Technische Universität Berlin

Institut für Telekommunikationssysteme
Fachgebiet Architektur der Vermittlungsknoten

Fakultät IV
Franklinstrasse 28-29
10587 Berlin

Master Thesis

Anomaly Detection in Infrastructure-agnostic Cloud Environments

Your Name

Matriculation Number: 1234567
June 19, 2020

Supervised by
Prof. Dr. Thomas Magedanz

Assistant Supervisor
Your second Supervisor

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

Berlin, June 19, 2020

. . . . . .
(Signature [your name])

Abstract

This template is intended to give an introduction of how to write diploma and master thesis at the chair 'Architektur der Vermittlungsknoten' of the Technische Universitï¿½t Berlin. Please don't use the term 'Technical University' in your thesis because this is a proper name.

On the one hand this PDF should give a guidance to people who will soon start to write their thesis. The overall structure is explained by examples. On the other hand this text is provided as a collection of LaTeX files that can be used as a template for a new thesis. Feel free to edit the design.

It is highly recommended to write your thesis with LaTeX. I prefer to use Miktex in combination with TeXnicCenter (both freeware) but you can use any other LaTeX software as well. For managing the references I use the open-source tool jabref. For diagrams and graphs I tend to use MS Visio with PDF plugin. Images look much better when saved as vector images. For logos and 'external' images use JPG or PNG. In your thesis you should try to explain as much as possible with the help of images.

The abstract is the most important part of your thesis. Take your time to write it as good as possible. Abstract should have no more than one page. It is normal to rewrite the abstract again and again, so probaly you won't write the final abstract before the last week of due-date. Before submitting your thesis you should give at least the abstract, the introduction and the conclusion to a native english speaker. It is likely that almost no one will read your thesis as a whole but most people will read the abstract, the introduction and the conclusion.

Start with some introductionary lines, followed by some words why your topic is relevant and why your solution is needed concluding with 'what I have done'. Don't use too many buzzwords. The abstract may also be read by people who are not familiar with your topic.

Zusammenfassung

Contents

List of Figures

List of Tables

1 Introduction

This chapter should have about 4-8 pages and at least one image, describing your topic and your concept. Usually the introduction chapter is separated into subsections like 'motivation', 'objective', 'scope' and 'outline'.

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.

This

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 are compared to those obtained from GPT-2 to use as input for a LSTM to learn error-free log sequences in a both supervised and unsupervised manner. For this purpose, three different ways of mapping the input of multiple log events to a target space, namely binary classification, multi-class classification and regression are evaluated.

1.3 Outline

The 'structure' or 'outline' section gives a brief introduction into the main chapters of your work. Write 2-5 lines about each chapter. Usually diploma thesis are separated into 6-8 main chapters.

This master's thesis is separated into 7 chapters.

Chapter ?? is usually termed 'Related Work', 'State of the Art' or 'Fundamentals'. Here you will describe relevant technologies and standards related to your topic. What did other scientists propose regarding your topic? This chapter makes about 20-30 percent of the complete thesis.

Chapter ?? analyzes the requirements for your component. This chapter will have 5-10 pages.

Chapter ?? is usually termed 'Concept', 'Design' or 'Model'. Here you describe your approach, give a high-level description to the architectural structure and to the single components that your solution consists of. Use structured images and UML diagrams for explanation. This chapter will have a volume of 20-30 percent of your thesis.

Chapter ?? describes the implementation part of your work. Don't explain every code detail but emphasize important aspects of your implementation. This chapter will have a volume of 15-20 percent of your thesis.

Chapter ?? is usually termed 'Evaluation' or 'Validation'. How did you test it? In which environment? How does it scale? Measurements, tests, screenshots. This chapter will have a volume of 10-15 percent of your thesis.

Chapter 6 summarizes the thesis, describes the problems that occurred and gives an outlook about future work. Should have about 4-6 pages.

2 Background

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

In 2.1 a description of the term cloud computing is given. In 2.2 the theoretical foundation of deep neural networks is described, followed by an introduction to LSTMs. 2.4 gives insights on how natural language can be represented using language models. In 2.3 techniques on tackling the problem of anomaly detection with deep learning techniques are presented.

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 logs to be able to retrace errors that have

occurred.

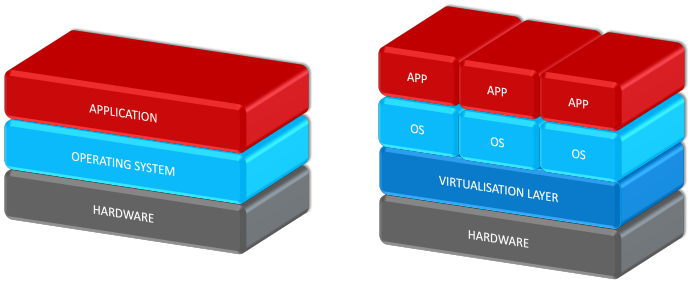

Figure 2.1: Traditional architeture vs. virtualised architecture

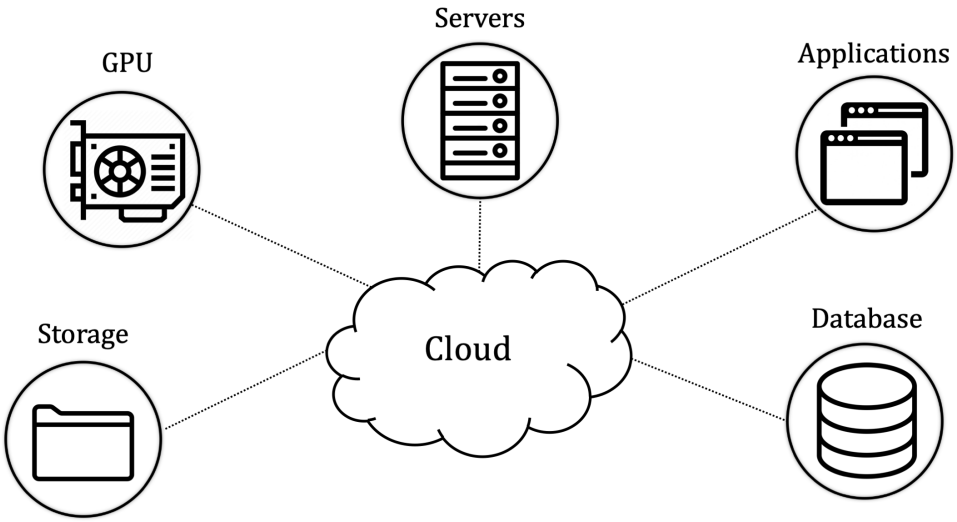

Figure 2.2: Traditional architeture vs. virtualised architecture

2.2 Deep Learning

2.2.1 Neural Networks

Neural networks are self-learning systems that constitute a subcategory of machine learning. By analysing training examples, it learns to compute a functional relationship between an input and an output [4]. Figure 2.4 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 the output [5]. 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 [4]. For **regression** problems, the linear activation function $h(x) = x$ can be used. For **multi-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 [5].

There are more cases, but regression and multi-classification are the most relevant here, as they are used in chapter 3.

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

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

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

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

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

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

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 multiclass 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.

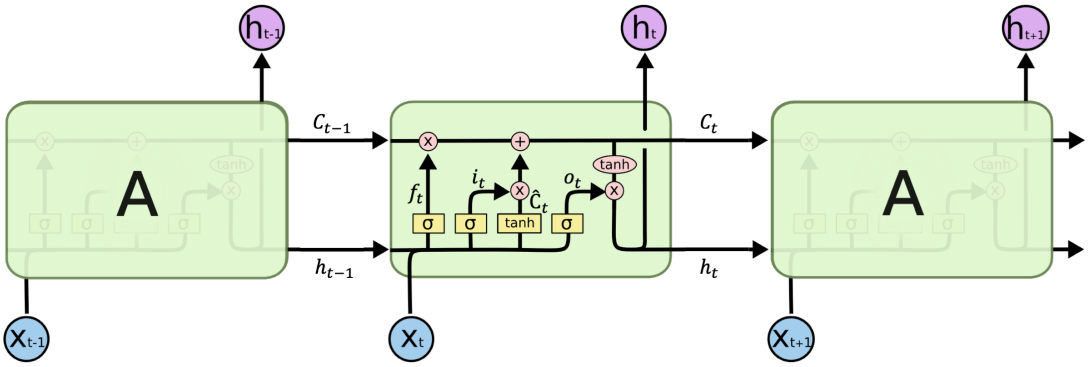

Figure 2.3: LSTM blocks [6]

2.2.2 Long Short-Term Memory

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 [7], 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 [8], allowing it to remove or add information. The traditional LSTM architecture was first described by [9]. The schematic structure of LSTM blocks is depicted in figure 2.3.

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, 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 [6].

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{2.6}$$

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \tag{2.7}$$

$$\hat{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \tag{2.8}$$

$$C_t = f_t \cdot C_{t-1} + i_t \cdot \hat{C}_t \tag{2.9}$$

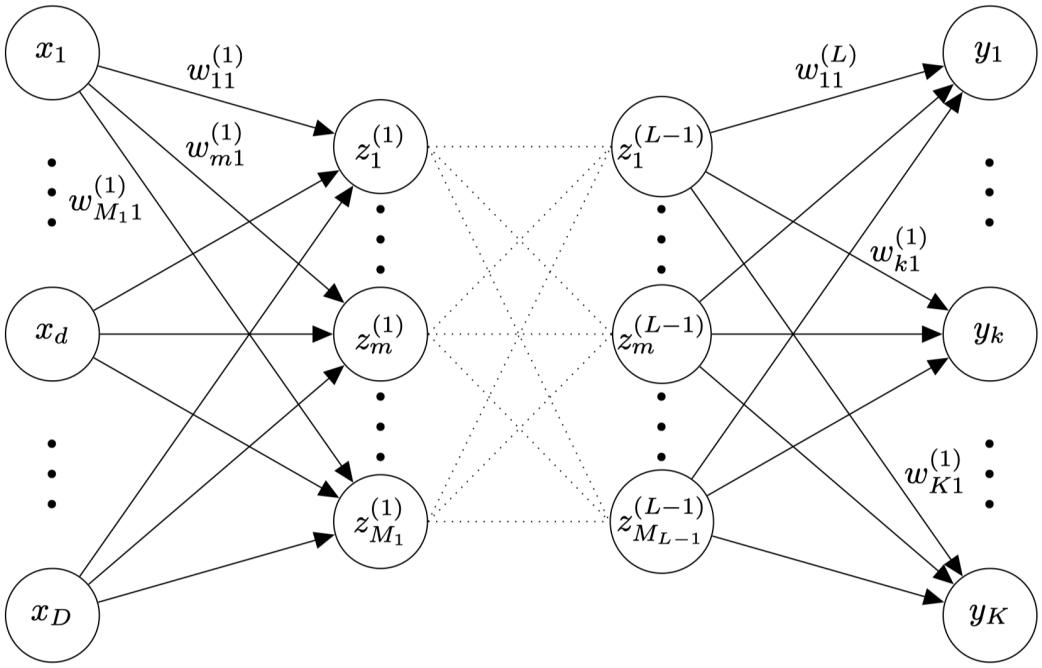

Figure 2.4: A neural network [5]

$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o) \tag{2.10}$$

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

2.3 Anomaly Detection

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 2.6, 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 [10].

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, machine learning which are applied to the domain in question [10]. Figure 2.5 outlines the central steps involved in finding an appropriate anomaly detection technique to a given problem.

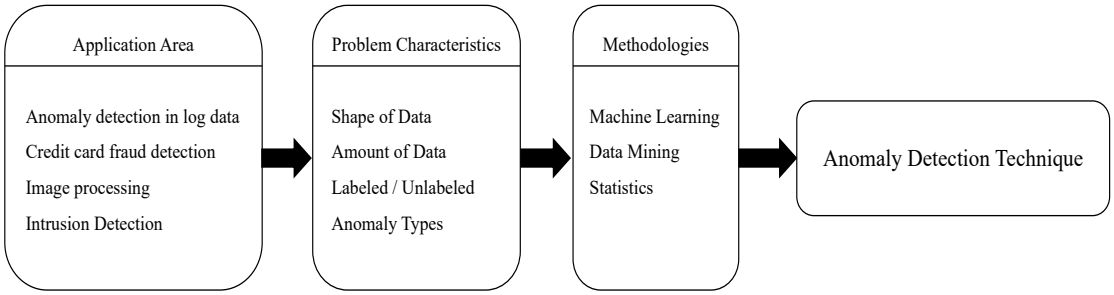

Figure 2.5: Schema of the development of an anomaly Detection technique [10].

2.4 Natural Language Processing

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.

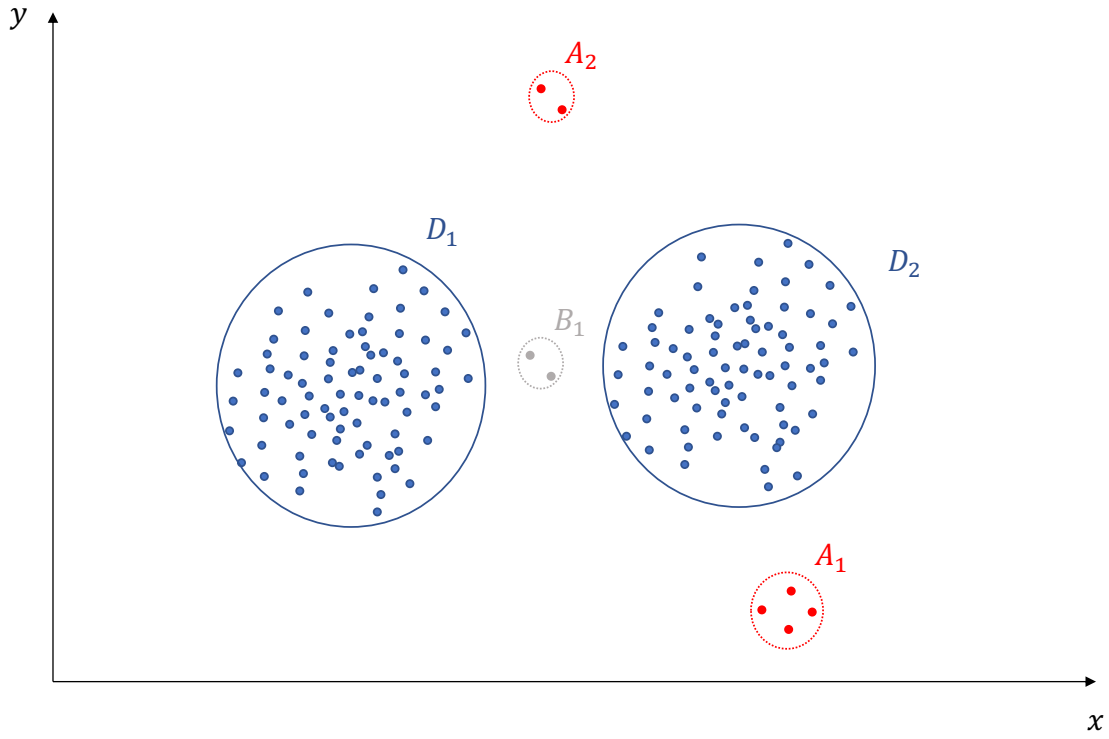

Figure 2.6: Example of anomalies in a dataset.

2.4.1 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* [11]. 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 [12]. Word embeddings can be retreived either by Principle Component Analysis or by using deep neural network models and capturing their internal states.

2.4.2 Bert

The development 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. [13]. The model architecture is a multi-layer bidirectional Transformer encoder. 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 2.7. 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 2.7 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 [14], 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. [15], a collection of 100k crowd-sourced question/answer pairs, or the General Language Understanding Evaluation (GLUE) benchmark by Wang et al. [16], which involves various natural language understanding tasks.

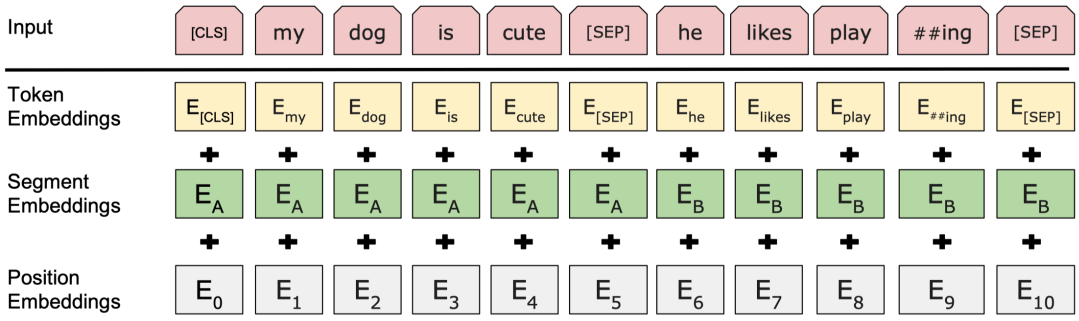

Figure 2.7: Bert

2.5 Log Parsing

Due to the unstructured nature of log data, the first crucial step in anomaly detection on log data is parsing. 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 [17][18]. Figure 2.8 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 re-

sulting 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 [19] or workload modelling [20]. Logs can also be used as data sources for performance modelling [21] 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 [18].

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 [22]. 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 [18].
- 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 [23]. 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 [22].

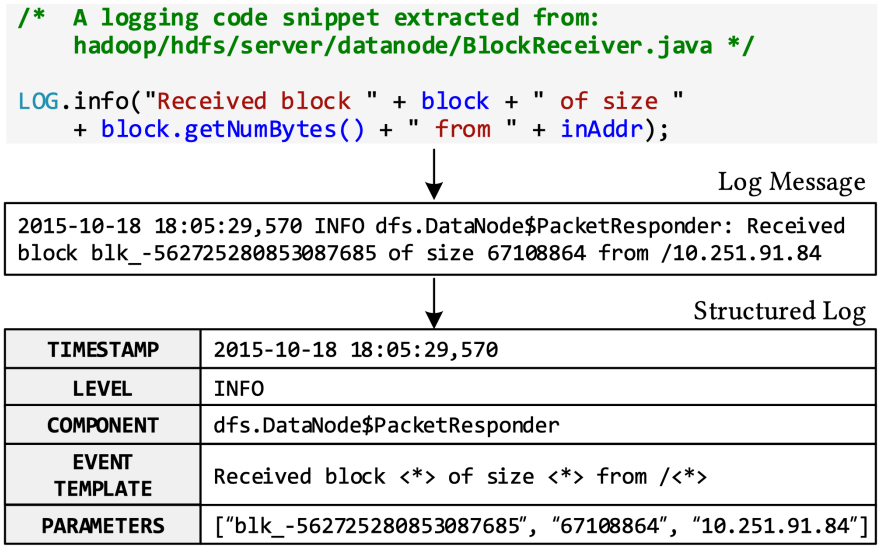

Figure 2.8: Schematic execution of log parsing [18]

3 Concept

Establishing a connection between the latest advances in NLP and anomaly detection in system log data is a recently emerged field. In particular, 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 and 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 given, and necessary requirements for this work are specified. In the following section 3.2, the overall system architecture with its components is outlined and visualised. In section **??** the developed approaches, including the baseline approach, are explained in detail. There exist three approaches, (1) the baseline approach featuring learning of sentence level log line representations with the help of an auto encoder on the basis of word level language representations and classification using regression, followed by the (2) regression and (3) multi-class classification approach using log line representations obtained by a language model which is able to transform log lines on sentence level. In section 3.5 the transfer learning mechanism is described in detail. In the final section 3.6 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. As a first step, these logs must be brought into an appropriate form, by these sequential patterns must be learned

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 $m_i \in \mathcal{A}$. 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 $f : \mathcal{A} \times \mathcal{A} \times ... \times \mathcal{A} \rightarrow \mathbb{R}^w$ be a function that takes a variable length of tokens that make up a log message and

maps them to a fixed length vector of dimension w. Let $\mathcal{D} = \{(e_1, b_1), ..., (e_n, b_n)\}$ be the dataset that consists of n log events, where $e_i \in \mathbb{R}^w$ is a w-dimensional vector (representing one of the aforementioned log events, computed by the function f), and where $b_i \in 0, 1$ 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.

Multiclass classification: Let $g : |s| \times \mathbb{R}^w \to \mathbb{N},\ g(e_i, s, \Psi) = c$ be a function computed by a neural network, that maps a sequence of s vectors $E = (e_j, ..., e_{j+s})$ to a natural number c out of a set of numbers that represent pre-defined log event class indices. This index is the subsequent index after s vectors. The objective is to learn the parameters Ψ, so that for each fixed length sequence of vectors, the function predicts the correct subsequent index. If the predicted index matches the actual index, then $\hat{b_i} = 0$ is returned, if it does not match the actual index, $\hat{b_i} = 1$ 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 thresholds, then $\hat{b_i} = 1$ is returned, if it is inside these thresholds, then $\hat{b_i} = 0$ is returned.

3.1.2 Requirements

1.

3.2 System Overview

In this section, a broad overview of the overall system is presented, with figure 3.2 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 3.2 or the next sequence class, which is called the classification task, specified in 3.4.2 and coloured blue in figure 3.2. 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 3.4.5. 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

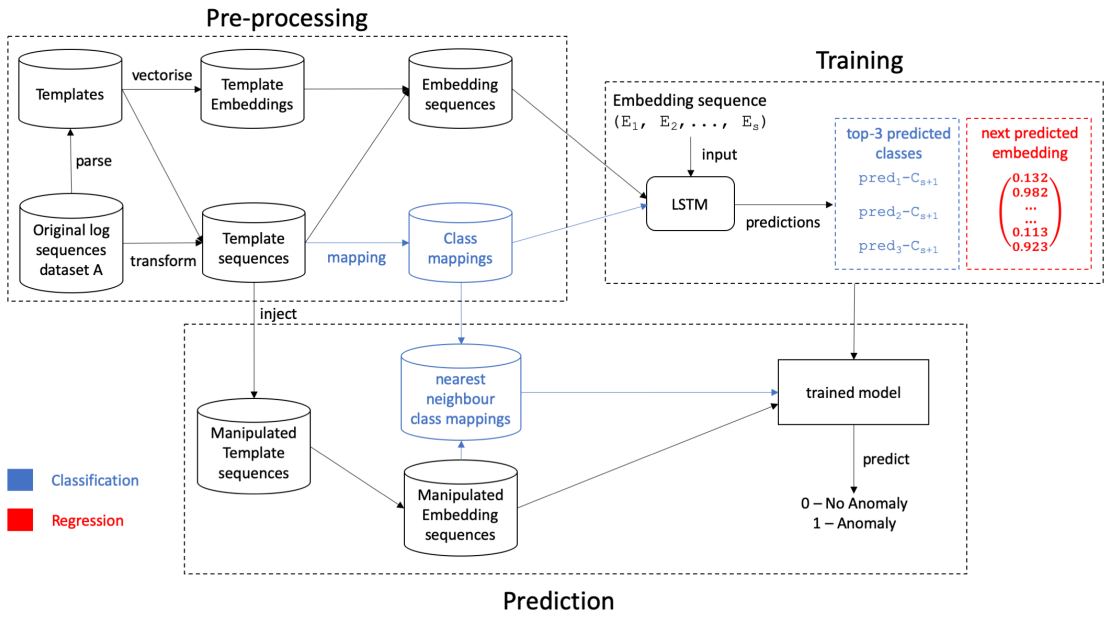

Figure 3.1: Anomaly Detection System

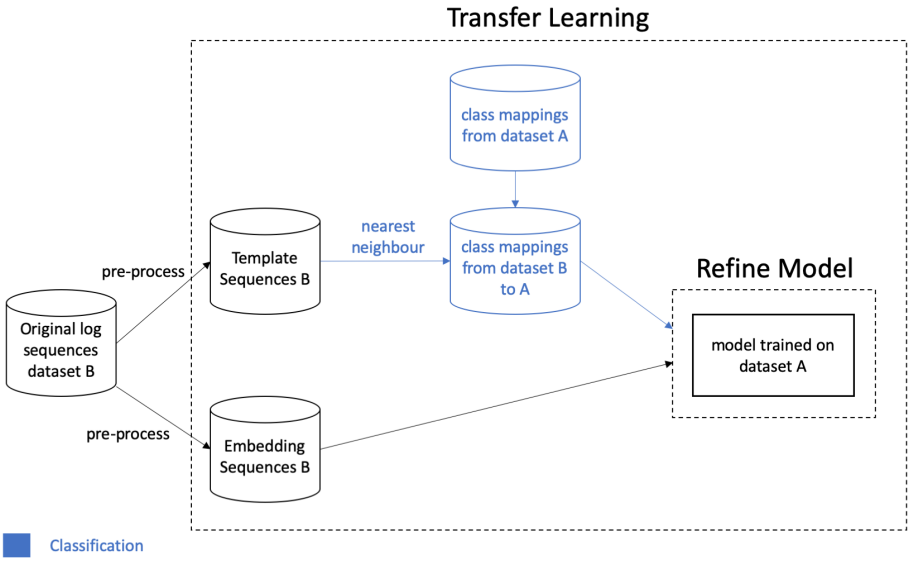

Figure 3.2: Transfer Learning System

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.

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 2.8. 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 2.8 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 and trailing whitespace characters are removed, and all repeated whitespaces are removed.

3.3.3 Word vectorisation

After the aforementioned pre-processing steps, the log events are transformed into word embeddings, using an arbitrary language model that is able to convert words or whole sentences into word or sentence embedding, effectively representing function f defined in 3.1.1. 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 [24].

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 [14]. 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 to the task in question.

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 order to learn sequences of logs, n embeddings are concatenated to form an embedding sequence E. Taking a sequence of embeddings as input, a *Bi-LSTM* neural network is utilised to predict the class or embedding at position $n + 1$. Figure 3.3 shows the structure of the Bi-LSTM. As a first step, a dropout is applied to the input sequence, before feeding it to the forward and backward layers of the Bi-LSTM. Then, the outputs of the

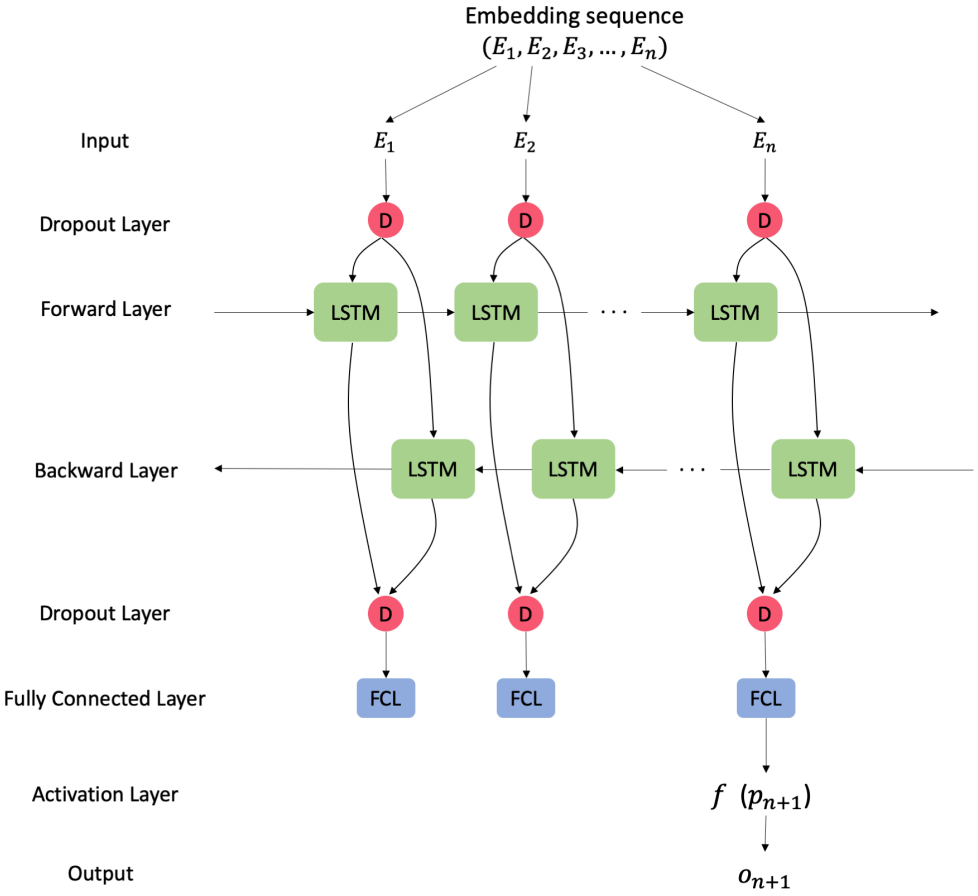

Figure 3.3: Bi-LSTM model

3.4.2 Classification

For the multi-classification approach, the finite set of n log event templates is mapped to class indices $\{c \in \mathbb{N}^+ : c \leq n\}$. After alterations are injected, as described in 3.4.5, the

set of templates has changed, and consists of more than n unique templates. For every template in the new set of templates, the nearest neighbour out of the templates of the original dataset is determined and will get the respective class assigned, as depicted in figure 3.5. This means in particular, that for every unique template, the corresponding word embedding is retrieved, and for every one of the word embeddings on the manipulated dataset, and is mapped to the word embedding from the original dataset with the lowest cosine distance. Additionally, a threshold has to be found, so that if for a given word embedding in the manipulated 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 original dataset template's classes, which is not desirable behaviour.

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 f, with n being the number of log events, s being the length of the sequence of word embeddings for which the neural network shall learn the consequent class and f being the dimensionality of the word vector.

- The *target* values of the training data is structured as follows: for every sequence of word embeddings $(i, ..., i + s)$, there is a corresponding class that stands for the embedding at position $s + 1$.

After training has been executed, the prediction phase starts, where the manipulated dataset can be processed by the neural network. For every sequence of log events of length s the system shall return the m most likely classes that it predicts as the consequent one. For $m = 3$, and an input of indices $(i, ..., i + s)$ the system would return a tuple (a, b, c) - if the true subsequent label at position $i + s + 1$ is one of these, the system will mark the log event at position $i + 7$ as non-anomalous (0), and anomalous (1) otherwise.

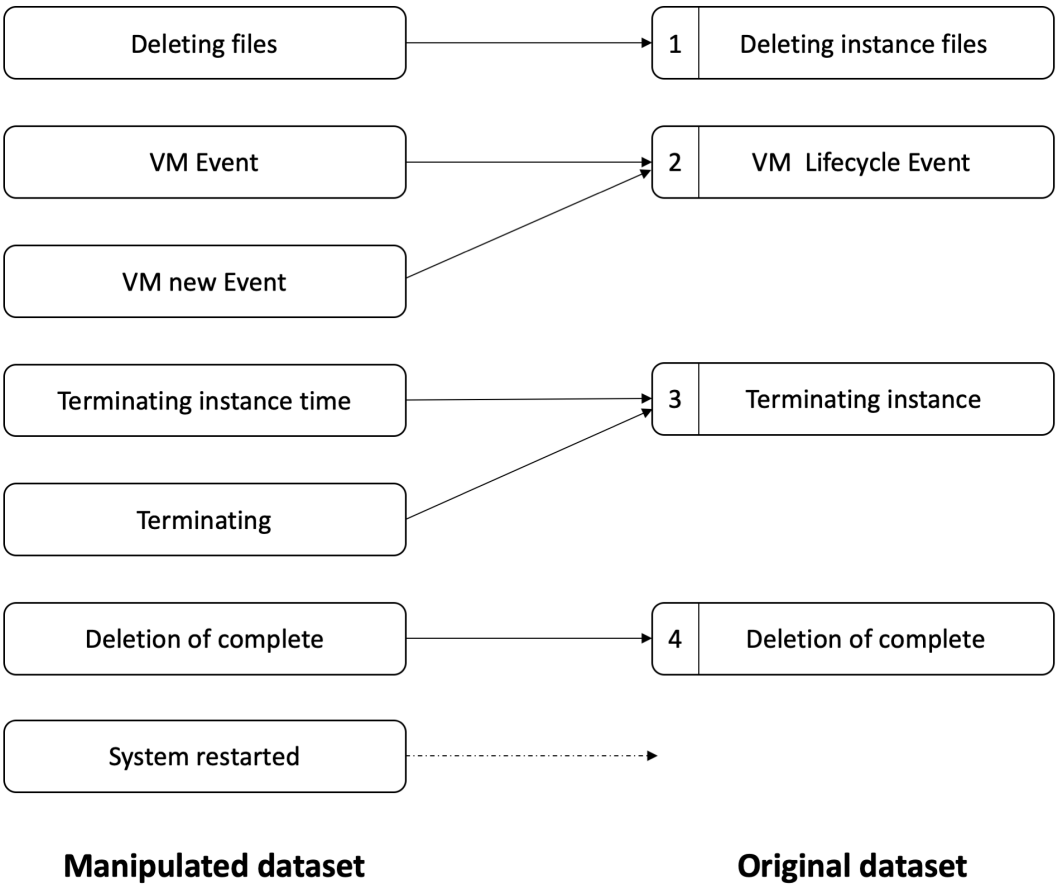

Figure 3.4: Template mapping

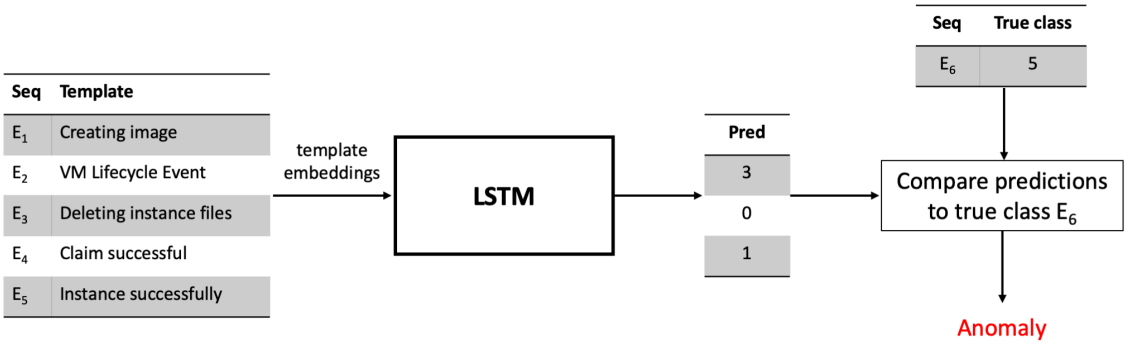

Figure 3.5: Template mapping

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 $(i, ..., i+s)$ is the embedding of the log event at position $i+s+1$, meaning the neural network shall predict the next embedding. 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 $(i, ..., i+s)$, and the predicted word vector of the neural network, is computed. Afterwards, the *q-th* percentile of the gathered loss values is computed (the optimal value q for the following purpose is to be determined by a simple grid-search). Now, for the sequences of the manipulated dataset, the loss values are computed the same way as for the original dataset. The system will then mark every log event whose word embedding's loss value is above the calculated percentile as an anomaly (1)f, otherwise as non anomalous (0).

3.4.4 Latent Space Representation

The Latent Space Representation approach consists of two main parts: learning a suitable representation for word embedding of different length and then continuing with the regression approach described in **??**. First, GloVe [25] is trained on the templates of a given dataset, to learn useful word embeddings for the given corpus. It is then used, to transform the templates into word vectors, word by word. The resulting lists of word vectors are then padded with zeros, to receive lists of equal length. These are then used as input for a simple auto encoder neural network, in order to learn a reduced representation of given sentence vectors. These representations are then used as input in the same manner as described in **??**.

3.4.5 Logs 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. 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 3.6 or on the sequence of log events as in figure 3.7.

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.

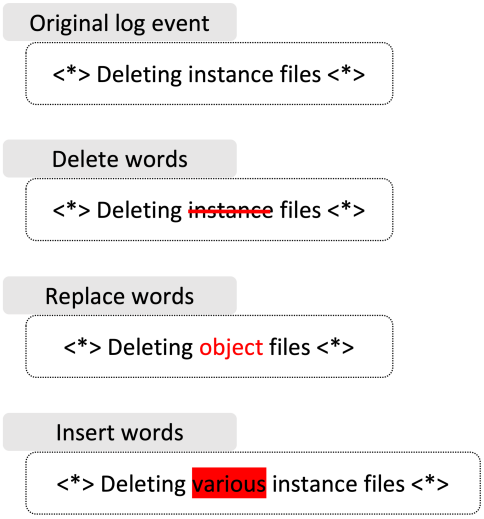

Figure 3.6: Raw log message

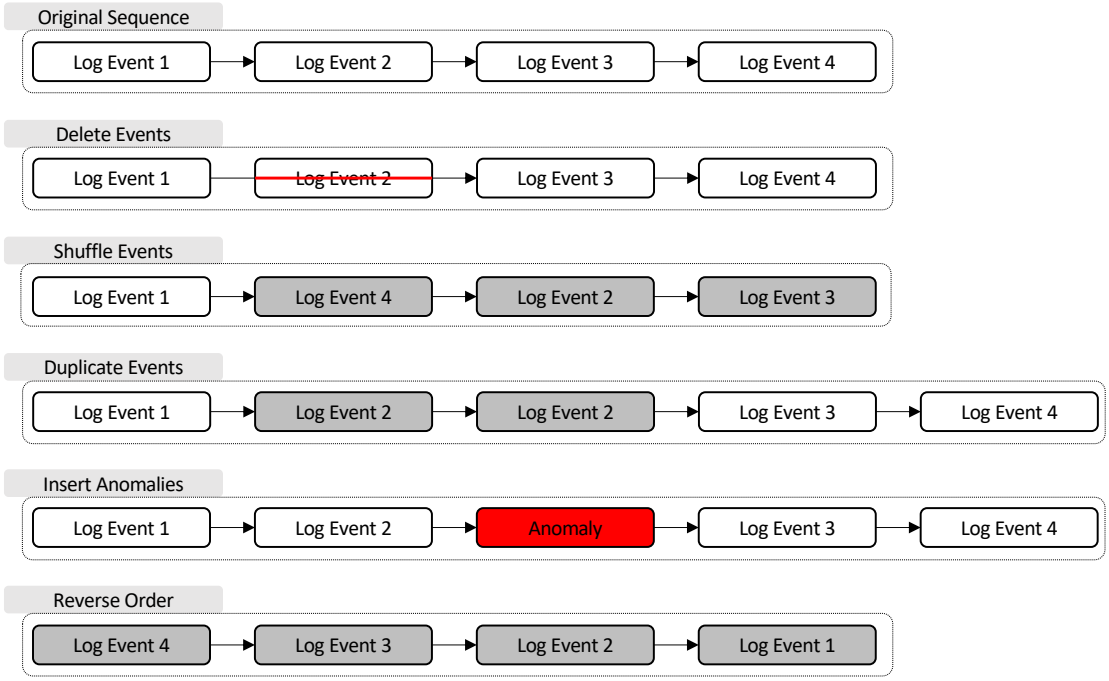

Figure 3.7: Raw log message

3.5 Transfer Learning

Transfer learning mainly involves training a model on a log dataset A, and then re-using the obtained knowledge to adjust the model on a log dataset B.

3.5.1 Classification

3.5.2 Regression

3.5.3 Smart Transfer

Since training on dataset 1 has been conducted using classification, a mechanism for mapping the templates of dataset 2 to the classes which have been assigned to the templates of dataset 1. For this, for every template of dataset 2, the nearest neighbour of, i.e. the one with the shortest cosine distance

3.6 Possible Improvements and Extendibility

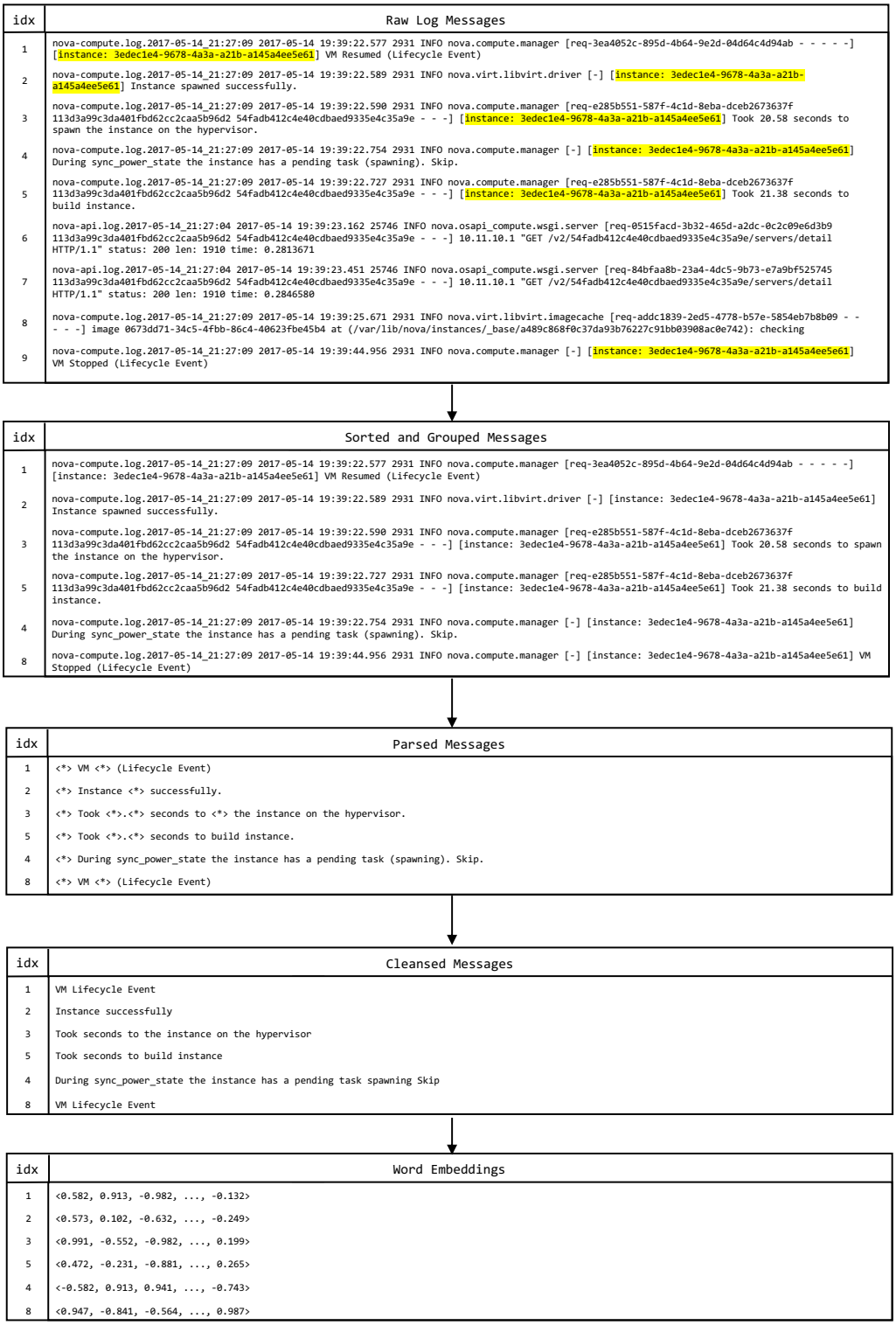

Figure 3.8: Parsing

4 Results

In the following, the results are presented

4.1 TODO

4.1.1 String cleansing

String cleansing, as described in **??**, drastically improves distinguishability between templates in the used dataset, as depicted in figure 4.1 and figure 4.2 - the templates corresponding to the indices can be found in table 4.1 and table 4.2, respectively.

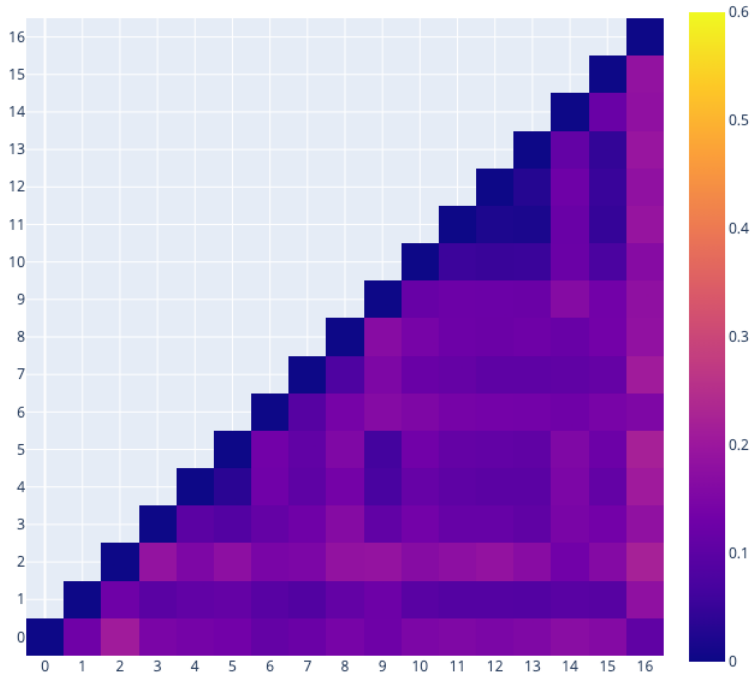

Figure 4.1: Cosine distance between templates without cleansing

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

Table 4.1: Templates before cleansing

4.1.2 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 that consists of the templates in table 4.2 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 [13], does not yield the desired results. As it can be seen in figure 4.4, it was not possible to increase the cosine distances between templates on the task of Masked LM, compared to the cosine distances depicted in figure 4.2. The increasing evaluation loss as depicted in figure 4.3 shows that training was not successful.

4.1.3 Regression

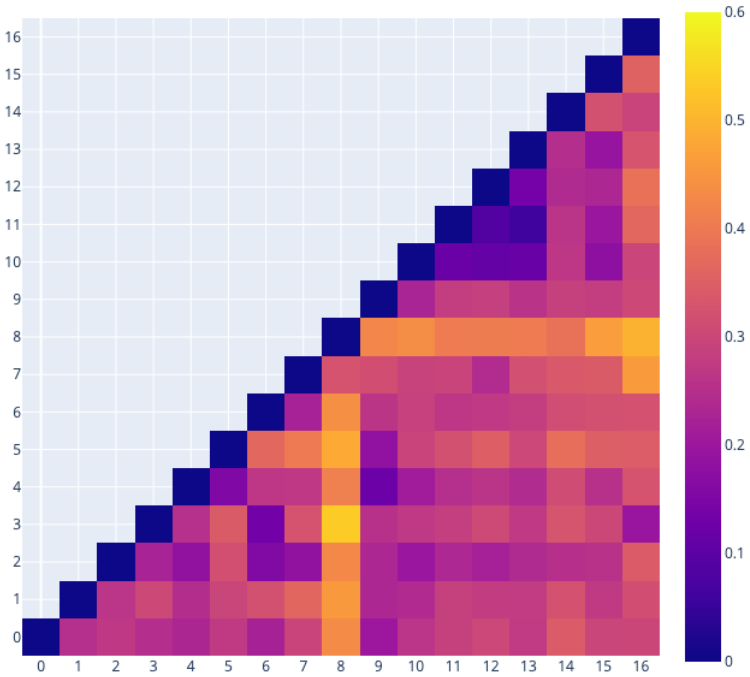

Figure 4.2: Cosine distance between templates after cleansing

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

Table 4.2: Templates after cleansing

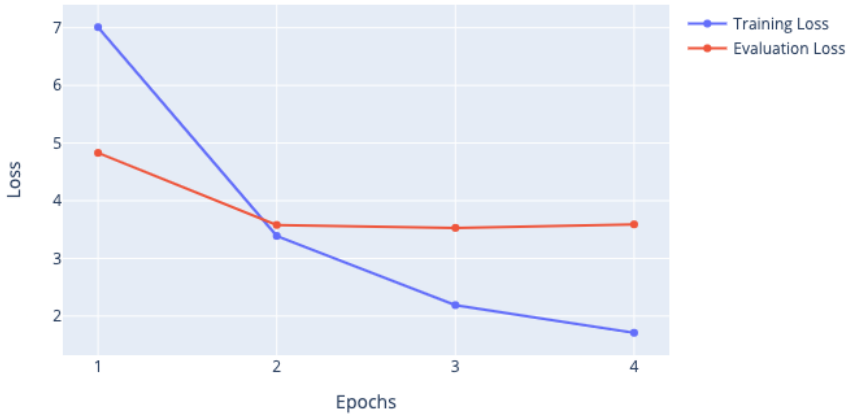

Figure 4.3: Training and evaluation loss for finetuning on masked

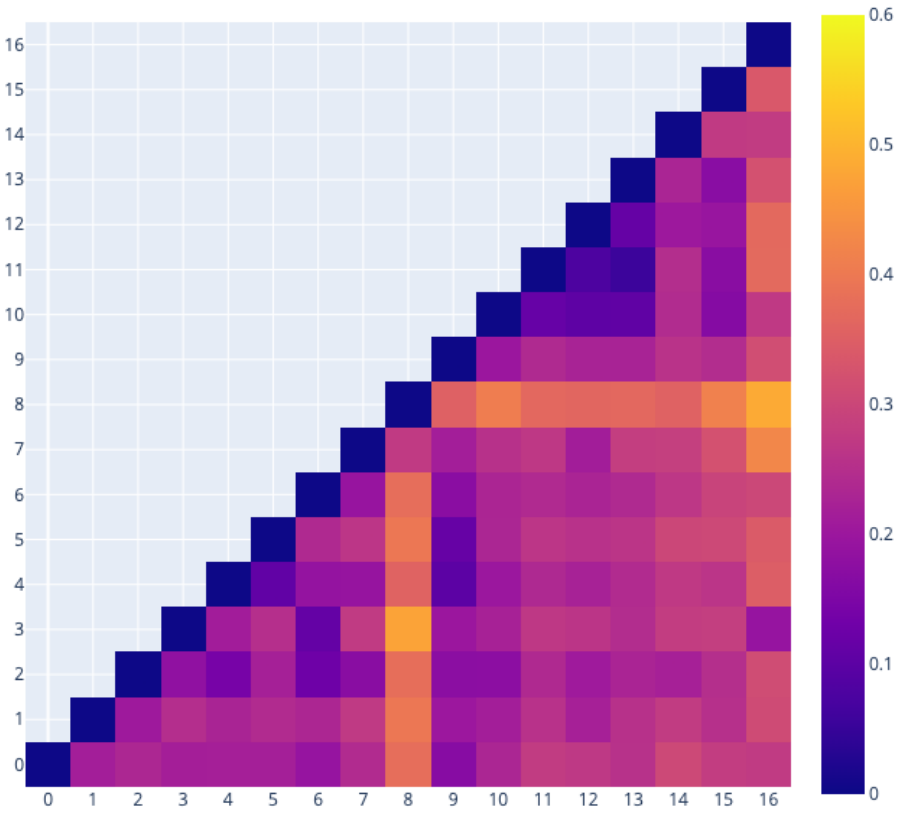

Figure 4.4: Cosine distance between templates after cleansing and finetuning

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. [26] and Yuan et al. [27] trained a SVM classifier to detect errors. Farshchi et al. [28] 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 [29] 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. [30] 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 [31] and [32] 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 [33], they have been used in anomaly detection in logs: Du et al. [2] present DeepLog: first, log events are mapped to keys, the sequential order of these keys is learned using a LSTM to detect anomalies. Zhang et al. [34] 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 [35] 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. [34] 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. [36] use combine attention based models together with word word embeddings. These approaches do not take into account the contextual information in log sequences.

Recently, LogRobust and LogAnomaly use pre-trained word embeddings, using an attention-based Bi-LSTM model to learn log sequences.

6 Conclusion

The final chapter summarizes the thesis. The first subsection outlines the main ideas behind Component X and recapitulates the work steps. Issues that remained unsolved are then described. Finally the potential of the proposed solution and future work is surveyed in an outlook.

6.1 Summary

Explain what you did during the last 6 month on 1 or 2 pages!

The work done can be summarized into the following work steps

- Analysis of available technologies
- Selection of 3 relevant services for implementation
- Design and implementation of X on Windows
- Design and implementation of X on mobile devices
- Documentation based on X
- Evaluation of the proposed solution

6.2 Dissemination

Who uses your component or who will use it? Industry projects, EU projects, open source...? Is it integrated into a larger environment? Did you publish any papers?

6.3 Problems Encountered

Summarize the main problems. How did you solve them? Why didn't you solve them?

6.4 Outlook

Future work will enhance Component X with new services and features that can be used ...

List of Acronyms

3GPP	3rd Generation Partnership Project
AJAX	Asynchronous JavaScript and XML

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] 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.

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

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

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] ``Mit technology review: King â 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.

[13] 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.

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

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] J. Pennington, R. Socher, and C. D. Manning, ``Glove: Global vectors for word representation,'' in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pp. 1532--1543, 2014.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

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

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

Annex