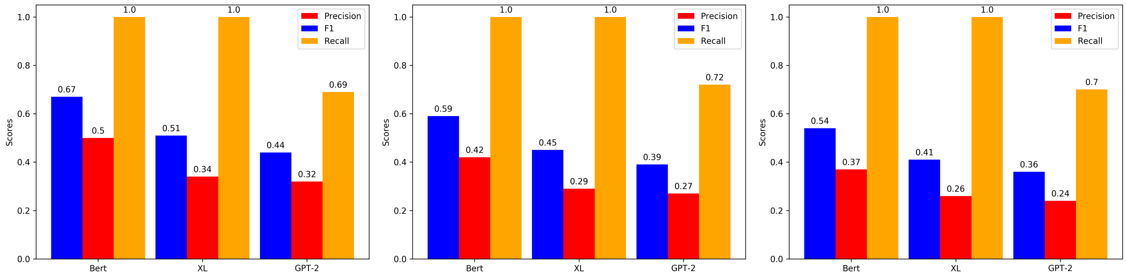
4.2.5 Classification-based approach using one dataset

In this subsection, the results of the classification-based approach are presented, with the exact same boundary conditions as in 4.2.4. The ability of the system to detect injected anomalies, despite injected alterations on the sequences of logs, can be seen in 4.10. The impact of alterations on the sequence of logs, i.e. deleting, shuffling and duplicating events are summarised in figure 4.7. Again, only average results on the individual injections are presented. Results broken down by each alteration can be found in the appendix .2.

In contrary to the results using regression, the classification approach seems most fit for Bert and much less for GPT-2. This is probably due to the smaller cosine distance between the templates in GPT-2, which makes it harder for the system to correctly assign the templates to the classes in the prediction process.

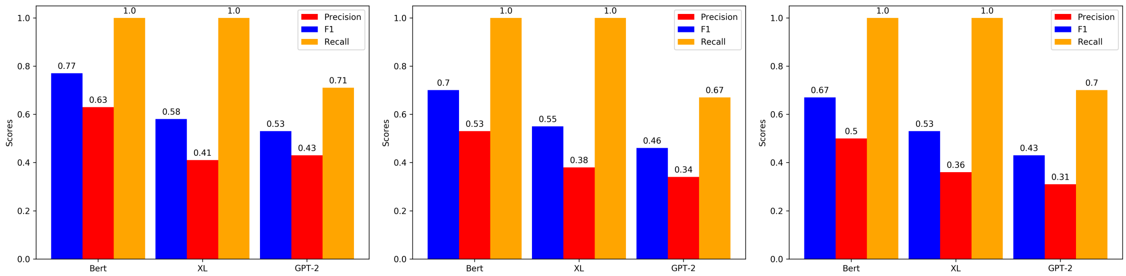
Additionally to the alterations on the log sequences, the results alterations on the log events themselves, i.e. inserting, removing and replacing words are of interest. The results of this experiment can be seen in figure 4.10. Again, the alterations are not injected all at once, but independently - the figure shows averaged results.

The impact of the input sequence length can be seen in 4.9. Bert again seems to profit slightly from sequence lengths longer than 6 or 7, 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.



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

Figure 4.10: Altering the order of log sequences at different ratios, using classification.



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

Figure 4.11: Altering log events at different ratios, using classification.

4.2.6 Transfer learning using the regression-based approach

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 B. Figure 4.13 shows the results for alterations on 5%, 10% and 15% of the log lines of all possible alterations at the same time, after 60 epochs of training on the train dataset A and 5 epochs of training on the train dataset B. It is clearly visible, that Bert and XL-Transformers are both degrading substantially in all metrics, yet GPT-2 achieves good results even with an increased alteration ratio.

Figure 4.14 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, although starting from a relatively low starting point, whereas Bert has a smaller increase per training epoch. The results of GPT-2 do not change much per epoch, but start at a very high level already, corresponding to the findings already made on GPT-2 using regression in 4.2.4. These findings are confirmed by the ROC curve plots which can be seen in figure 4.15, showing nearly perfect results for GPT-2, very good results for Bert and far less satisfying results for XL-Transformers.

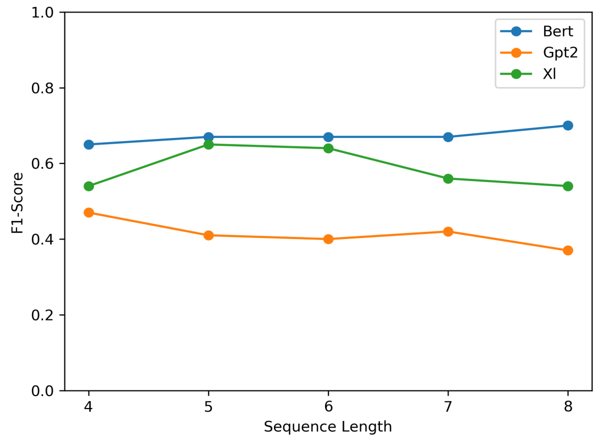
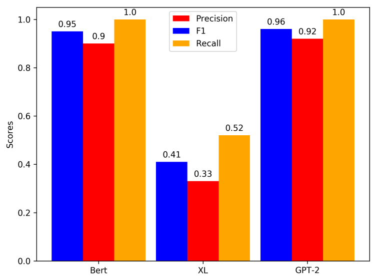
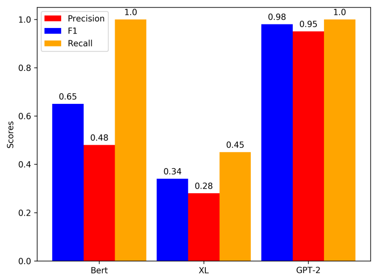


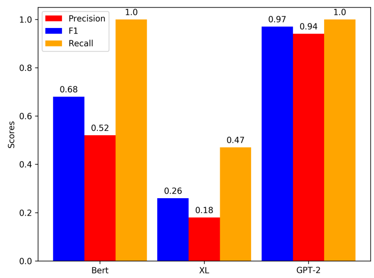
Figure 4.12: F1-Score for different input sequence lengths, with 15% word insertion, using classification.



(a) 5% alteration

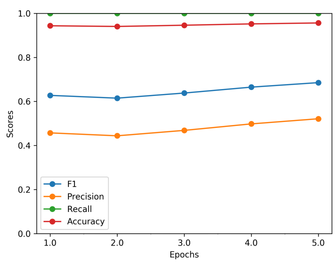


(b) 10% alteration

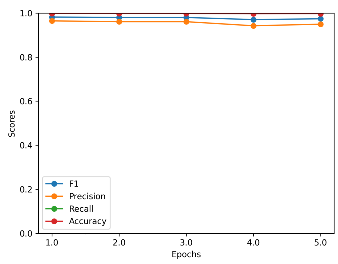


(c) 15% alteration

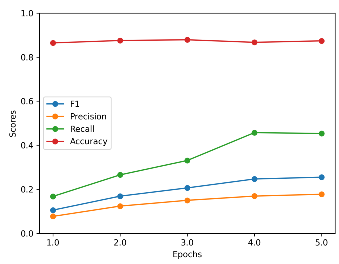
Figure 4.13: Transfer learning with different ratios of alteration, using regression.



(a) Bert

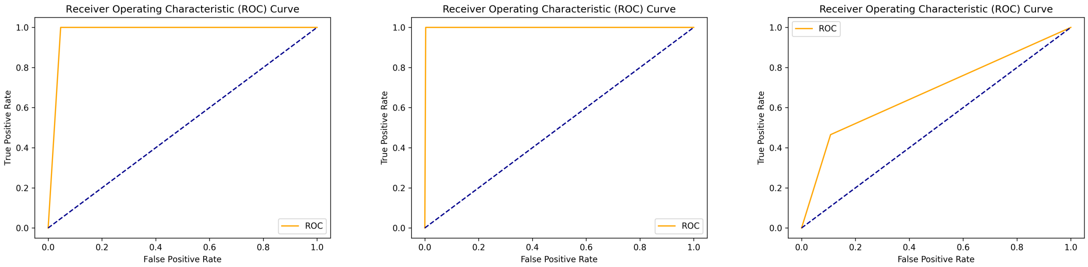


(b) GPT-2



(c) XL

Figure 4.14: Improvement of metrics per additional learning epoch, using regression.



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

Figure 4.15: ROC-Curve for transfer learning using regression with 15% alterations.

4.2.7 Transfer learning using the classification-based approach

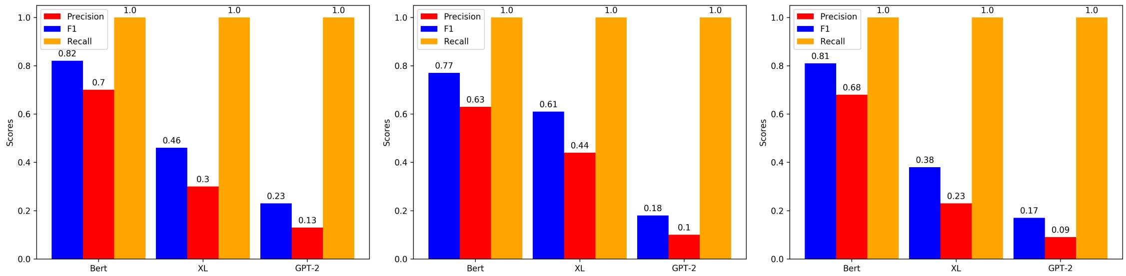
For transfer learning using the classification-based approach, the same experiments as described in 4.2.6 were conducted. Figure 4.16 shows the results of the transfer learning experiment. Interesting observations include the stable results achieved using Bert, which are little sensitive to increasing alteration ratios. Bert achieves a F-1 score of 0.82 for 5% alterations, 0.77 for 10% alterations and 0.81 for 15% alterations, proving that Bert has clear advantages over GPT-2 and XL-Transformers for the classification-based approach and the results of GPT-2, which are not as good as compared to the regression-based approach. It is noticeable, that all language models achieve a recall value of 1.0. Figure 4.17 depicts the development of the metrics of detecting 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, ramping up steeply in accuracy and especially for F-1 score and precision between epoch four and five. GPT-2 shows almost no improvements, with F1, precision and accuracy and XL-Transformers has small improvements over the course of the 5 epochs. These findings are confirmed by the ROC curve plots which can be seen in figure 4.18, showing very good results for Bert, acceptable results for Xl-Transformers and far less satisfying results for GPT-2.

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.

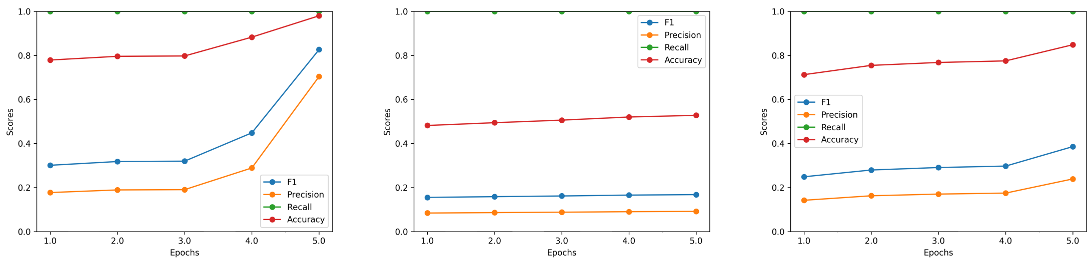
For the regression approach on one dataset, GPT-2 beats Bert and XL-Transformers clearly with regards to all metrics.

For the classification approach on one dataset on the other hand, GPT-2 shows weaker results than XL-Transformers and Bert. This is probably party due to the small cosine



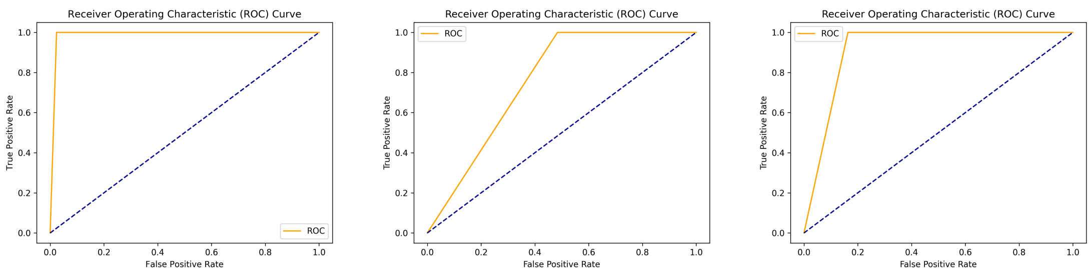
(a) 5% alteration (b) 10% alteration (c) 15% alteration

Figure 4.16: Transfer learning with different ratios of alteration, using classification.



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

Figure 4.17: Improvement of metrics per additional learning epoch, using classification.



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

Figure 4.18: ROC-Curve for transfer learning using regression with 15% alterations.

distance between the embeddings of the templates, where small changes on the templates can have a high impact already.

For the transfer learning approach, the results show a slightly different picture. Using the regression approach, GPT-2 is still ahead of the other two, while Bert seems to show better results than for detection on one dataset.

On the other hand, for transfer learning using the classification-based approach, for 15% alteration, GPT-2 shows results that are not much better than random, while still being able to find all anomalies, but showing a small F1-Score of only 0.17, where Bert is able to reach a F1-Score of 0.81. It is also visible, that GPT-2 seems to not profit as much from the few-show learning on dataset B, where the metrics stay almost the same throughout, yet Bert shows improvements from an F1-Score of 0.3 to 0.8.

It is evident, that with more hyperparameter-tuning, far better results can be accomplished, especially with XL-Transformers, since it obviously requires a different setup of hidden units and layers within the LSTM to achieve optimal results, however, since the model is a prototype and is supposed to show the possibility of comparing different word embeddings, not all potentials were fully tapped.

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. [30] and Yuan et al. [31] trained a SVM classifier to detect errors. Farshchi et al. [32] adopt a regression-based method to find correlations between an operation's logs and the operation activity's effect on cloud resources. Chen et al [33] 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. [34] 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 [35] and [36] both propose approaches that cluster logs.

The recent remarkable advances of deep learning depict new promising paths for anomaly detection in logs. While LSTMs have been put to use in detecting anomalies in time series generally [37], they have been used in anomaly detection in logs: Du et al. [2] present DeepLog, which is described in detail in section 5.1. Zhang et al. [38] use 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 [39] 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. [38] 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. [40] 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 [2], 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 like in most of the works in the area: Logs are first pre-processed with a log parser, they separate the constant from the variable part. Log templates are mapped to log keys k_i, which are just 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 results 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, thus learning the probability distribution $Pr(m_t = k_i | m_{t-h}, ..., m_{t-1})$, so given a log key sequence of length h, outputs a probability distribution of all possible log key values. A log key value is treated as normal, if it's 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. Then, the trained model is used, to check if the incoming log key is normal, by sending $w = \{m_{t-h}, ..., m_{t-1}\}$ as input. Additionally, the parameter value vector is checked. If the log entry is labeled 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, re-training of the model is necessary.

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

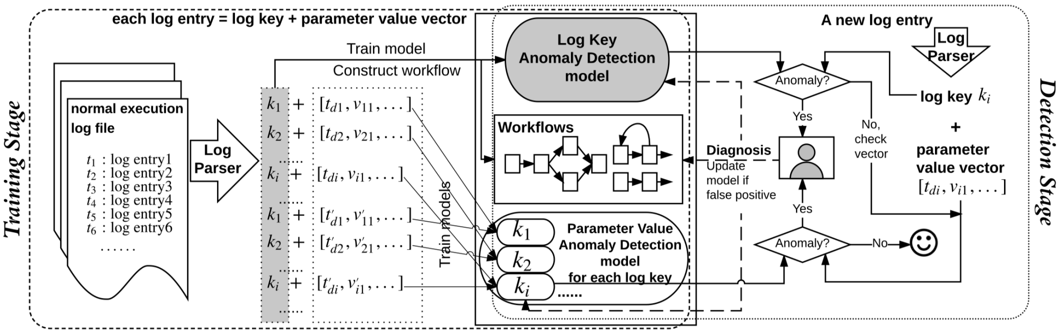


Figure 5.1: DeepLog model overview [8]

FT-Tree. These sequences are then transformed into embedding sequences using a novel word representation method, template2Vec, which is inspired by word2Vec [41]. Figure 5.4 shows the steps included for template2Vec in the offline learning stage: (1) shows the inclusion of common synonyms and antonyms in the English language extracted from WordNet [42]. Additionally, log-data-specific synonyms and antonyms can be added. In the second (2) step, dLCE [43] 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 log the sequences of log events $S = (s_1, s_2, ..., s_m)$ have been transformed into template vector sequences $V = (v_{s_1}, v_{s_2}, ..., v_{s_m})$, they apply an Attention-based Bi-LSTM 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}. Additionally 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 $v_{i-w+1}, s_{i-w+2}, ..., v_i$. 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 matches 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 employ 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].

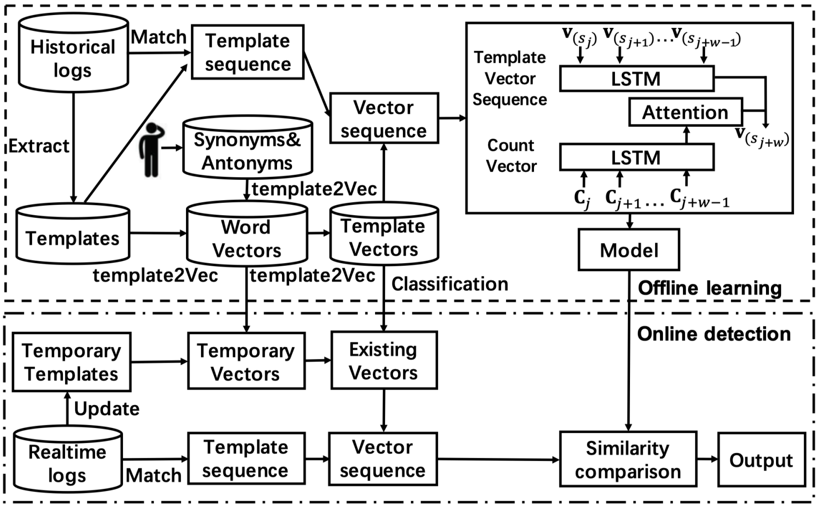


Figure 5.2: LogAnomaly model overview [1].

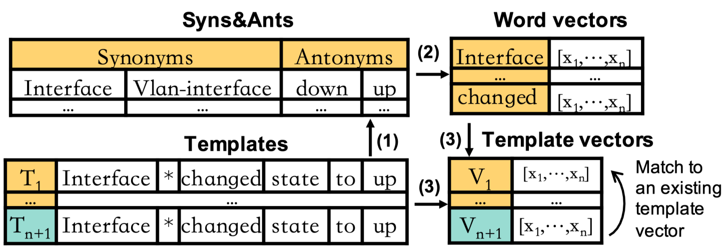


Figure 5.3: Example of Template2vec [1].

5.3 Robust Log-Based Anomaly Detection on Unstable Log Data

Zhang et al. proposed *LogRobust* [27], emphasising the necessity of anomaly detection on log data being sufficiently robust to the unstable nature of log data which is twofold: They are *evolving*, meaning that frequently modifying source code naturally leads to logging statements getting altered. Also 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 like in 5.2. After adopting Drain to parse log data and extract log templates from it, they apply FastText [44] 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 being 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. Then, 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.

The log sequences are then split into subsequences and 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, which 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 [34] which contains around 24 million log messages with about 2.9% labelled anomalies, commonly used for benchmarking. They simulate processing noise by applying various small changes to the order of log sequences (unstable seq.) and the evolution of log events by randomly removing or adding words from log events (unstable event). The resulting datasets can be seen in 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.

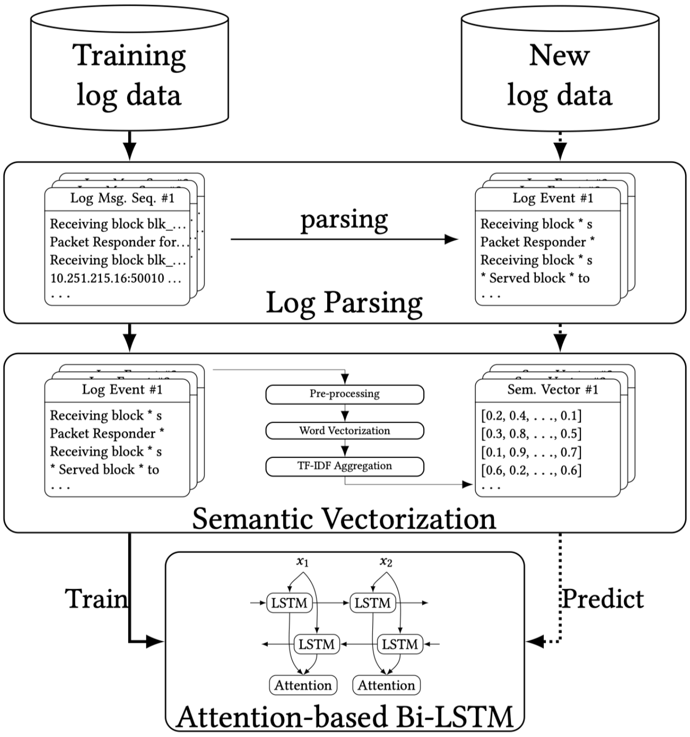


Figure 5.4: Overview of LogRobust [27].

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

Finding suitable representations for log data in the form of word embeddings is an important step towards building a robust, environment-agnostic anomaly detection model. This work presents an evaluation of three different word embedding models, namely Bert, GPT-2 and XL-Transformers and compares them using a regression-based and a classification-based approach. Additionally, in order to evaluate the robustness of the model, different alterations are injected into the logs. These alterations are finally combined together, to simulate the existence of a different log dataset B and the portability of obtained knowledge from training on a log dataset A.

The work done can be summarised by the following steps:

- Finding a suitable log parser by evaluating the available log parsers on their performance on the log dataset.
- Finding suitable language models in order to represent log templates so that they can be used for the task of anomaly detection
- Finding a means to alter log datasets in such a way that the existence of a different log dataset and the evolution of log data can be simulated in oder to evaluate the performance of the transfer learning method.
- Evaluation of the proposed model by comparing the quality of word embeddings obtained by different pre-trained language models with regards to the task of anomaly detection in system logs.

6.2 Problems Encountered

Several impediments occurred during the development of the model.

At the beginning, it was not obvious, which word embeddings to use in order to represent log events. First attempts were made with GloVe, using word vectors that were trained only the available small log corpus, since the publicly available pre-trained vectors did not have representations for log-domain-specific words like "MB", "GB", "deallocate", "VM". These representations of log events of unequal length were padded with zeros and fed into an Auto Encoder, in order to learn the sentence representations. Then, the latent space representation of every log event was used in order to obtain representations of equal length. This attempt yielded acceptable results on the easiest case of injecting anomalies without alterations, but were disappointing when injecting alterations, making it unfit for the transfer approach. This was unfortunate, since a lot of time was invested into this approach. Using more sophisticated language models like Bert delivered better results overall.

Finding appropriate hyperparameters for the model, for example sequence length, number of hidden units and number of layers for the LSTM or clipping, was not easy in the beginning, since all of them heavily influence the end result if not set correctly. A lot of development time was invested, trying to find the reasons for problems elsewhere, when the only problem was for example a wrongly chosen value for gradient clipping.

6.3 Outlook

In this section, possible improvements for the model are identified.

In order to have more ways of evaluating different language models, next to the regression and classification approach, a binary classification approach can be implemented.

The word embeddings are taken from the used language models as they are. Finetuning on the log corpora is only conducted using the default tasks that have been used to train the language model on extremely 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.

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 finding a proper attention mechanism, as described by Vaswani et al. [16], 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.

Fine-tuning a pre-trained language model explicitly on the task of anomaly detection by integrating the fine-tuning step into the model's pipeline, can potentially improve the quality of the word embeddings. This can especially be useful for the transfer learning task.

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] 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.

[3] 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.

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

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

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.

[14] "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.

[15] 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.

[16] 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.

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

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] “University of utah, datasets for the deeplog paper..” `https://www.cs.utah.edu/~mind/papers/deeplog_misc.html`. Accessed: 2020-06-11.

[29] “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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

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

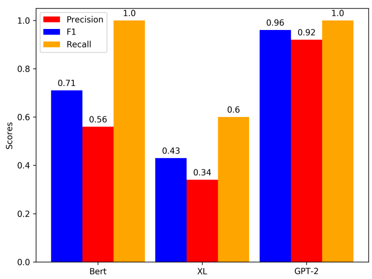
[43] K. A. Nguyen, S. S. i. Walde, and N. T. Vu, "Integrating distributional lexical contrast into word embeddings for antonym-synonym distinction," *arXiv preprint arXiv:1605.07766*, 2016.

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

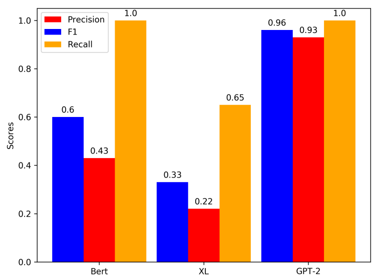
Annex

.1 Regression

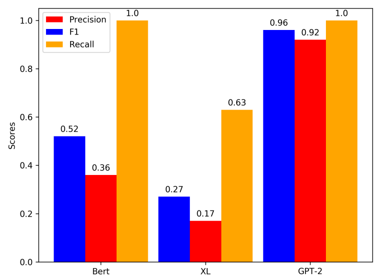
.1.1 Sequence-based injections



(a) 5% alteration

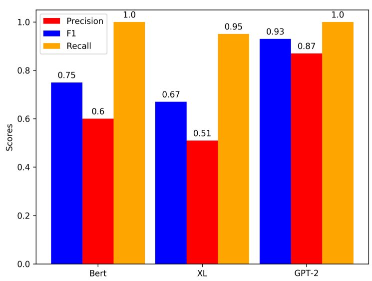


(b) 10% alteration

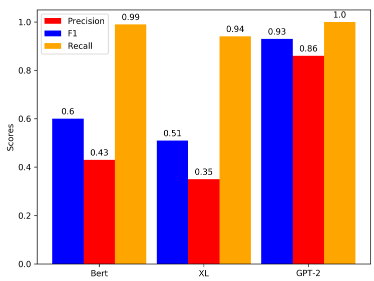


(c) 15% alteration

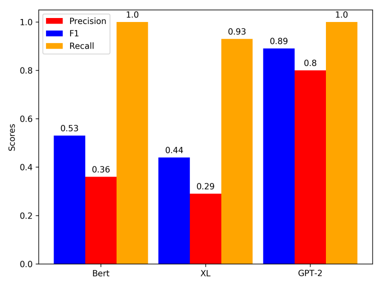
Figure .1: Delete lines at different ratios using, regression.



(a) 5% alteration

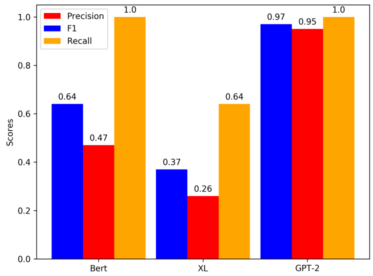


(b) 10% alteration

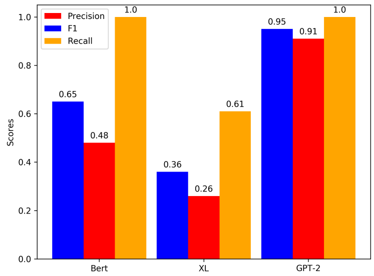


(c) 15% alteration

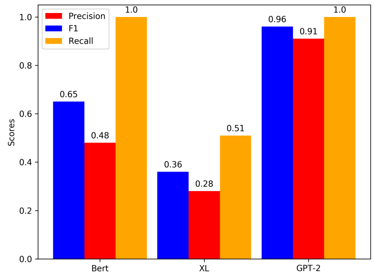
Figure .2: Duplicate lines at different ratios, using regression.



(a) 5% alteration

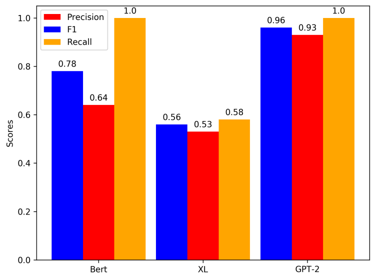


(b) 10% alteration

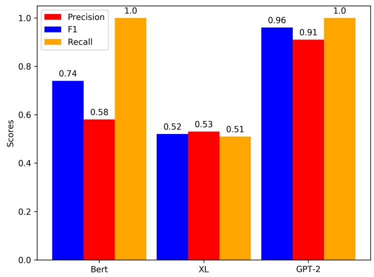


(c) 15% alteration

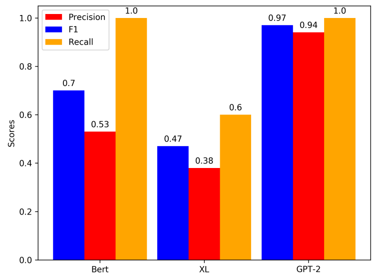
Figure .3: Shuffle lines at different ratios, using regression.



(a) 5% alteration

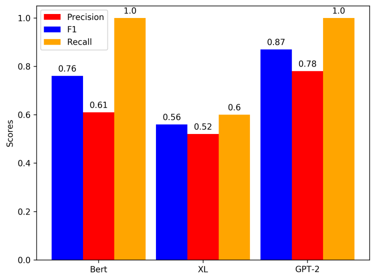


(b) 10% alteration

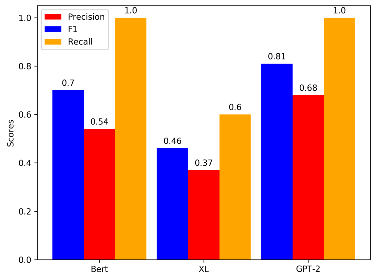


(c) 15% alteration

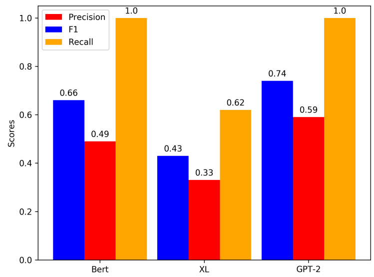
Figure .4: Insert words at different ratios, using regression.



(a) 5% alteration



(b) 10% alteration



(c) 15% alteration

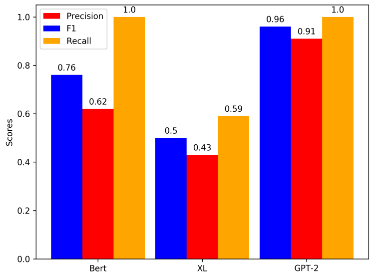
Figure .5: Remove words at different ratios, using regression.

.1.2 Injections on log events

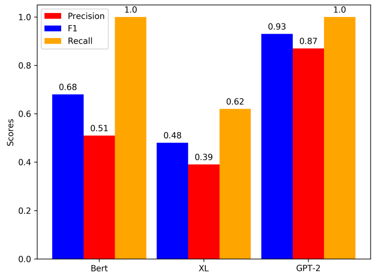
.2 Classification

.2.1 Sequence-based injections

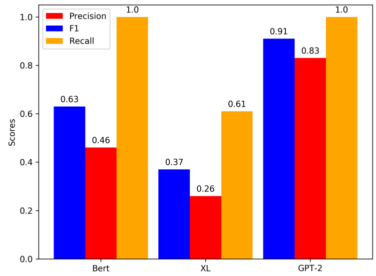
.2.2 Injections on log events



(a) 5% alteration

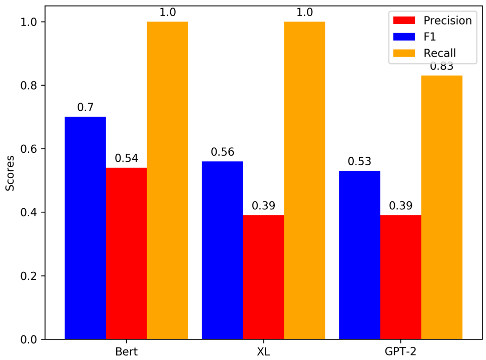


(b) 10% alteration

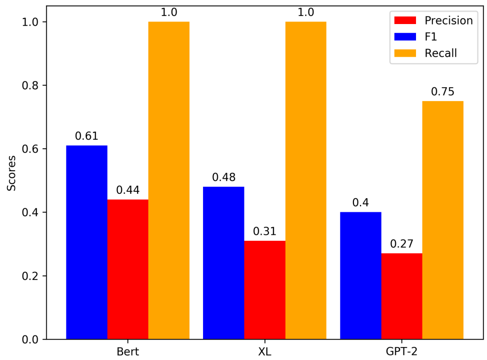


(c) 15% alteration

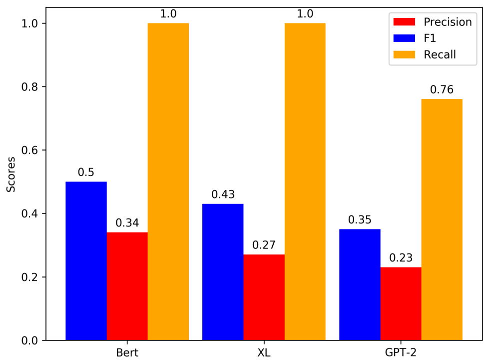
Figure .6: Replace words at different ratios, using regression.



(a) 5% alteration

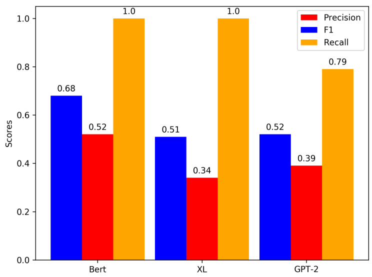


(b) 10% alteration

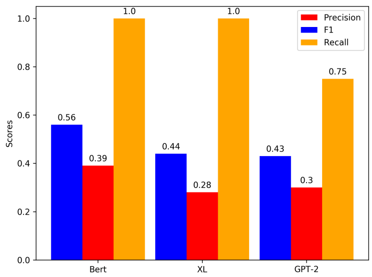


(c) 15% alteration

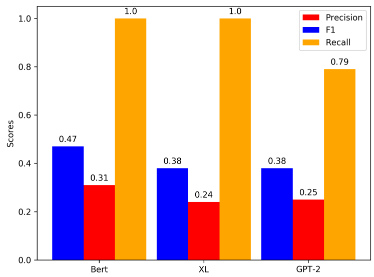
Figure .7: Delete lines at different ratios using, regression.



(a) 5% alteration

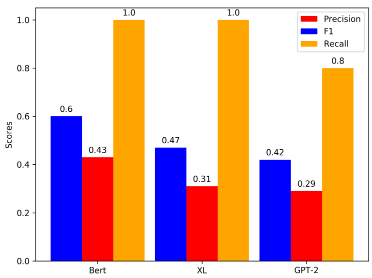


(b) 10% alteration

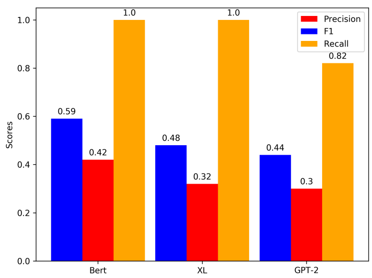


(c) 15% alteration

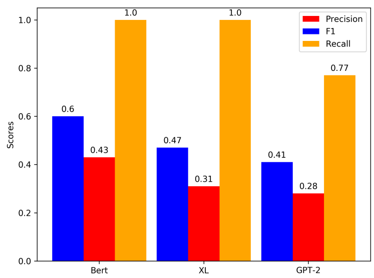
Figure .8: Duplicate lines at different ratios, using regression.



(a) 5% alteration

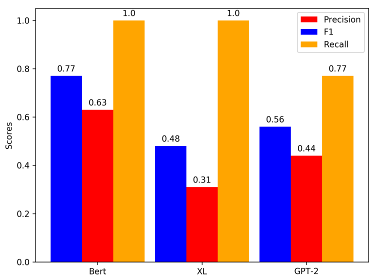


(b) 10% alteration

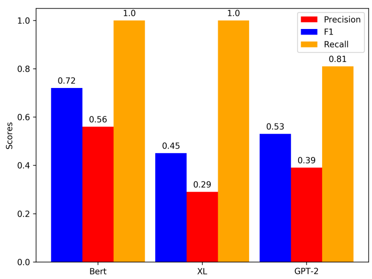


(c) 15% alteration

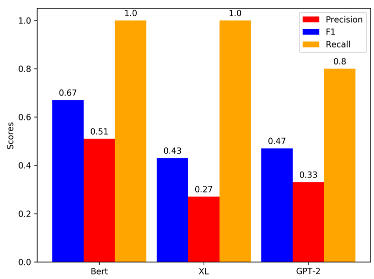
Figure .9: Shuffle lines at different ratios, using regression.



(a) 5% alteration

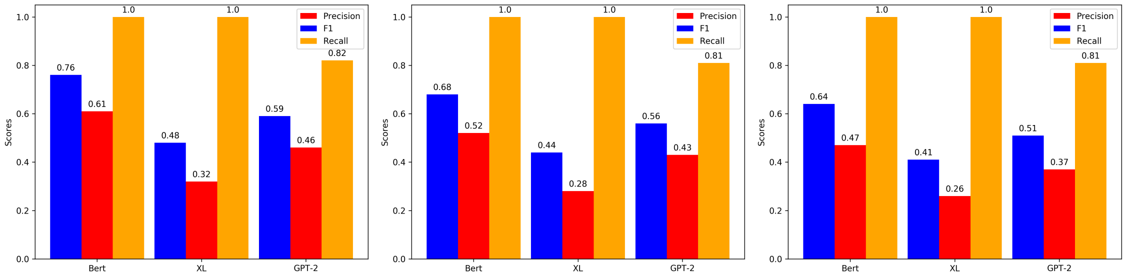


(b) 10% alteration



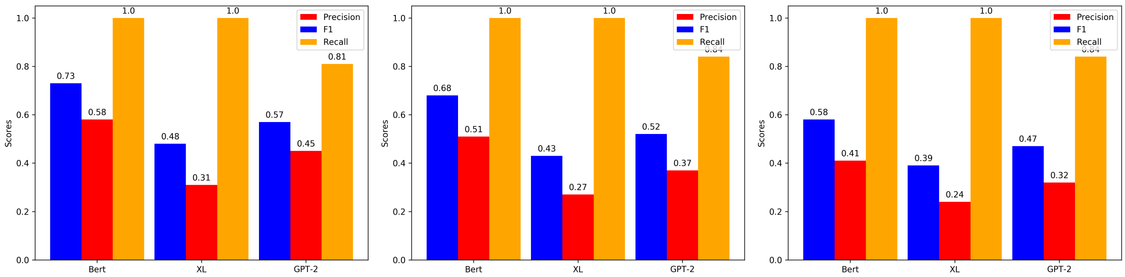
(c) 15% alteration

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.