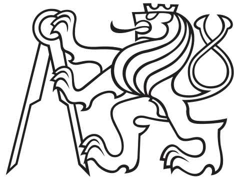
Master Thesis



Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

Contextual Embeddings for Anomaly Detection in Log Files

Prokop Černý

Supervisor: Ing. Jan Drchal, Ph.D.
Field of study: Open Informatics
Subfield: Data Science
May 2021

ČVUT
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Černý** Jméno: **Prokop** Osobní číslo: **466375**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Specializace: **Datové vědy**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Kontextové reprezentace pro detekci anomálií v souborech logů

Název diplomové práce anglicky:

Contextual Embeddings for Anomaly Detection in Log Files

Pokyny pro vypracování:

The task is to develop, implement, and evaluate methods for extracting fixed-size embeddings for log files. Focus on embeddings applicable to the anomaly detection downstream tasks. The methods will be based on related approaches known from NLP.
1) Familiarize yourself with state-of-the-art contextual embedding methods used in the NLP domain.
2) Select and modify an appropriate method to work on log lines (either single lines or several successive log lines).
3) Compare with state-of-the-art algorithms on a downstream task of log anomaly detection. Datasets will be supplied by the supervisor (HDFS, BGL, OpenStack from https://github.com/logpai/loghub are possible choices).
4) This work focuses on embeddings, not on anomaly detection methods - use basic approaches implemented in PyOD (https://pyod.readthedocs.io/) or methods supplied by the supervisor.

Seznam doporučené literatury:

[1] Devlin, Jacob, et al. "Bert: Pre-training of deep bidirectional transformers for language understanding." arXiv preprint arXiv:1810.04805 (2018).
[2] Marek Souček, "Log Anomaly Detection", master thesis, supervisor Jan Drchal, FEE CTU, 2020.
[3] Du, Min, et al. "Deeplog: Anomaly detection and diagnosis from system logs through deep learning." Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. ACM, 2017.
[4] Saxe, Joshua, and Konstantin Berlin. "eXpose: A character-level convolutional neural network with embeddings for detecting malicious URLs, file paths and registry keys." arXiv preprint arXiv:1702.08568 (2017).
[5] Wang, Jin, et al. "LogEvent2vec: LogEvent-to-Vector Based Anomaly Detection for Large-Scale Logs in Internet of Things." Sensors 20.9 (2020): 2451.

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Jan Drchal, Ph.D., katedra teoretické informatiky FIT

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **12.02.2021** Termín odevzdání diplomové práce: **21.05.2021**

Platnost zadání diplomové práce: **30.09.2022**

Ing. Jan Drchal, Ph.D.
podpis vedoucí(ho) práce

podpis vedoucí(ho) ústavu/katedry

prof. Mgr. Petr Páta, Ph.D.
podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

.

Datum převzetí zadání

Podpis studenta

Acknowledgements

I would like to thank my supervisor, Ing. Jan Drchal, Ph.D., for all the help and guidance given to me. Special thanks belong to Jonáš Černý, for the painstaking editorial corrections of this thesis, and to Zuzana Chalupová, for her numerous encouragements stopping me from procrastinating. I also wish to thank my family for the unwavering support they have given me during my studies.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

In Prague, 21. May 2021

Abstract

Anomaly Detection in log data from large computer systems is an area of growing importance over the past few years. As logs are text data, they first must be embedded into vector spaces for performing Anomaly Detection.

This thesis explores the use of current state-of-the-art NLP methods for contextual embedding of log-lines into vectors. Specifically, we have used BERT models, which are Deep Neural Networks, as the base component for our Sentence Encoders. We have used the Inverse Cloze Task for unsupervised Sentence Encoder training on unlabeled publicly available log datasets. The quality of log-line embeddings produced by our encoders was evaluated by performing Anomaly Detection experiments on the labeled HDFS1 log dataset, using the Auto Encoder Temporal Convolutional Network anomaly detection method. We have used fastText embeddings for obtaining the baseline anomaly detection performance.

Our contextual embeddings have not been able to match the quality of the baseline fastText embeddings. Still, they show promise, as the used anomaly detection dataset may not be complex enough to reap the benefits of contextual embedding, but this has not been verified due to the lack of publicly available complex labeled log datasets.

Keywords: vector embedding, logs, anomaly detection, NLP, BERT

Supervisor: Ing. Jan Drchal, Ph.D.
Czech Technical University,
Faculty of Electrical Engineering,
Karlovo náměstí 13, E-406,
Praha 2

Abstrakt

Detekce anomálií v datech logů z velkých počítačových systémů je v posledních letech oblastí rostoucího významu. Logy jakožto textová data musí být nejdříve převedeny (embedovány) na jejich vektorové reprezentace (embeddingy) aby bylo možné provádět detekci anomálií.

Tato práce zkoumá použití moderních metod zpracování přirozeného jazyka (NLP) pro vytváření kontextových reprezentací z logů. Konkrétně jsme použili hluboké neuronové sítě BERT, jako základ našich enkodérů vět. Naše enkodéry byly trénovány bez učitele pomocí úlohy ICT na veřejně dostupných neanotovaných log datasetech. Kvalitu reprezentací z našich enkodérů jsme ověřili provedením experimentů na detekci anomálií v anotovaném log datasetu HDFS1, za použití AETCN detektoru anomálií. Pro získání výchozího bodu kvality detekce anomálií jsme použili reprezentace vytvořené metodou fastText.

Naše kontextové reprezentace nebyly schopné dosáhnout stejné kvality detekce anomálií jako základní fastText reprezentace, ale i tak výsledky vypadají nadějně, jelikož použitý dataset pro detekci anomálií nemusí být dostatečně komplexní pro využití potenciálu nabízeného kontextovými reprezentacemi. Toto nemohlo být ověřeno kvůli nedostatku veřejně dostupných komplexních anotovaných log datasetů.

Klíčová slova: vektorové reprezentace, logy, detekce anomálií, NLP, BERT

Překlad názvu: Kontextové reprezentace pro detekci anomálií v souborech logů

Contents

1 Introduction 1

2 Related Work 3

2.1 Tokenization . . . 3
2.2 Global embeddings . . . 4
2.2.1 word2vec . . . 4
2.2.2 fastText . . . 5
2.3 Contextual embeddings . . . 5
2.4 Transformer . . . 7
2.4.1 Attention . . . 7
2.4.2 Encoder and decoder stacks . . 8
2.5 BERT . . . 9
2.5.1 Architecture . . . 10
2.5.2 Pre-training . . . 10
2.5.3 Fine-tuning . . . 11
2.6 Sentence Embedding . . . 12
2.6.1 Sentence BERT . . . 12
2.6.2 Unsupervised sentence embedding pretraining tasks . . . 14
2.7 Anomaly Detection . . . 15

3 Solution design 19

3.1 Sentence encoder architecture . . 19
3.2 Pretraining task - Inverse Cloze Task . . . 20
3.3 ICT pretraining data creation . . 21
3.3.1 Chunking log lines . . . 22
3.3.2 Creating a target and context from chunk . . . 22
3.3.3 Tokenization and final preprocessing of target and context 23
3.4 ICT pretraining architecture . . . 24
3.5 Evaluation on Anomaly Detection task . . . 26
3.6 Overall solution design . . . 27

4 Datasets 29

4.1 Raw Datasets . . . 29
4.1.1 HDFS1 . . . 29
4.1.2 HDFS2 . . . 30
4.1.3 Spark . . . 31
4.1.4 BGL . . . 31
4.1.5 Hadoop . . . 32
4.1.6 Zookeeper . . . 32
4.2 Anomaly Detection Dataset . . . 33
4.3 ICT Datasets . . . 34
4.3.1 HDFS1 based datasets . . . 34
4.3.2 Mix datasets . . . 35

5 Implementation 39

5.1 Sentence Encoders and ICT networks . . . 40

5.2 ICT data pipeline 40

5.3 Anomaly Detection pipeline 41

6 Experiments 43

6.1 Setup 43

6.2 Results 47

6.2.1 Baseline 47

6.2.2 HDFS1-blocks and HDFS1-time 47

6.2.3 Mix datasets 49

6.2.4 Overall results 50

6.3 Encoder hypotheses 53

7 Exploration of model behaviour 57

7.1 LIME explanation of HDFS1-blocks model behaviour . . . 57

7.2 LIME explanation of Anomaly Detection predictions 60

8 Discussion 65

9 Conclusion 67

A Bibliography 69

B All experiment results 73

C List of Attachements 77

Figures

2.1 Transformer architecture 9

3.1 Proposed Sentence Encoder architecture 20

3.2 Proposed ICT pretraining architecture 25

3.3 Overall solution design 28

4.1 Histograms of tokenized HDFS1 train sentence lengths 35

4.2 Histograms of tokenized Mix dataset sentence lengths 36

6.1 F1-scores when trained on HDFS1 datasets 48

6.2 F1-scores when trained on Mix datasets 50

6.3 F1-score comparison of the dataset types 51

6.4 F1-score comparison of DistilBERT to DistilRoBERTa 51

6.5 F1-scores of different pooling strategies 51

6.6 F1-scores of ICT tower counts . . 52

7.1 LIME HDFS1-blocks prediction probabilities 59

7.2 LIME HDFS1-blocks word weight explanation 59

7.3 Translated Sigmoid 61

7.4 LIME explanation of Anomaly Detection prediction 62

Tables

4.1 Summary of raw used datasets . 29

4.2 Training, validation and test splits of HDFS1 33

4.3 Composition of the Mix dataset 36

6.1 Dataset configurations for experiments 45

6.2 Baseline anomaly detection scores using fastText 47

6.3 Best Sentence Encoders for HDFS1 datasets 48

6.4 Best Sentence Encoders for Mix datasets 50

6.5 Averaged metrics for Sentence Encoder type comparison, ordered by F1-score 52

Introduction

Detecting anomalies in logs is an area of growing importance in large modern computer systems, as the volume of logs keeps growing with computer systems. Detecting anomalous behaviour automatically can significantly help human operators of these systems find causes of issues faster, or even prevent more significant problems down the line by detecting anomalous events at their start before they can become a larger issue.

Logs are typically semi-structured to unstructured text data, the format of which changes rapidly with even minute differences in software versions changing the log structure significantly. Therefore anomaly detection approaches need to be able to deal with these changing environments.

Current anomaly detection methods employ a combination of automatic structure detection for extracting log templates from log lines, such as timestamps, error levels, etc., and common text embedding algorithms such as word2vec [MCCD13, MSC$^+$13], and fastText [BGJM17] to embed the unstructured part of the logs into vectors.

There have been significant breakthroughs in the past few years using Deep Neural Networks (DNN) in various Computer Science tasks, especially in Natural Language Processing (NLP) tasks, such as Machine Translation, Summarization, Question Answering, Text classification, etc. Many of these breakthroughs were enabled by using DNNs as Language Models (LMs) trained on large corpora of unlabeled text to gain understanding of the semantics of human languages. These Language Models can then embed text into high-dimensional vectors upon which solutions to particular downstream tasks can be built.

This thesis aims to explore the possibility of training large Language Models to create text embeddings for use with the downstream task of Anomaly

Detection in log files. The first step is to explore current state-of-the-art contextual embedding methods from the NLP domain. The next step is to select and modify an appropriate method to work on log lines. The final step is to compare the quality of the obtained embeddings to other current embedding methods on a downstream log anomaly detection task. This thesis will not explore anomaly detection methods, as the main focus is on log embedding, and an anomaly detection method provided by the supervisor will be used.

The structure of the thesis is as follows. Chapter 2 presents related work and prerequisites, chapter 3 presents the methodology and proposed solution design, chapter 4 presents the used datasets, chapter 5 presents notes on the implementation of the solution design, chapter 6 presents the experiments and evaluation on the anomaly detection downstream task, chapter 7 touches beyond the main scope of this thesis and explores how to explain the Machine Learning models used in this thesis, chapter 8 presents discussion on the results achieved in this thesis and chapter 9 presents the conclusion and future work.

Chapter 2

Related Work

2.1 Tokenization

One of the first steps when working with text data is to split the input text into smaller manageable pieces called tokens, which can then be one-hot encoded (into vectors of length equal to the total number of possible tokens), upon which further transformations can be done, as tokenization enables working with text-data as with points in vector spaces.

One of the simplest tokenization methods would be to split the input text by spaces, e.g., separate the input text into individual words, given that the language we are working with uses spaces as word separators. This technique is basic, and there are several basic issues. It does not handle punctuation "attached" to words at the end of sentences and creates different tokens for the same word depending on whether it is at the end of the sentence, et cetera.

This approach also disregards any subword information and semantics. Consider for example words "runner", "running" and "runnable". Using simple splitting by spaces, these would all be considered different words, although we see that it could be beneficial to split the words into subwords, here for example "run" and different suffixes ("-ing", "-able", etc.).

WordPiece. One of the currently most used tokenization approaches is WordPiece, by Wu et al. [WSC^{+}16], and now commonly used as a tokenizer for many Neural Network models. It works by initializing its vocabulary with all individual letters in a selected alphabet as initial tokens and creating new tokens by selecting pairs of tokens already in its vocabulary, combining them, and adding the new combined token to the vocabulary if its addition increases the likelihood of the text corpus the tokenizer is being trained

on. This approach usually results in vocabularies where common words are represented whole as their own tokens, while enabling to represent arbitrary words by several tokens, as the single-character alphabet is always included in the vocabulary.

BPE. A similar approach to tokenization called BPE tokenization is presented by Sennrich, Haddow and Birch [SHB15], which works similarly to WordPiece, but chooses pairs of tokens to combine from its vocabulary not by maximizing the likelihood of training data, but combining the pairs with the highest frequency in the training data.

2.2 Global embeddings

A common approach for creating vector representations of text is to find a matrix $E \in \mathbb{R}^{|V| \times d}$, where V is the vocabulary, e.g. finite set of text units (words, most typically tokens obtained from a tokenizer), and d is the dimension of the vector space we want to embed the text in. Each row of E is a d dimensional vector representing a particular word from V. It is desirable that the vectors in E carry some semantic structure, for example that vectors for words "dad" and "father" are similar/close by.

These techniques only have one representation for each word, disregarding the context it is used in, which is disadvantageous because meaning of a word can change dramatically depending on its context.

2.2.1 word2vec

Technique presented by Mikolov et al. over two papers [MCCD13, MSC+13] builds vector representations of words from a given vocabulary. This model is designed as a shallow two-layer neural network trained on the Skip-Gram task, which tries to predict the context (words) surrounding a word. This is based on the assumption that the meaning of a word is strongly related to its surrounding.

The shallow neural network contains a linear layer of the shape (# words in vocabulary $\times$ output vector dimension), as it is a projector from one-hot encoding of words in the vocabulary to the vector space into which we want to project the words. The resulting weight matrix extracted from the layer dictates the vector embeddings for each word, as each row of the matrix is the vector representation of each word.

A downside of this technique that it does not handle out-of-vocabulary words.

2.2.2 fastText

Improvement of word2vec by Bojanowski, Grave, Joulin and Mikolov [BGJM17], with the goal of enriching the vector representations with subword information.

To achieve this, the words are broken into bags of overlapping character n-grams, which are n -tuples of characters, for example, the word "where" using $n = 3$ would be tokenized into a bag of triples "<wh", "whe", "her", "ere", "re>". The symbols "<" and ">" denote the start and end of a word, respectively.

A word is then represented as a bag of these n-grams, upon which a Skip-Gram task is trained, as in word2vec. The final word embeddings are sums of the learned vectors of each n-gram the word is comprised of.

Using n-grams enables fastText to represent even unknown words in addition to understanding prefixes and suffixes.

2.3 Contextual embeddings

Although global embeddings brought much success in dealing with NLP tasks, their inability to take the context of a token into account limits their usability, as each token only has one vector representation.

Contextual embeddings take the whole sequence into account to create an embedding of a token from the sequence, e.g. the contextual embedding e of the $i - th$ token in sequence $\mathbf{s}$ is obtained as a function of the whole sentence, e.g. $e_{s_i} = f(s_1, s_2, ..., s_n)$. The contextual embedding function f usually includes some non-contextual token embedding as a first step to transform the text into some meaningful vector space upon which the contextual representation is built (as noted by Liu, Kusner and Blunsom [LKB20] in their survey on contextual embeddings).

ELMo. Model by Peters et al. [PNI+18] which creates contextualized embeddings by using a bidirectional Language Model, based on LSTMs (a kind of RNN, introduced by Hochreiter and Schmidhuber [HS97]). The bidirectional model is created by concatenating vectors from an L-layer left-to-right

(forward) LSTM and an L-layer right-to-left (backward LSTM). The final contextual embedding for a token is then created as a task-specific affine combination of all L hidden concatenated vectors. In the paper, ELMo-produced embeddings are typically used concatenated with the global embedding of the same tokens so that both representations can be used to possibly strengthen the downstream tasks.

GPT. GPT by Radford, Narasimham, Salimans and Sutskever [RNSS18] is one of the first approaches to successfully apply transfer learning in an NLP setting. The main model is based on a part of the Transformer architecture (by Vaswani et al. [VSP+17], see section 2.4), specifically its core is a stack of Transformer Decoders (see section 2.4.2), which are left-to-right components, where tokens can only attend to its left context. This architecture, combined with a two-stage training process, where the model is pretrained on an unsupervised Language Modelling task and then fine-tuned on a supervised downstream task, yielded state-of-the-art results on many NLP tasks.

GPT2. Followup work by Radford et al. [RWC+19] is architecturally the same as GPT, but the model is larger and is trained on a much larger dataset, called WebText, which was created by scrapping outbound links from Reddit. In this paper, the authors posed that a Language Model pretrained unsupervised on a vast amount of text begins to learn some common NLP tasks without specific direction to do so. They validated this by using GPT2 on ten NLP datasets in a zero-shot setting, and the model performed strongly on several of them, which shows the potential of unsupervised pretraining of large models on very large datasets.

BERT. Compared to GPT, which attends to the left context, and ELMo, which although it is bidirectional does not include interactions between the left and right context, BERT by Devlin, Change, Lee and Toutanova [DCLT18] is an innately bidirectional model which can consider information from both contexts at the same time. Similarly to GPT, the strength of this model lies in its ability to be successfully used in transfer learning, with a similar workflow of long pretraining on large unlabeled text datasets and much less computationally demanding supervised fine-tuning. It's architecture is based on Transformer Encoders (see section 2.4.2). Its input/output format is flexible, allowing it to easily integrate into various downstream tasks, where it achieved many state-of-the-art results. The model is explored in more detail in section 2.5. Due to its surprising empirical strength, many variants of BERT were created after the release of the original paper.

2.4 Transformer

Introduced by Vaswani et al. [VSP+17], the Transformer is a Neural Network architecture primarily designed for sequence-to-sequence tasks (such as translation), with the goals of improving performance over Recurrent Neural Network (RNN) models. This is achieved by relying solely on Attention instead of using recurrence.

2.4.1 Attention

Attention (originally introduced by Bahdanau, Cho and Bengyo [BCB14]) is a mechanism to handle long-term dependencies which allows the neural network to attend to specific parts of the input for the current token. A high-level example is figuring out what subject the word "it" refers to in the sentence "The chicken crossed the road because it was hungry.".

The attention function can be described as a mapping from a query and a set of key-value pairs to an output, which is a weighted combination of values, weighted by the softmax of the product between the query and key corresponding to a particular value.

Scaled Dot-Product Attention

The specific Attention used in the Transformer, called scaled dot product attention, can be seen in eq. (2.1), where $Q \in \mathbb{R}^{n \times d_q}$, $K \in \mathbb{R}^{m \times d_q}$, $V \in \mathbb{R}^{m \times d_v}$, $d_q, d_v, n, m \in \mathbb{N}$. d_q is the dimensionality of the query and key vectors, d_v is the dimensionality of value vectors, and n, m are counts of query and key & value vectors respectively. The attention output is of the shape $n \times d_v$.

$$Attention\,(Q, K, V) = softmax\left(\frac{QK^\intercal}{\sqrt{d_q}}\right)V \tag{2.1}$$

The query-key dot product is scaled by $\frac{1}{\sqrt{d_q}}$ to counteract the possible magnitude of high-dimensional dot-products before applying *softmax*, to prevent pushing the function into extreme regions.

Multihead Attention

The transformer model internally works with vectors of some dimension d_m, but instead of just using attention directly with $d_q = d_v = d_m$, the authors found it beneficial to use multiple separate attention heads, with their own representational space and concatenate their results together to get higher quality attention mechanism.

$$\begin{aligned} \textit{MultiheadAttention}\,(Q, K, V) &= \textsc{Concat}\,(\text{head}_1, ..., \text{head}_k)\, W^O \\ \text{where head}_i &= \textit{Attention}\left(QW_i^Q, KW_i^K, VW_i^V\right) \end{aligned} \tag{2.2}$$

Where $W_i^Q, W_i^K \in \mathbb{R}^{d_m \times d_q}$, $W_i^V \in \mathbb{R}^{d_m \times d_v}$ for $i \in \{1, ..., k\}$ are projection matrices for each head, and $W^O \in \mathbb{R}^{k \cdot d_v \times d_m}$ is the matrix projecting all the concatenated attentions back into the d_m dimensional space of the model.

In the paper, $k = 8$ attention heads were used, and the attention dimensionality was set to $d_q = d_v = \frac{d_m}{8}$

2.4.2 Encoder and decoder stacks

The Transformer is composed of a stack of encoder blocks and a stack of decoder blocks. The whole architecture, including the inner details of the encoder and decoder blocks, is shown in fig. 2.1.

Transformer Encoder. An encoder block is composed of two sublayers, first is the Multihead Attention, where the block input vectors serve as queries, keys, and values at the same time. The second layer in the encoder block is a Fully Connected Feed Forward network, which comprises two linear layers with ReLU activation between them. The input and output dimensions of the FCFFN are d_m, but the inner dimension between the two linear layers is some dimension d_{ff} typically higher than d_m. Each of the two sublayers has a residual connection around it, and it is followed by layer normalization.

Transformer Decoder. A decoder block is very similar to the encoder block, with additional multihead attention in the middle. The first sublayer in a decoder block is again multihead attention, again having the block input vectors serve as queries, keys, and values at the same time, with the difference that the future positions are masked (decoder is mainly used for generating text - translations, and cannot have access to following words, e.g., it attends only to the left context). The second sublayer is another multihead attention, but here the output of the encoder stack serves as the queries and keys, while

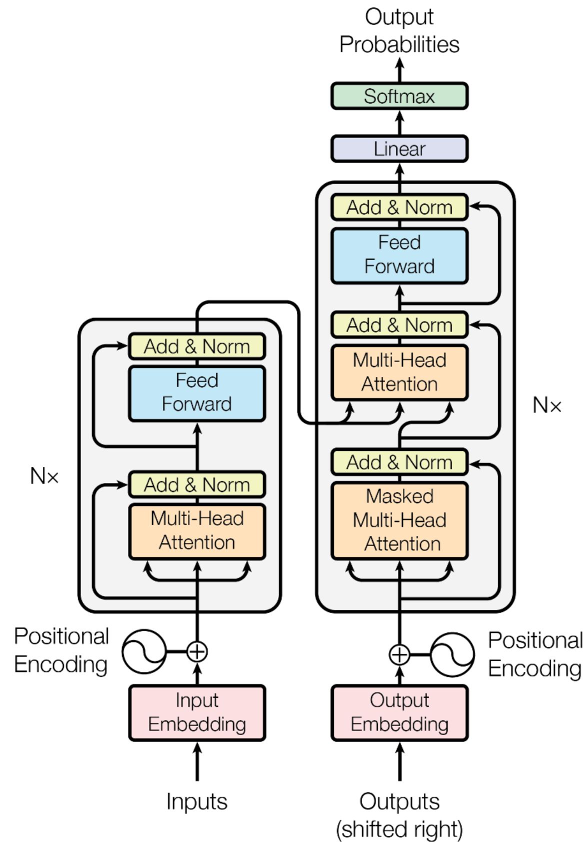


Figure 2.1: Transformer architecture. Encoder block on the left and decoder block on the right. Inputs to the middle multihead attention in decoder block always come only from the top-most encoder in the encoder stack. Figure from [VSP+17].

the output of the decoder self-attention serves as the values. The last layer in the decoder block is the same Fully Connected Feed Forward network as in the encoder. All sublayers have residual connections and layer normalization applied to them.

2.5 BERT

The regular Transformer proved to be groundbreaking in the field of sequence to sequence tasks, with its Encoder-Decoder architecture. Devlin, Chang, Lee and Toutanova [DCLT18] posed that the Encoder stack from the Transformer holds a lot of information about the language it is trained on and could be used by itself as a general pretrained Language Model.

To that end, they have created BERT - Bidirectional Encoder Representa-

tions from Transformer, which is a stack of Transformer Encoder blocks (see section 2.4.2), pretrained on large unlabeled corpora. With just small task-specific networks (often single-layer) attached to it and a bit of fine-tuning on the specific tasks, it achieves state-of-the-art results on many NLP tasks such as Question Answering, Textual Entailment, etc.

2.5.1 Architecture

Standard BERT is a stack of L Transformer Encoder blocks, with hidden dimension d_m, and the number of heads in multihead attention A. In the paper, two configurations were explored the most, $\mathbf{BERT_{BASE}}$, with $L = 12$, $d_m = 768$ and $A = 12$, with a total of 110M parameters, and $\mathbf{BERT_{LARGE}}$, with $L = 24$, $d_m = 1024$ and $A = 16$, with a total of 340M parameters. The hidden dimension d_{ff} of the FCFNN inside the encoder block was always set to $d_{ff} = 4d_m$.

Besides the stack of Transformer Encoders, BERT uses the WordPiece tokenizer [WSC+16] and a single layer embedding the tokens into the d_m dimensional space the Encoder stack expects as input. Various layers/networks can then be attached to the output of the Encoder stack for solving various downstream tasks.

2.5.2 Pre-training

Two unsupervised training tasks were devised to pretrain the network so it could be used as a general Language Model and enable the usage of BERT for transfer learning in NLP.

The input representation for BERT was designed to be flexible to allow for a large variety of downstream tasks. It accepts tokenized WordPiece tokens, with each input token sequence prepended with the special [CLS] (classification) token and ending with a special [SEP] token.

To handle sentence-pair task (where sentence is an arbitrary text sequence, from a sentence or a larger chunk of text), the input for two sentences A, B is constructed into a single token sequence as "[CLS] *tokens of sentence A* [SEP] *tokens of sentence B* [SEP]".

After the construction of token sequences, each token is embedded into a d_m dimensional vector with which the network operates. Each of the token embeddings gets summed with a positional embedding, coding the position of each token in a sentence, and for sentence-pair tasks additionally summed

with segment embeddings signifying whether the token belongs to sentence A or B.

Masked Language Modelling (MLM)

The first pretraining task is Masked Language Modelling, originating from the Cloze task [Tay53]. As BERT is bi-directional, each word (token) can indirectly see itself, which would enable it to trivially predict each word.

To solve this problem, a certain percentage of input tokens gets replaced by a [MASK] token, and the network is trained to predict the original word present at the masked position. To predict the tokens, the final hidden vector for the token to predict gets fed into a Softmax layer over the vocabulary.

The authors randomly masked 15% of non-special tokens (e.g., excluding [CLS], [SEP]) during training, out of which 80% gets replaced with the [MASK] token, 10% are replaced with a random token, and the remaining 10% is left unchanged. This is done to mitigate the mismatch between pretraining and finetuning, as the [MASK] token never appears in the data during finetuning.

Next Sentence Prediction (NSP)

As sentence-pair level tasks are very common in NLP, the Next Sentence Prediction task was used as a simple pretraining task, data for which can be trivially constructed from a large corpus of training text. During creation of training data, for sentence A, sentence B is the actual sentence following A 50% of the time, and a random sentence otherwise.

To predict whether B is the following sentence, a shallow two-layer classifier network is attached to BERT, consuming the final hidden vector corresponding to the classification [CLS] token. This sort of pretraining conditions the network to treat the [CLS] token as an aggregate sequence level embedding, although without fine-tuning, the output of a [CLS] token only pretrained on NSP is not an ideal sentence representation.

2.5.3 Fine-tuning

Joint end-to-end fine-tuning

After pretraining, BERT can be easily finetuned to specific tasks by appending task-specific output layers, which can be very small, typically even single-layer.

For sentence-level tasks, classifiers can be appended to the final hidden vectors corresponding to the [CLS] token, e.g., for sentiment analysis, textual entailment, etc. Work is also being done on using concatenations of the hidden vectors from several last encoder blocks, instead of only using the final hidden vector [DCLT18].

For token level tasks, e.g., Question Answering, Named Entity Recognition, Sequence Tagging, etc., the per-token representations are passed into an output layer.

Feature based fine-tuning

In cases where joint training of the whole stack of BERT and a more complex downstream task solution, BERT can be used to precompute embeddings of text sequences beforehand, which can then be fed as inputs to downstream solutions. This can present significant computational savings, so input data do not have to be processed multiple times.

Although not strictly necessary, it is better for this approach to perform additional task-specific pretraining on the (NSP,MLM)-pretrained BERT model to obtain better task-specific embeddings.

2.6 Sentence Embedding

For anomaly detection in logs, our goal is to embed whole log-lines into some vector space. As a log-line can be thought of as a sentence, we want to explore the various current sentence embedding approaches.

For global embedding methods, such as word2vec (section 2.2.1) and fastText (section 2.2.2), the most commonly used approach is to average the token embeddings for the whole sentence, with optional vector normalization done prior to the averaging.

For contextual embeddings, there are more approaches that we will discuss in this section.

2.6.1 Sentence BERT

Introduced by Reimers and Gurevych [RG19], Sentence BERT (SBERT) is an approach for creating semantically meaningful sentence embeddings from BERT-like models. The paper poses that the common practice of simply taking the [CLS] token embedding, or pure averaging of all token embeddings

for a sentence is not a good sentence embedding with regards to preserving semantic meaning in the vector space, e.g., that semantically similar sentences do not get mapped to similar coordinates.

The paper presents a supervised pretraining approach using a siamese and triplet network architectures to solve this problem. The first modification is the addition of a pooling layer onto the output of BERT, which can either average the output token embeddings, select element-wise maximum for each dimension over all output token embeddings, or output the embedding of the [CLS] token.

The proposed approach to making SBERT presumes the existence of a labeled dataset of sentence pairs or triples, which can have several formats, depending on which the pretraining task is selected. The pretraining task then dictates the model architecture. Depending on whether the dataset consists of sentence pairs or triples, a siamese or a triplet model architecture is chosen. The sentences are passed through BERT individually, and only after an embedding is created for each of them is the objective function computed. The sentences pass through the same BERT network, but the architecture can be pictured as two or three separate BERT towers with tied weights, outputs of which are then used as inputs to an objective function.

Classification objective. This pretraining task is used with datasets comprising of labeled sentence pairs (example labels can be the relationship of the sentences, e.g. "contradiction", "entailment", "neutral"). A siamese architecture is then used, creating embedding vectors a, b for the input sentences, which are then concatenated together and also with their elementwise difference $|a - b|$, and passed through a linear layer to softmax trying to predict the correct label.

Regression objective. Given that a dataset contains sentence pairs labeled with their distance (cosine similarity was used in the paper), this task again uses a siamese architecture, producing embeddings u, v for the sentences, and computing cosine distance of the embedding vectors. Mean squared error is then minimized between the computed cosine distance and the labeled distance.

Triplet objective. Given three sentences, a being an anchor sentence, p being positive sentence, and n negative sentence, the task is to tune the network to produce embeddings e_a, e_p, e_n such that $\|e_a - e_p\| < \|e_a - e_n\|$, e.g., the distance between the anchor sentence and positive sentence is smaller than the distance between the anchor and the negative sentence. This task requires a triplet model architecture, e.g., three passes through the embedding BERT network, one for each sentence.

These pretraining approaches are shown to produce quality sentence embeddings by achieving top results on several benchmark datasets for sentence embedding, dealing with Semantic Text Similarity, clustering, and information retrieval via semantic search.

The downside of this approach is the necessity to have a quality labeled sentence-pair dataset, which is not always feasible. Although there are approaches for creating higher quality labeled datasets by augmentation and automatic labeling, as explored in Augmented SBERT by Thakur, Reimers, Daxenberger and Gurevych [TRDG20], these techniques still require the existence of at least a small labeled sentence pair dataset.

2.6.2 Unsupervised sentence embedding pretraining tasks

Given sentence embedding applicability for large-scale semantic search, several pretraining tasks were explored by Chang et al. [CYC+20] in the context of query-document search.

Their explored task was to find a set of relevant documents from a large corpus given a query (usually a natural language sentence). This problem can be solved by embedding the query and documents in the same vector space which captures the semantic meaning and finding the documents closest to the query in the vector space.

Their proposed architecture is to create two separate encoders, one for the queries and one for the documents. In the paper, each encoder was an instance of a BERT network with a linear layer applied to the output of the [CLS] token. The BERTs did not share parameters.

They present three pretraining tasks, the Inverse Cloze Task (ICT), Body First Selection (BFS), and Wiki Link Prediction (WLP). BFS and WLP are original pretraining tasks from [CYC+20]. The training data for each of the tasks is presumed to consist of positive query-document pairs, e.g., the data consists only of pairs of documents and queries that relate to each other.

Formally, they are trying to maximize the log likelihood of a document given a query, e.g. $\max \sum_{(q,d)\in\mathcal{T}} \log p\left(d \mid q\right)$, where $p\left(d \mid q\right)$ is the softmax of similarity scores between the query q and the correct document d in relation to the similarity of q with all other documents d' (which present the negative examples) in the training data.

Computing this full softmax is very expensive because of the need to compute the similarity between all training documents and the current query. To get around this during training, Sampled Softmax is employed, where the

negative examples are the other documents present in the same mini-batch of training query-document pairs.

Inverse Cloze Task (ICT). Given a passage of n sentences (for example a paragraph), the query q is a randomly selected sentence from the passage, and the document d is the remaining $n-1$ sentences from the passage. This task captures the semantic meaning of a sentence related to its surrounding context. It was originally proposed by Lee, Chang and Toutanova [LCT19].

The remaining tasks were created specifically with the usage of Wikipedia as the source dataset.

Body First Selection (BFS). To capture non-local semantics outside of a contiguous passage of sentences, this task selects the query q as a random sentence from the first paragraph of a Wikipedia page, and the document d is a random passage from the remainder of the same Wikipedia page. Given that the first paragraph on a Wikipedia page is usually a summary, this task is expected to capture core semantics of the text.

Wiki Link Prediction (WLP). To capture even longer-range semantics, the query q is again selected as a random sentence from the first paragraph of a Wikipedia page, but the document d is selected as a passage of text from a different Wikipedia page containing a hyperlink to the page the query q comes from.

Training data for each of the tasks were generated from Wikipedia articles, but can be generated from other data sources without too much hassle for unsupervised sentence embedding pretraining. ICT is the most straightforward pretraining task to apply to various source datasets.

The authors pretrained the query and document encoders using a mixed dataset consisting of data from all three tasks and evaluated their approach on common Question Answering benchmark datasets, where they have achieved state-of-the-art results, beating previous techniques.

2.7 Anomaly Detection

Anomaly Detection techniques are methods for detecting *anomalous* data points in various kinds of data. Typical applications include fraud detection in finances, preventing cyberattacks by analyzing traffic, et cetera. It is also an important tool when working with large-scale computer systems, where

detecting unwanted anomalous behaviour in log-files quickly is very desirable to prevent system interruptions and maintain high service availability.

Anomaly detection methods can be categorized into supervised, semi-supervised, and unsupervised methods depending on the kind of needed training data.

Supervised anomaly detection methods assume that there exists a label for each training point specifying whether it is anomalous or not. These methods typically try to model both anomalous and normal data. Although supervised methods are typically the best at detecting anomalies, collecting the necessary training data is usually difficult. Typical supervised approaches for anomaly detection are SVMs [LZXS07] and decision trees [CZL+04].

Semi-supervised anomaly detection method assumes that anomalous data points are significantly different from normal data points. Using this assumption enables them to only need normal data for training, which can be significantly easier to obtain compared to fully labeled data. One-class SVMs [SWS+99] and autoencoders [ZP17] are typical examples of semi-supervised anomaly detection methods.

Unsupervised methods do not need labeled data and assume that anomalies are infrequent compared to normal data and are intrinsically different so that they can be detected. A typical unsupervised approach is the Isolation Forest [LTZ08], which identifies anomalies by isolating them from normal data.

In the past few years, there have been various successes in using Deep Neural Networks for anomaly detection. Du, Li, Zheng and Srikumar [DLZS17] explore using LSTMs and custom log-parsing to detect anomalies.

Souček [Sou20] finds that using NLP approaches for log-parsing in conjunction with DNNs provides promising results. Specifically, fastText sentence embeddings were used to embed log-lines into a high dimensional vector space.

Motivated by Souček's findings, this thesis explores the effect of contextual embeddings on the task of anomaly detection in log files. To isolate the effect the contextual embeddings have, we will use the AETCN anomaly detection method, from Koryťák's [Kor21] exploration of neural network architectures for anomaly detection in logs, as the single fixed anomaly detection method.

AETCN. Auto Encoder Temporal Convolution Network [Kor21] is a semi-supervised anomaly detection method based on Temporal Convolutional Networks [LFV+16] and Autoencoders. An autoencoder is a neural network architecture trying to learn an identity function by learning to compress an input vector with a high dimensionality d_H into a hidden vector with a low dimension d_L, and then reconstruct the original vector of dimension

d_H from the d_L dimensional hidden vector. The AETCN is a special kind of autoencoder that works with sequences of data, which in our case are embeddings of several log-lines, with the detector then predicting whether the whole sequence is anomalous or not. The AETCN anomaly detection is trained on normal data, and it is presumed that the reconstruction error on them will be low. To detect anomalies, a threshold on the reconstruction error is used, such that any vectors with a higher reconstruction error than the threshold will be classified as anomalous. The threshold selected is typically one which maximizes the F1-score (defined below, see eq. (2.5)) on a validation dataset.

There are three metrics typically used for measuring anomaly detection quality - precision, recall, and F1-score, as known from binary classification, with the anomalous label being the positive class.

Precision (eq. (2.3)) is the proportion of true detected anomalies from all data points the detector labeled as anomalous. Recall eq. (2.4) is the proportion of true detected anomalies from all truly anomalous data points in the dataset. It is not recommended to only use precision or recall as an observed metric, as for example it is possible to have perfect recall by labeling all data as anomalous, while achieving very high precision by only labeling very few very likely data points as anomalous.

As precision and recall usually have an inverse relationship, the F1-score (eq. (2.5)) was devised to combine these two metrics, being defined as the harmonic mean of precision and recall. The F1-score is a one-sided measure, as it does not consider true negative counts at all, but in the context of anomaly detection, where there is a heavy emphasis on the positive class, it is used as a standard basic quality measure for comparing anomaly detectors.

$$\text{precision} = \frac{\text{true positives}}{\text{true positives} + \text{false positives}} \tag{2.3}$$

$$\text{recall} = \frac{\text{true positives}}{\text{true positives} + \text{false negatives}} \tag{2.4}$$

$$\text{F1-score} = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} \tag{2.5}$$

Chapter 3

Solution design

To create contextual embeddings of log-lines, we propose to use BERT-like architectures, as they are the current state-of-the-art methods in NLP tasks and contextual sentence embeddings. In the rest of the thesis, we will use the BERT model to mean any BERT-like network unless explicitly stated.

Using BERT models allows us to leverage pretrained network checkpoints, which are saved weights of a BERT network already trained on large text corpora. These checkpoints already contain some learned knowledge of semantics of the language (or languages) they were trained on. Using the pretrained checkpoint allows us to significantly shorten the training task for learning the semantics of our log-line text data, as well as hopefully leverage the language semantics already learned by the checkpoints, which were learned during their long pretraining on large compute clusters.

3.1 Sentence encoder architecture

First building block of our solution is the design of a sentence encoder for encoding log-lines. Our proposed architecture, which can be seen in fig. 3.1, is inspired by Chang et al. [CYC+20] (see section 2.6.2).

Prior to feeding a log-line through a network, it must first be tokenized (see section 2.1). Because we are using pretrained BERT checkpoints, we are forced to use the tokenizer which was used to pretrain the given checkpoint. As described in section 2.5.2, after tokenization each sentence is prepended with the [CLS] token and appended with the [SEP] token before being fed into a BERT network.

To create an embedding for a sentence (log-line), we feed its tokenized form

Figure 3.1: Proposed sentence (log-line) encoder architecture showing the pipeline for a single sentence

through a BERT network, applying a pooling operation on the hidden state vectors created by the network (usually only on the hidden vectors of the last transformer block) followed by feeding the pooled vector through a linear layer to transform the potentially differently sized vector from the pooling operation to the chosen output dimension. The output of the linear layer is the desired contextual embedding of a log-line.

Pooling

We use several pooling operations. The simplest one, which was used by Chang et al. is the selection of the last hidden state vector corresponding to the [CLS] token.

Another pooling operation we use is averaging last hidden state vectors of all non-special tokens (e.g., vectors for all tokens except [CLS] and [SEP]), as proposed by Reimers and Gurevych [RG19] (see section 2.6.1).

The last pooling operation we will explore is concatenation of hidden vectors for the [CLS] token from the last n Transformer blocks the network is composed of, which is an approach suggested by Devlin et al. [DCLT18] in the original BERT paper.

3.2 Pretraining task - Inverse Cloze Task

To obtain meaningful log-line embeddings using our proposed sentence encoder architecture, we must first pretrain it with a pretraining task semantically meaningful for the "language" of log-lines. Such pretraining task must be unsupervised, with the ability to easily generate the needed task data from the large available corpora of raw logs from production systems.

Finding such a training task is difficult, as most tasks designed for sentence embedding either need labeled datasets of pairs of sentences, labeled by either the relationship of the sentences or even their numeric similarity (SBERT [RG19] see section 2.6.1).

Even relaxing the need to have labeled datasets, other pretraining tasks,

such as BFS and WLP (see section 2.6.2) require that some higher level structures, such as paragraphs or links between source documents are present in the raw data to create the training dataset. This makes them unsuitable for use with raw log datasets, as logs are typically just a stream of log-lines, not containing paragraph-like structures.

Considering that, we decided to use the Inverse Cloze Task ([LCT19], as briefly described in section 2.6.2) as the pretraining task for log data, as it only needs passages of sentences to create training data. To be able to use ICT, we must make an assumption that related log-lines happen at similar times, e.g., we must presume semantic temporal-locality. Although in current large-scale computer systems, many different things may be happening simultaneously, we argue that given the nature of how log-lines are inspected manually, by sequential examination, the assumption of temporal-locality does at least partially hold.

Given a dataset of N positive target and context pairs $\mathcal{T} = \{(t_i, c_i) \mid i = 1, ..., N\}$, the ICT objective is defined as maximizing the log-likelihood of the conditional probability of a context c on the target t, as seen in eq. (3.1)

$$\max \sum_{(t,c)\in\mathcal{T}} \log p(c \mid t) \tag{3.1}$$

The conditional probability is defined as a softmax in eq. (3.2), where $\mathcal{C}$ is the set of all possible contexts, $\phi(x)$ and $\psi(x)$ are the target and context sentence encoders respectively.

$$p(c \mid t) = \frac{\exp\left(\phi(t)^\intercal \psi(c)\right)}{\sum_{c'\in\mathcal{C}} \exp\left(\phi(t)^\intercal \psi(c')\right)} \tag{3.2}$$

Computing the full softmax as in eq. (3.2) is infeasible. To get around this, we employ the Sampled Softmax, where the contexts in the denominator are taken only from the current batch of B pairs of targets and contexts. This can be seen in eq. (3.3), where $\mathcal{C}'$ is the set of contexts in the current batch and $|\mathcal{C}'| = B$.

$$p_{sampled}(c \mid t) = \frac{\exp\left(\phi(t)^\intercal \psi(c)\right)}{\sum_{c'\in\mathcal{C}'} \exp\left(\phi(t)^\intercal \psi(c')\right)} \tag{3.3}$$

3.3 ICT pretraining data creation

To employ ICT, we need to have passages of several sentences to create pairs of a target sentence and its context (known as query and document

respectively in the nomenclature used in section 2.6.2, but we will use target and context in the rest of this thesis). Assuming the temporal-locality introduced above holds, we can create passages of n sentences by taking n consecutive lines in the raw source log file and grouping them together to create a single passage, to which we will refer to as a chunk.

3.3.1 Chunking log lines

The general chunking operation can be seen in eq. (3.4), taking an input log file consisting of m lines, which is then chunked into C chunks of length n, where $C \in \mathbb{N}_0$, $C \leq \frac{m}{n}$ and $n \lll m$. This leads to not using the last ($m \bmod n$) lines when $Cn \neq m$, e.g., dropping the last possible chunk, but we argue that dropping a single chunk out of a very large number of chunks does not matter, and it allows us to simplify further data preparation when we have chunks of uniform length, although nothing is preventing us from using chunks of uneven lengths.

$$\underbrace{(l_1, ..., l_m)}_{m \text{ log lines}} \rightarrow \Big(\underbrace{(l_1, ..., l_n)}_{\text{Chunk } 1}, \underbrace{(l_{n+1}, ..., l_{2n})}_{\text{Chunk } 2}, ..., \underbrace{(l_{(C-1)n+1}, ..., l_{Cn})}_{\text{Chunk } C}\Big) \tag{3.4}$$

Alternative chunking. Although typically chunking is done on log-lines which are ordered by time, if there is additional information, it can be used to group log-lines into more related groups prior to chunking, such as grouping the log lines belonging to the same machine or logs coming from the same application. This depends on the kind of anomalies we wish to detect and whether we presume that an anomaly can span multiple applications / machines / sessions or not. The HDFS1 dataset (section 4.1.1) contains block IDs that can and were used for grouping during the creation of our HDFS1-blocks ICT dataset (section 4.3.1).

3.3.2 Creating a target and context from chunk

We now need to obtain a target and a context for ICT. This is straightforward once we have a chunk. Given a chunk $(l_1, l_2, ..., l_n)$ containing n lines, we select a positive integer $i \leq n$, and select the target $t = l_i$, with the context c being the remaining lines from the chunk, with the target line being kept in only a small percentage of the time. This can be seen as two operations on a chunk, with the target selection given an index being defined in eq. (3.5a),

and context creation defined in eq. (3.5b).

$$\textsc{GetTarget}((l_1, ..., l_n), i) = l_i \tag{3.5a}$$

$$\textsc{GetContext}((l_1, ..., l_n), i) = \begin{cases} (l_1, ..., l_{i-1}, l_{i+1}, ..., l_n), & \text{with probability } p. \\ (l_1, ..., l_n), & \text{otherwise.} \end{cases} \tag{3.5b}$$

The context c contains the target l_i with a small probability. This is done to increase the possible semantic similarity between the target and context vectors (after the target and contexts have been encoded by a sentence encoder). But the context c does not contain l_i with probability p, with p being a high probability, in our experiments set to $p = 0.9$.

Each chunk can be (and is) used for creation of multiple target-context pairs by selecting a different target log-line from the chunk each time.

3.3.3 Tokenization and final preprocessing of target and context

Tokenization. After chunking and creation of targets and contexts from chunks, the data still needs to be tokenized to enter the sentence encoder. This is straightforward with the target sentence, as it can be tokenized straight away and sent through the sentence encoder. The context needs a bit more care, as it is a list of sentences. Each sentence in the context is tokenized as usual into a list of tokens, which are then collected into a list of lists of tokens, representing the tokenized context.

Flattening of tokenized context and truncation

BERT models have a maximum tokenized input sentence length, and therefore very long tokenized sentences need to be truncated. Tokenized target sentences are simple, if they are over the limit (which is usually not the case for a single log line), they are simply truncated to the maximum input length, and the rest is thrown away.

The situation is more complicated with tokenized contexts, as they are not a flat list of tokens, but a list of tokenized representations of each context sentence. Therefore we need to decide on a flattening strategy in conjunction with the maximum input length, as a context will typically have more total tokens than the maximum input length.

Concatenation to max length. The first strategy we use is simple, we simply concatenate the tokenized context sentences one after another until the maximum input length is reached, and the rest of the context is thrown away.

Smart Length Averaging. Our other approach considers the maximum input length and tries to uniformly represent each context sentence in the flattened and truncated representation of the context. We want to compute a threshold on the maximum amount of tokens to take from each sentence so they fit into the maximum input length.

The simplest threshold $T = \left\lfloor \frac{maximum\ input\ length}{\#\ of\ sentences\ in\ context} \right\rfloor$ works when all sentences in the context are long, but usually there are sentences shorter than this initial threshold, so we want to find the maximum threshold on the length of each sentence, such that the total length of all tokenized context sentences after truncation by this threshold is as close to the maximum input length as possible. The algorithm for finding such a threshold is described in algorithm 1.

Algorithm 1 Compute Smart Average Threshold

1: **function** SMARTAVGTHRESHOLD($contextLengths$, $maxLength$)
2: $n \leftarrow$ Length of $contextLengths$, e.g. # of sentences
3: $prevThreshold \leftarrow$ Null
4: $curThreshold \leftarrow \left\lfloor \frac{maxLength}{n} \right\rfloor$
5: **while** $curThreshold \neq prevThreshold$ **do**
6: $short \leftarrow \{i \mid i = 1, ..., n\, , contextLengths[i] < curThreshold\}$
7: $lenToDistribute \leftarrow maxLength - \sum_{i \in short} contextLenghts[i]$
8: $cntMoreThanCurThresh \leftarrow |\{i \mid i = 1, ..., n\, , i \notin short\}|$
9: $prevThreshold \leftarrow curThreshold$
10: $curThreshold \leftarrow \left\lfloor \frac{lenToDistribute}{cntMoreThanCurThresh} \right\rfloor$
11: **return** $curThreshold$

After finding the smart average threshold T, we will concatenate all sentences from the context, taking only up to T tokens from each sentence.

3.4 ICT pretraining architecture

To pretrain the log-line sentence encoder we need to create a larger composite neural network to use the pairs of ($target$, $context$) created from our raw log datasets, where $target$ and $context$ are both lists of tokens, e.g., the $context$ is already flattened and truncated by some method from section 3.3.3.

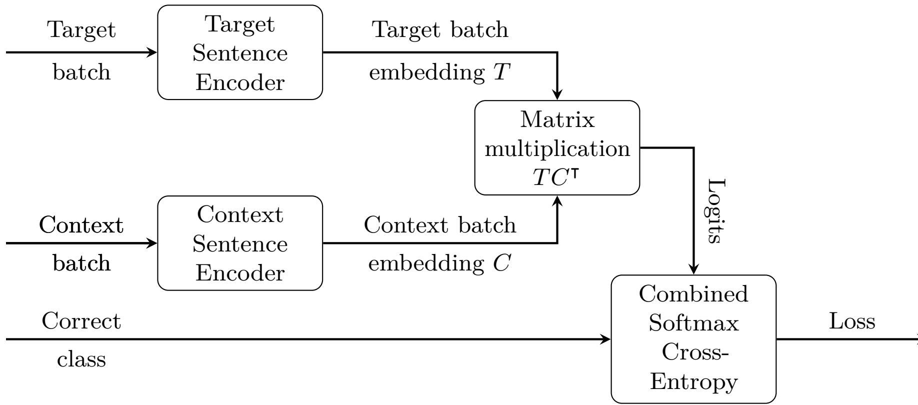


Figure 3.2: Proposed ICT pretraining architecture

The (*target*, *context*) pairs are batched into batches of size B as follows. $(t_1, t_2, ..., t_B)$ are the targets and $(c_1, c_2, ..., c_B)$ are the contexts for the current batch. For each target t_i, $i = 1, 2, ..., B$, the context c_i is the positive example for t_i and the other contexts $\{c_j, | \ j = 1, 2, ..., B, j \neq i\}$ are the negative examples for t_i, as the sampled softmax from eq. (3.3) can be thought of as performing a classification task with B classes.

The ICT network architecture can be seen in fig. 3.2. The network takes three inputs for the current batch, the targets, the contexts and the correct class labels $\mathbf{k} = (k_1, k_2, ..., k_B)$, where k_i is the index of the context c_{k_i} that corresponds to the target t_i. As we have described, due to the way we create batches, the corresponding index $k_i = i$, but we created our network architecture in such a way that it could handle even different input batch structures.

The targets and contexts for the current batch are then separately passed through a Sentence Encoder (as described in section 3.1). The targets and contexts are all padded so they all have the same length within their group, resulting in inputs to the sentence encoders having shapes (B, max_tokens_{target}) and $(B, max_tokens_{context})$ for the target and context encoder respectively, where the max_tokens_{group} is the maximum sequence length of the tokenized sentence within each *group* in the current batch.

After padding and passing through the sentence encoders, the batch of targets is encoded as T, and the batch of contexts is encoded as C, where $T, C \in \mathbb{R}^{B \times D}$, with D being the embedding dimension. These matrices are then passed through a linear kernel, e.g. multiplied together into logits $L = TC^\intercal$, $L \in \mathbb{R}^{B \times B}$, where each element $L_{i,j}$ is the similarity of target embedding T_i to context embedding C_j.

The logits L are then passed together with the correct class labels $\mathbf{k}$ (correct contexts) into a combined Softmax and Cross-Entropy layer, which computes

a single loss value for the entire batch of B targets and contexts, facilitating the training of the ICT objective from eq. (3.1) using Sampled Softmax from eq. (3.3).

We explore two possibilities for the Sentence Encoder training architecture, one inspired by Sentence-BERT (see section 2.6.1), and one inspired Chang et al. ([CYC+20], see section 2.6.2), which we call One Tower and Two Tower architectures, respectively.

One Tower. This approach ties the weights of the target and context Sentence Encoders, or in other words, uses the same Sentence Encoder to encode both the targets and the contexts. This results in training a single Sentence Encoder jointly embedding both target log-lines and their contexts into a vector space. This is the same approach as used by Sentence-BERT for encoding pairs of sentences into the same vector space. The advantage of this approach is faster training due to only needing to train a single Sentence Encoder. Furthermore, given our assumption that log-lines are related to their surrounding contexts, embedding both the context and target using the same encoder could result in it learning a good vector space representation of the log-lines.

Two Tower. This approach directly corresponds to fig. 3.2, where the Sentence Encoders for the targets and contexts are separate entities. This approach is commonly used when training ICT together with tasks such as BFS and WLP (see section 2.6.2). Using separate Sentence Encoders presents greater flexibility during training, as the contexts and targets can have different structures, and having Sentence Encoders tailored to each specific case can present better embeddings for each. The resulting context and target embeddings are still being jointly embedded into the same vector space due to the loss function combining the outputs of both sentence encoders.

After training on the ICT task, the desired log-line Sentence Encoder is the target Sentence Encoder (fig. 3.2) from the ICT network.

3.5 Evaluation on Anomaly Detection task

To evaluate how well our trained Sentence Encoders embed log-lines for the downstream Anomaly Detection task, we will train an AETCN ([Kor21], see section 2.7) anomaly detector with fixed training hyperparameters, so that the only observed difference should come from the embeddings produced by different Sentence Encoders.

As AETCN is a sequence anomaly detection method, it expects the dataset to be in the form of groups of log-lines, with the whole group being labeled as either Normal or Anomalous. Given a labeled anomaly detection logs dataset, we will divide it into training, validation, and testing splits. These three splits will then be embedded from text log-lines into vectors by our trained Sentence Encoder. It must be noted that although AETCN works with groups of log-lines, no aggregation of log-lines is being done, the group of strings gets transformed into a group of embeddings, line by line.

After embedding, the AETCN detector will be trained only on the embedded groups from the training split which are labeled Normal. Then to detect anomalies using AETCN we need to find a threshold on the reconstruction error. It is selected to such a value which maximizes the F1-score (eq. (2.5)) on the validation split, as described in section 2.7.

Precision, recall, and F1-score are then computed on the testing split using the threshold selected on the validation split. We will use these scores to compare the embedding quality of different Sentence Encoders on the anomaly detection task.

3.6 Overall solution design

Figure 3.3 on page 28 shows the overall structure of the whole solution. The left part illustrates the bulk of our work and the topic of this thesis, the unsupervised Sentence Encoder pretraining, while the right part shows how the Sentence Encoder gets evaluated using the Semi-Supervised AETCN Anomaly Detection method by Koryťák [Kor21].

The *ICT pretraining* operation, on the left side of fig. 3.3, in addition to the shown inputs of the BERT checkpoint and the training data, also encompasses the remaining training configuration for the Sentence Encoder, such as pooling strategy and which ICT architecture was used. They are not shown to prevent unnecessarily cluttering the figure.

The different coloring of some operations, such as *AETCN Training*, on the right side of fig. 3.3, which shows the Anomaly Detection flow, signifies which parts of the solution we cooperated on with Koryťák. The *Split* of the anomaly detection datasets was cooperated such that we could easily compare our Anomaly Detection scores, because we used the same HDFS1 anomaly detection dataset (as described in section 4.2).

The *AETCN Training* and *Find threshold* operations were provided by Koryťák when providing the whole AETCN architecture, as described in section 3.5. The *Anomaly Detection Task Evaluation* producing the performance metrics was also coordinated upon for easier model comparisons.

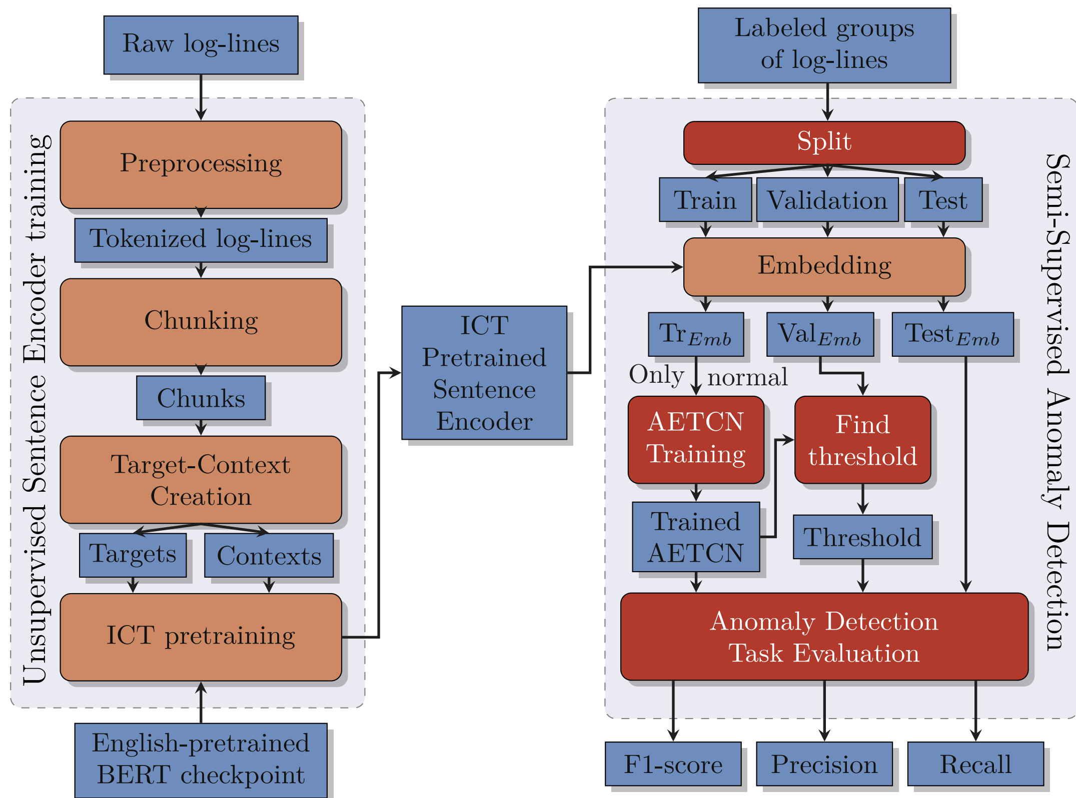


Figure 3.3: Overall solution design showing Sentence Encoder training and subsequent Anomaly Detection evaluation. Blue nodes show objects, orange and red nodes show operations. Orange nodes are our work, while red nodes were done in cooperation with Koryťák [Kor21]

Chapter 4

Datasets

4.1 Raw Datasets

	HDFS1	HDFS2	Spark	BGL	Hadoop	Zookeeper
Lines	11 175 629	71 115 180	33 236 604	4 747 963	393 431	74 380
Labels	Per group[1]	-	-	Per line	Per group[2]	-
Raw size	1.47GB	16.06GB	2.75GB	708MB	48.61MB	9.95MB

[1] has anomalies labeled by block ids (per session)
[2] has anomalies labeled by whole tasks

Table 4.1: Summary of raw used datasets

All of our datasets come from Loghub [HZHL20], which is a collection of log datasets freely available for automatic log analysis research. We have needed datasets for different purposes, some for ICT training data creation and some for anomaly detection. This section briefly describes all of them. The overview of all used datasets can be seen in table 4.1.

4.1.1 HDFS1

```
081109 203519 143 INFO dfs.DataNode$DataXceiver: Receiving block blk_
    −1608999687919862906 src: /10.250.10.6:40524 dest: /10.250.10.6:50010
081109 203519 145 INFO dfs.DataNode$PacketResponder: PacketResponder 1 for block blk_
    −1608999687919862906 terminating
...
081109 210201 32 WARN dfs.FSNamesystem: BLOCK∗ NameSystem.addStoredBlock:
    Redundant addStoredBlock request received for blk_−2995535508265484441 on
    10.251.194.129:50010 size 67108864
```

Listing 4.1: HDFS1 example log-lines

HDFS is the *Hadoop Distributed File System* designed to run on commodity hardware to provide high availability and maintainability. This dataset

published by Xu et al. [XHF+09] was generated on a private cloud using benchmark workloads. 3 sample log-lines can be seen in listing 4.1.

Each log in the dataset contains 'blk_NUMERIC_ID', which is a block ID using which log-lines can be grouped. This dataset is labeled, with the anomaly labels being only available per block ID. The anomalies were labeled manually using handcrafted rules.

The authors preprocessed this dataset such that each line of the raw text file corresponds to a single log message, each having a timestamp and basic message type at the beginning.

4.1.2 HDFS2

```
2016−10−22 13:28:13,176 INFO BlockStateChange: BLOCK∗ addStoredBlock: blockMap
       updated: 10.10.34.30:50010 is added to blk_1075559733_1818909 size 160
2016−10−22 13:28:13,176 INFO org.apache.hadoop.hdfs.StateChange: DIR∗ completeFile: /
       pjhe/test/42/_temporary/0/_temporary/
       attempt_201610221328_0157_m_000155_26359/part−00155 is closed by
       DFSClient_NON
MAPREDUCE_248870818_112
...
2015−08−21 11:16:17,538 INFO org.apache.hadoop.hdfs.server.namenode.SecondaryNameNode
       : registered UNIX signal handlers for [TERM, HUP, INT]
2015−08−21 11:16:18,094 INFO org.apache.hadoop.metrics2.impl.MetricsConfig: loaded
       properties from hadoop−metrics2.properties
...
2015−08−21 11:16:44,610 WARN org.apache.hadoop.hdfs.server.datanode.DataNode:
       IOException in offerService
java.io.EOFException: End of File Exception between local host is: "mesos−master
       −2/10.10.34.12"; destination host is: "mesos−master−1":9000; : java.io.EOFException;
       For more details see:  http://wiki.
apache.org/hadoop/EOFException
        at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)
```

Listing 4.2: HDFS2 example log-lines

Second dataset from a Hadoop cluster, this one was collected by the authors of Loghub [HZHL20] at the CUHK (Chinese University of Hong Kong) compute cluster. Example raw lines can be seen in listing 4.2.

This dataset is unlabeled and available as raw log outputs captured from 32 DataNodes, one NameNode and one Secondary NameNode. Due to its raw nature, a single log message can span multiple lines in this dataset, and the log files do contain full exception stack traces and other multi-line messages in addition to structured log messages containing timestamps and other structured information.

The contents of log messages slightly differ by the node type they come from, as there are 3 types of nodes in an HDFS cluster, a NameNode, serving as the main coordinator node, a Secondary NameNode, which is offloading

some of the work from the NameNode, and a lot of DataNodes, which store the actual data.

4.1.3 Spark

```
17/06/09 20:10:40 INFO executor.CoarseGrainedExecutorBackend: Registered signal handlers
      for [TERM, HUP, INT]
17/06/09 20:10:40 INFO spark.SecurityManager: Changing view acls to: yarn,curi
17/06/09 20:10:41 INFO slf4j.Slf4jLogger: Slf4jLogger started
17/06/09 20:10:41 INFO Remoting: Starting remoting
17/06/09 20:10:41 INFO Remoting: Remoting started; listening on addresses :[akka.tcp://
      sparkExecutorActorSystem@mesos-slave-07:55904]
...
16/04/07 10:46:31 WARN executor.CoarseGrainedExecutorBackend: An unknown (mesos-
      slave-24:43670) driver disconnected.
...
15/09/01 18:19:50 ERROR executor.CoarseGrainedExecutorBackend: Driver 10.10.18.32:65488
       disassociated! Shutting down.
```

Listing 4.3: Spark example log-lines

Unlabeled dataset collected by authors of Loghub [HZHL20] at the CUHK Apache Spark cluster, which is an analytics platform for big data processing. Example logs can be seen in listing 4.3.

The logs have a clear timestamp and log level at the beginning of each message, and due to the varied capabilities Apache Spark provides there is a lot of variety in the following log texts.

4.1.4 BGL

```
- 1118271740 2005.06.08 R03-M1-N9-C:J09-U11 2005-06-08-16.02.20.600478 R03-M1-
      N9-C:J09-U11 RAS KERNEL INFO 1 ddr errors(s) detected and corrected on rank 0,
      symbol 25, bit 1
- 1118285722 2005.06.08 R15-M1-N6-C:J04-U11 2005-06-08-19.55.22.798062 R15-M1-
      N6-C:J04-U11 RAS KERNEL INFO CE sym 14, at 0x06047860, mask 0x20
...
APPREAD 1117869872 2005.06.04 R04-M1-N4-I:J18-U11 2005-06-04-00.24.32.432192
      R04-M1-N4-I:J18-U11 RAS APP FATAL ciod: failed to read message prefix on
      control stream (CioStream socket to 172.16.96.116:33569
...
KERNDTLB 1118536327 2005.06.11 R30-M0-N9-C:J16-U01 2005-06-11-17.32.07.581048
       R30-M0-N9-C:J16-U01 RAS KERNEL FATAL data TLB error interrupt
```

Listing 4.4: BGL example log-lines

Labeled dataset comprising of data on the operation of the BGL supercomputer, a system with over 131 thousand processors and 32TB of memory. The dataset was collected by Oliner and Stearley [OS07]. Each log line begins with an alert category, with normal (non-anomalous) messages starting with the '-' symbol, as can be seen in listing 4.4. It contains lower-level (kernel, networking) messages compared to other used datasets. Due to its simple

anomaly labeling strategy, it is not well suited for benchmarking anomaly detection methods, as it can be easily learned, but it can be a good source of unlabeled data.

4.1.5 Hadoop

```
2015−10−18 18:01:47,978 INFO [main] org.apache.hadoop.mapreduce.v2.app.MRAppMaster:
      Created MRAppMaster for application appattempt_1445144423722_0020_000001
...
2015−10−18 18:01:53,744 INFO [AsyncDispatcher event handler] org.apache.hadoop.
      mapreduce.v2.app.job.impl.JobImpl: job_1445144423722_0020Job Transitioned from
      INITED to SETUP
...
2015−10−18 18:04:11,034 ERROR [RMCommunicator Allocator] org.apache.hadoop.
      mapreduce.v2.app.rm.RMContainerAllocator: Container complete event for unknown
      container id container_1445144423722_0020_01_000012
...
2015−10−18 18:05:27,570 WARN [LeaseRenewer:msrabi@msra−sa−41:9000] org.apache.
      hadoop.ipc.Client: Address change detected. Old: msra−sa−41/10.190.173.170:9000 New:
       msra−sa−41:9000
```

Listing 4.5: Hadoop example log-lines

Hadoop is a distributed big data processing platform capable of working with very large datasets due to its MapReduce programming paradigm.

This labeled dataset was artificially created by Lin et al. [LZL+16] for research on log clustering. It comes from a small cluster of 5 machines running two benchmark MapReduce applications. Each log message contains a timestamp and log level. Example log lines can be seen in listing 4.5.

The anomaly labels are provided per groups of log messages coming from the same MapReduce subtask. The anomalies were artificially caused and are of three types, *Machine failure*, *Network failure* and *Disk full*.

4.1.6 Zookeeper

```
2015−07−29 19:37:19,676 − WARN  [RecvWorker:188978561024:
      QuorumCnxManager$RecvWorker@765] − Interrupting SendWorker
...
2015−07−29 21:01:41,504 − INFO  [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:2181:
      ZooKeeperServer@839] − Client attempting to establish new session at
      /10.10.34.19:33425
2015−07−29 21:34:45,452 − INFO  [CommitProcessor:1:ZooKeeperServer@595] − Established
      session 0x14ed93111f20027 with negotiated timeout 10000 for client /10.10.34.13:37177
...
2015−07−29 23:44:28,903 − ERROR [CommitProcessor:1:NIOServerCnxn@180] −
      Unexpected Exception:
```

Listing 4.6: Zookeeper example log-lines

Zookeeper is a service providing centralized configuration for a large amount of distributed services within the Hadoop ecosystem. This dataset was collected at CUHK by the authors of Loghub. [HZHL20]

This dataset is unlabeled and preprocessed in a way that each log-message only spans one line. Each log message contains a timestamp and a log level, example messages can be seen in listing 4.6.

4.2 Anomaly Detection Dataset

There are few high-quality labeled datasets for anomaly detection, which could be used as benchmark datasets. From the datasets we presented, only three are labeled, from which we have chosen only the HDFS1 dataset (see section 4.1.1).

We have decided to drop BGL (section 4.1.4) as an anomaly detection benchmark due to findings by Souček [Sou20], where it was found that due to the automatic nature of anomaly label assignment, it is typically easy to learn to detect anomalies based on the appearance of symbols such as "Warning" or "Error". As we are interested in detecting more complex anomalies that require understanding the semantics of the messages, BGL as a dataset mainly used with methods that detect anomalies per log-line and do not employ NLP is not interesting to us for anomaly detection.

We have also decided not to use Hadoop (section 4.1.5) dataset for anomaly detection, even though it is not labeled per line, its simplistic automatic labeling process makes it less appealing to us.

As described in section 4.1.1, the anomaly labels in HDFS1 are provided per block and not per line. There are 575 061 blocks in the dataset, 16 838 of which are anomalous, which means that roughly 2.93% of all blocks are anomalous.

	HDFS1-Train	HDFS1-Val	HDFS1-Test	Full HDFS1
Lines	9 051 028	1 004 865	1 119 736	11 175 629
Blocks	465 798	51 756	57 507	575 061
Anomalous blocks	13 639	1 515	1 684	16 838

Table 4.2: Training, validation and test splits of HDFS1

We have then split the HDFS1 dataset three ways into a training, validation, and test split. The split was done such that no blocks were divided. The anomalous blocks were uniformly distributed among the splits to preserve

the original distribution, e.g., around 2.93% of blocks in each split would be anomalous. The split details are listed in table 4.2.

4.3 ICT Datasets

For the creation of ICT pretraining data for training sentence encoders we do not need labels, as the raw lines from the datasets get transformed into ($target$, $context$) pairs, as described in section 3.3. We have created 4 ICT datasets, 2 based on HDFS1 and 2 created as a mixture of all the datasets described in section 4.1.

4.3.1 HDFS1 based datasets

A common practice when creating embeddings, if the method needs training, is to use a training split of the anomaly detection data for learning embeddings. Therefore we created our first two datasets from the HDFS1 training split as described in section 4.2.

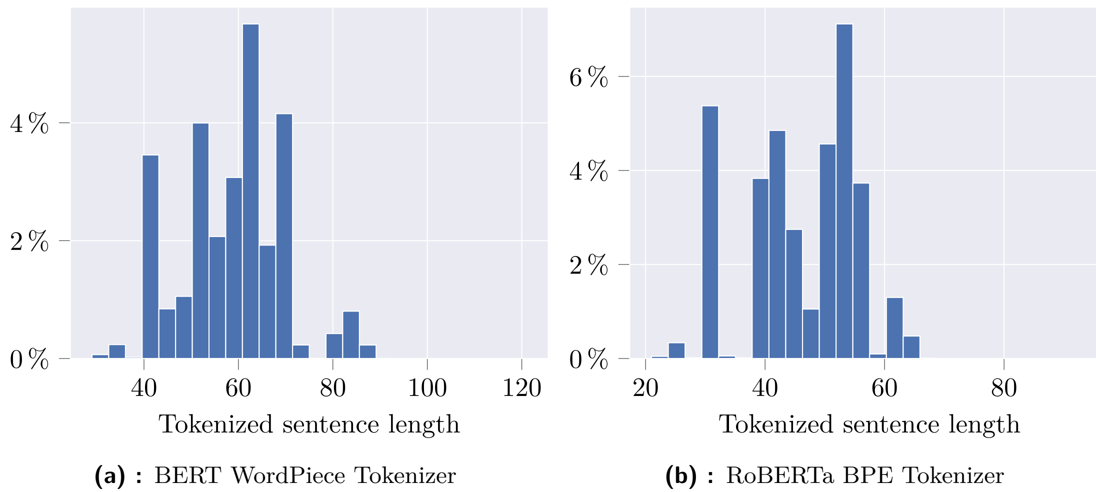
Given the general three steps of ICT data creating, being chunking, target-context pair creation, and then the context flattening-truncation step, these datasets differ in how the chunks were created.

Both datasets were created to use chunks of 10 lines each. This was chosen by examining the tokenized (see section 2.1) lengths of the log-lines after using both kinds of tokenizers we used. BERT tokenizer resulted in an average sentence length of 58 tokens, with a maximum length 129, while using RoBERTa BPE tokenizer resulted in an average of 46, with a maximum 94. Detailed histograms can be seen in fig. 4.1.

Using chunks of 10 lines will result in contexts usually having only slightly more tokens than the maximum of 512 the base encoder models can accept. Due to not usually having to truncate many tokens, and with relatively low maximum tokenized lengths in the HDFS1 dataset, we decided to only use the Concatenation to max token count context truncation strategy (see section 3.3.3) for both derived datasets.

HDFS1-time

This dataset orders all log-messages by timestamp in ascending order to simulate how log messages are usually collected. The data is subsequently



(a) : BERT WordPiece Tokenizer

(b) : RoBERTa BPE Tokenizer

Figure 4.1: Histograms of tokenized HDFS1 train sentence lengths

chunked and preprocessed for ICT as described in section 3.3.1) and following sections.

HDFS1-blocks

As previously described in section 4.1.1, the HDFS1 log-lines contain block IDs, which denote log lines relating to the same filesystem block. Logs about the same block could be more relevant to each other, and it could be more meaningful to create target sentences and contexts for ICT from chunks containing log lines about the same block.

To explore this, we created this dataset by first grouping the log lines by their block IDs and keeping them ordered by time within each group. These log-line groups were then chunked, such that each chunk only contained log lines with the same block ID. The remaining preprocessing steps are the same as in section 3.3.

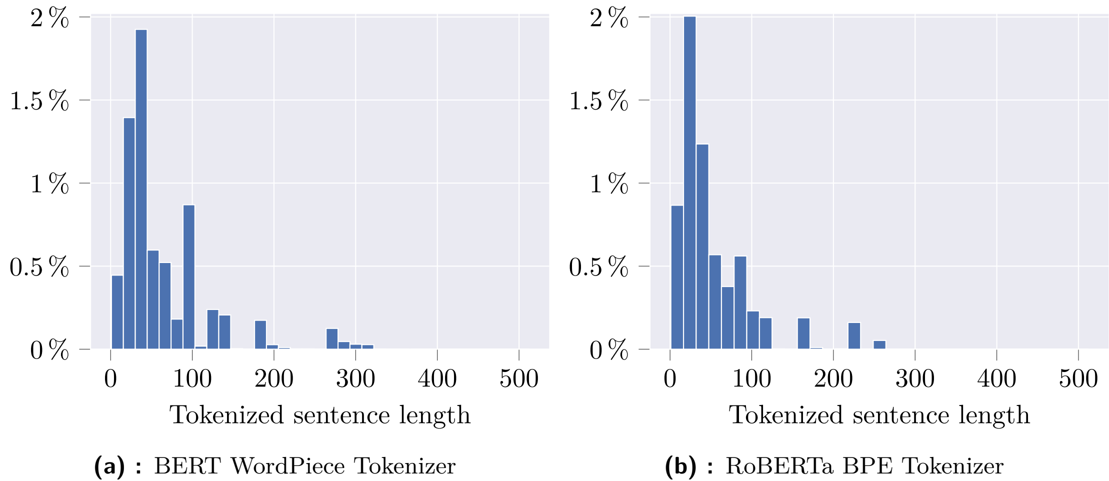
4.3.2 Mix datasets

Our other pair of ICT datasets was created to test whether a sentence encoder can learn semantic embeddings by training on multiple different sources of log data, not just on the log data on which anomaly detection is then performed.

Another factor contributing to our decision to create a mixed dataset were the surprising anomaly detection experiment results, where sentence encoders

	HDFS1-Train	HDFS2	Spark	BGL	Hadoop	Zookeeper
Lines	570 000	2 280 000	2 280 000	570 000	373 690	70 600
Chunks	57 000	228 000	228 000	57 000	37 369	7 060
Proportion	9,28%	37,11%	37,11%	9,28%	6,08%	1,15%

Table 4.3: Composition of the Mix dataset



(a) : BERT WordPiece Tokenizer

(b) : RoBERTa BPE Tokenizer

Figure 4.2: Histograms of tokenized Mix dataset sentence lengths

trained on HDFS1-time performed better than when trained on HDFS1-blocks dataset (see section 6.2.2).

The dataset was created from all of the datasets from section 4.1, the specific composition can be seen in table 4.3. For HDFS1, only the train split as described in section 4.2 was used. We wanted to have data from HDFS account for roughly 50% of the dataset, but with having a much larger representation of the more varied HDFS2 data compared to HDFS1. The other remaining datasets also come from other various distributed services, except for BGL, which comes from a supercomputer and was included for more variety.

The lines from the datasets could not be sampled outright because of the need to create the (*target*, *context*) pairs for ICT. But we do not want to just take the beginning of each log file. To solve this, we decided to chunk all datasets first. The datasets had all of their lines ordered by time, and then each dataset got partitioned into chunks of 10 lines, the same as for HDFS1 datasets.

The chunks themselves could then be randomly sampled, which resulted in us having valid chunks for target and context creation, while also having log lines from any part of the datasets, not just the beginning.

We analyzed the tokenized lengths in this dataset to verify that the chunk size of 10 was appropriate for this dataset. The histograms can be seen in fig. 4.2. We found out that the average tokenized lengths were not much higher than for HDFS1, with the average token length being 64 tokens when using BERT tokenizer, and 53 when using RoBERTa tokenizer. But there is a much higher variance of the tokenized lengths, with the maximum lengths being over 512 tokens, which is the maximum length the base models can accept and therefore were truncated to 512 tokens straight away.

For this reason we have decided to explore both context truncation methods from section 3.3.3, as this truncation could have a higher impact on this dataset compared to HDFS1.

Mix Concat. This is the version of the dataset using Concatenation to maximum token count context truncation strategy

Mix SmartAvg. This version uses the Smart Average Threshold (see algorithm 1) technique to find the fair truncation length for each log-line in the context.

Chapter 5

Implementation

The language *Python*[1] 3.7 [VRD09] was used for implementation of our approaches described in chapter 3, along with most of the support scripts, as it is currently the most widely used language for scientific computing and Artificial Intelligence.

We have used the widely used machine learning library *PyTorch* [PGM+19] for implementing our neural network architectures. In conjunction we have used the *Huggingface Transformers* [WDS+20] library for the base implementations and source of pretrained model checkpoints of the various BERT models we used within our Sentence encoders, as described in section 3.1.

For dataset preprocessing and manipulation, we have created our scripts in *Python*, while also using the *Huggingface Datasets* [WLvP+20] library, which provides a simple interface for manipulating very large datasets.

For anomaly detection, we were provided an AETCN anomaly detector implementation by Koryťák [Kor21], which we've used for our experiments. It is also implemented in *Python* using the *PyTorch* library. *fastText*[2] [BGJM17] was used to obtain baseline sentence embeddings for anomaly detection experiments.

The overall implementation follows the overall solution design described in section 3.6, illustrated in fig. 3.3.

The access to the computational infrastructure of the OP VVV funded project CZ.02.1.01/0.0/0.0/16_019/0000765 "Research Center for Informatics" is also gratefully acknowledged, which was used for running our experiments.

[1] `https://www.python.org`
[2] `https://fasttext.cc/`

5.1 Sentence Encoders and ICT networks

Implementation of Sentence Encoders was straightforward and follows the architecture described in section 3.1 without significant changes. We create a Sentence Encoder class for each pair of base BERT-model type and pooling strategy, list of which can be seen in section 6.1.

The One Tower and Two Tower ICT training network implementations also closely follow their descriptions in section 3.4. Our initial implementation for training on the ICT task presumed the dataset was in the form of chunks, and the creation of targets and contexts from chunks was done on the fly during training.

Although this is convenient, our experiments have shown that generating targets and contexts on the fly significantly slows down the training process. Therefore we have decided to redo our training implementation so that it accepts datasets already containing the $(target, context)$ pairs. Although this means preparing the ICT data in advance, the training speed was increased threefold.

5.2 ICT data pipeline

The implementation follows the steps described in section 3.3. Initially, we have only needed to tokenize and chunk (and in the case of HDFS1-blocks also group) the raw input log-lines, with the target and context creation being done online at training time.

During this stage we have discovered a bug in the *Huggingface Datasets* library, which has significantly slowed down the chunking process. This has led us to reimplement the chunking without the use of the library, such that it was done in-memory. This has resulted in a 72-fold speedup in chunking. We have reported the issue on *Huggingface Datasets* GitHub, and a fix for our issue was merged, although only after we have already completed our experiments.

After we decided to prepare the whole ICT dataset ahead of time, we have also implemented the $(target, context)$ creation and context flattening steps, which did not present any issues. Preparing the ICT dataset ahead of time also meant selecting training epochs ahead of time. This is implemented by selecting as many different targets from each chunk as there are epochs. This also means that we cannot use more epochs than there are lines in a chunk if we do not want to have duplicate training data.

5.3 Anomaly Detection pipeline

Implementation of anomaly detection experiments was straightforward. After a Sentence Encoder has been trained, we have created a data pipeline which uses the Sentence Encoder to embed the training, validation, and testing datasets needed for anomaly detection ahead of time. Then we created an experiment runner script which used the three embedded datasets to train the provided AETCN [Kor21] detector on the train split, find the anomaly detection threshold on the validation split and then compute the precision, recall, and F1-score metrics on the test split. See the right side of fig. 3.3 for illustration of this process.

Chapter 6

Experiments

In this chapter we present the findings of our experiments. The chapter is structured as follows. Section 6.1 describes the various configuration details, and an overview of what kind of experiments were run. Section 6.2 presents the main results of our experiments. Section 6.3 tests various hypotheses on the performance of our Sentence Encoders using statistical hypothesis testing methods.

6.1 Setup

We have trained a total of 72 different Sentence Encoders, with embedding dimension $D = 100$. The experiments were run on the RCI computing cluster employing the NVIDIA Tesla V100 GPUs with 32GB of VRAM. ICT training time for each Sentence Encoder ranged from 14 to 22 hours, with the follow-up Anomaly Detection experiments for each encoder taking further 4 hours, including embedding time.

Sentence Encoder configurations

The encoders are of several basic types. Three building blocks describe each Sentence Encoder - the base BERT model, the pooling strategy, and the number of towers used for ICT pretraining.

We have used two base BERT models, both of which were smaller, distilled versions of common large BERT models. Distillation, introduced by Hinton, Vinyals and Dean [HVD15], is a process of training a smaller, so-called student, neural network to behave the same as a larger teacher neural network.

Distilling a network enables it to retain most of the accuracy of the original network while being much smaller and executing much faster.

We have opted to use distilled BERTs to be able to explore more Sentence Encoder architectures more quickly, as the computing requirements for pretraining full BERTs are much higher.

The checkpoints we used were pretrained on English language corpora, as our log datasets contain English text in them.

- **DistilBERT** - Checkpoint name *distilbert-base-cased*. Presented by Sanh, Debut, Chaumond and Wolf [SDCW19] this base model is a distilled version of $\text{BERT}_{\text{BASE}}$ model from the original BERT paper. Its size is only 60% compared to the original, while retaining over 97% of its performance on common benchmarks, while being over 60% faster. It uses $L = 6$ Transformer Encoder blocks, compared to the original model's 12. The model uses the same WordPiece (see section 2.1) tokenizer as the original BERT. This model uses a case-sensitive tokenizer with a vocabulary of 28996 tokens.
- **DistilRoBERTa** - Checkpoint name *distilroberta-base*. This is a distilled version of the RoBERTa model, distilled by the authors of the *Huggingface Transformers* library. It is the same size as the DistilBERT model. The original RoBERTa was created by Liu et al. [LOG+19]. They set out to replicate the original BERT paper [DCLT18] to find the impacts different hyperparameter choices have on the final model. Their findings resulted in a different training strategy for the network, using only the MLM task (see section 2.5.2) and training the network for longer. This resulted in creating RoBERTa, which surpassed BERT in the same benchmarks. The model uses a case-sensitive BPE tokenizer with a vocabulary of 50265 tokens.

We have used the three pooling strategies as described in section 3.1. We will use the following names for the strategies in the remainder of this chapter.

- **Cls** - This pooling strategy uses the output vector of the last transformer block corresponding to the [CLS] token.
- **Mean** - This pooling strategy averages all the output vectors of the last transformer block corresponding to non-special tokens, e.g., all tokens except [CLS] and [SEP]
- **LastNCls** - This pooling strategy is similar to the Cls strategy, but it concatenates the hidden vectors of the last N transformers blocks corresponding to the [CLS] token. In our experiments we've used N = 3.

The pooling layer outputs vectors which usually have the same dimensionality as the hidden size d_h of the BERT model, except for LastNCls, which outputs vector with $\mathrm{N} \cdot d_h$ dimensions. The vectors are then passed through a linear layer which transforms them from the pooling layer output dimension to the desired final embedding dimension.

The Sentence Encoder can also be trained with either the **One Tower** or **Two Tower** ICT architecture, which may be abbreviated to **1T** and **2T** respectively in the remainder of this chapter.

This results in $2 \times 3 \times 2 = 12$ possible Sentence Encoder types for each training dataset configuration.

Dataset configuration

Each Sentence Encoder configuration was tested on six dataset configurations, which can be seen in table 6.1.

Dataset	Epochs	Training pairs[1]
HDFS1-blocks	4	3 620 384
HDFS1-time	4	3 620 384
Mix Concat	3	1 843 264
Mix Concat	5	3 072 128
Mix SmartAvg	3	1 843 264
Mix SmartAvg	5	3 072 128

[1] Total number of (*target*, *context*) training pairs in a given configuration

Table 6.1: Dataset configurations for experiments

It must be noted that although timestamps in the log files were used for ordering during chunk creation, all timestamps were removed from the datasets prior to ICT pretraining and Anomaly Detection experiments. This was done because handling time-based features for log-lines is typically handled by different automatic methods, using log-message templates, etc. This is done so that only the remaining text information is passed into NLP methods. For anomaly detection, after creating NLP embeddings, it is common to concatenate these vectors with the extracted time-based features prior to executing the anomaly detection task. This is the approach explored by Souček [Sou20].

ICT training configuration

We have used batches containing 32 $(target, context)$ pairs, which means that each $target$ will have 31 negative examples when employing the Sampled Softmax from eq. (3.3). We have wanted to use larger batch sizes, but a batch size of 64 did not fit into memory, so we used 32 due to the general recommendation to use powers of 2 batch sizes, which usually work best for current GPU architectures.

We did not use gradient accumulation, which is a common approach to simulate larger batch sizes in memory-constrained environments. Although our task when using Sampled Softmax is reminiscent of performing a classification task with $batch_size$ classes, because the classes are the $context$s which differ for every batch, implementing gradient accumulation is not straightforward.

Anomaly Detection configuration

For the anomaly detection experiments using AETCN, we have used hyperparameters found by Koryťák [Kor21]. They have run neural network architecture grid search to find the AETCN hyperparameters maximizing the F1-score while using fastText embeddings of the same dimension $D = 100$ as our sentence encoders. Although the architecture was not tuned for each specific embedding, we do not expect it to be of large importance due to the embeddings having the same dimensionality.

The used AETCN architecture consists of a TCN [LFV+16] encoder with 142 filters, which is followed by two linear layers, creating the autoencoder bottleneck, with the inner dimension between them being 1246. Finally, the network ends with a TCN decoder with 100 filters.

As our HDFS1 anomaly detection dataset (section 4.2) is labeled per block (see section 4.1.1), our anomaly detection experiments were also designed to predict anomalies per block, instead of per line.

The data flow (roughly illustrated in fig. 3.3) for anomaly detection is as follows. Given a block of n log-lines from the dataset $(l_1, l_2, ..., l_n)$, each log-line is then embedded using a pretrained Sentence Encoder into a $D = 100$-dimensional vector individually, and the block is then a collection of n D-dimensional vector $(e_1, e_2, ..., e_n)$, where each $e_i \in \mathbb{R}^D$. The whole embedded block is then used as a matrix $B \in \mathbb{R}^{n \times D}$, where each row i of the matrix B is the embedding e_i of log-line l_i. The matrix B is then fed as an input to the anomaly detector, which the predicts either a *Normal* or *Anomalous* label.

6.2 Results

Here we present summarised results of all of our 73 experiments (including the baseline). Each results table will contain an overall rank column which signifies the concrete method's ranking among all of our experiments. The full table with all experimental results can be found in appendix B.

6.2.1 Baseline

Our baseline was running AETCN on the HDFS1 anomaly detection dataset (section 4.2) using log-line sentence embeddings produced by fastText [BGJM17] with dimension $D = 100$, learned from the HDFS1-train split. The baseline scores can be seen in table 6.2

Overall rank	Model	Precision	Recall	F1-score
1	fastText	0.7768	0.9365	0.8492

Table 6.2: Baseline anomaly detection scores using fastText

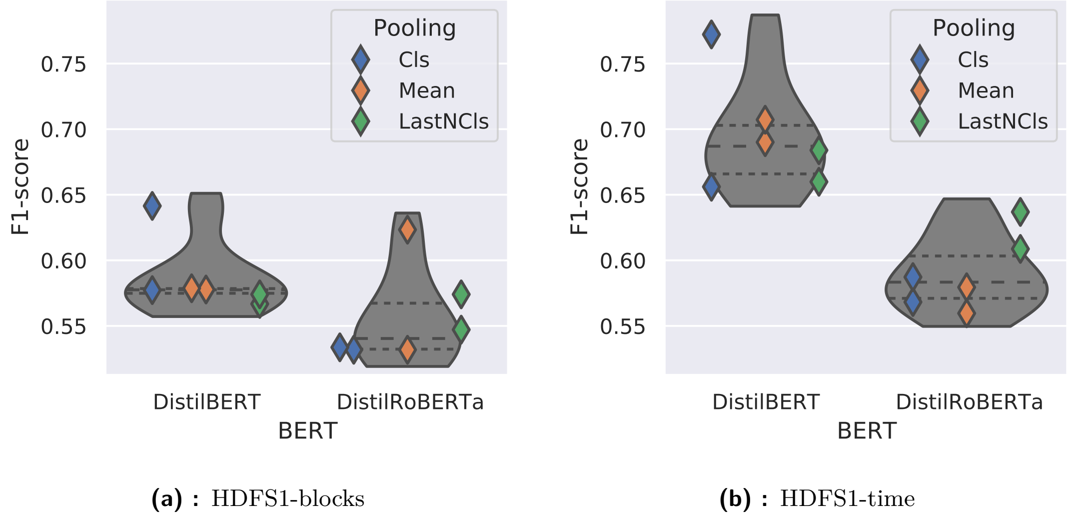
6.2.2 HDFS1-blocks and HDFS1-time

Our first experiments were trained on the HDFS1-based datasets for four epochs on both. The overall results can be seen in fig. 6.1, with overall violinplots in the background for BERT types, showing the quartiles, while also showing each experiment result, colored by its pooling type. The results for the best three models can be seen in table 6.3a for HDFS1-blocks, and in table 6.3b for HDFS1-time.

The results were surprising to us, with the encoders trained on HDFS1-blocks performing worse than those trained on HDFS1-time. We have expected to find models trained on HDFS1-blocks to perform better, as the chunking procedure used when creating HDFS1-blocks creates chunks of lines related to the same block and should theoretically be closer in usage to the way the anomaly detection task is handled - per block.

We hypothesize that this low performance when training on HDFS1-blocks could be caused by the Sentence Encoder paying too much attention to the block ID. We explore this in section 7.1.

On the other hand, the good results for HDFS1-time trained encoders encouraged us that our choice to pretrain using ICT seems to be valid,



(a) : HDFS1-blocks

(b) : HDFS1-time

Figure 6.1: F1-scores when trained on HDFS1 datasets

Rank	BERT	Pooling	Towers	Precision	Recall	F1	% of baseline
40	DistilBERT	Cls	2T	0.7916	0.5392	**0.6415**	75.54%
46	DistilRoBERTa	Mean	1T	0.5811	**0.6722**	0.6233	73.40%
59	DistilBERT	Mean	1T	**0.9747**	0.4115	0.5787	68.15%

(a) : HDFS1-blocks

Rank	BERT	Pooling	Towers	Precision	Recall	F1	% of baseline
4	DistilBERT	Cls	2T	0.7418	**0.8052**	**0.7722**	90.93%
18	DistilBERT	Mean	2T	**0.8449**	0.6081	0.7072	83.27%
24	DistilBERT	Mean	1T	0.7799	0.6188	0.6901	81.26%

(b) : HDFS1-time

Table 6.3: Best Sentence Encoders for HDFS1 datasets

with the best result achieving almost 91% of the accuracy of the baseline embeddings. These results also encouraged us to create the Mix dataset (section 4.3.2), to explore whether there are some innate log semantics that could be learned by learning from multiple datasets.

Examining the results in fig. 6.1 we see that the DistilBERT base models typically result in better embeddings, particularly for the HDFS1-time dataset, where the difference is quite pronounced. DistilRoBERTa models perform worse, but they still show some improvement when using HDFS1-time.

For the pooling strategies, it seems that for DistilBERT the best embeddings are usually created by using Cls pooling, followed by Mean and the LastNCls. The situation is different for DistilRoBERTa, where the Cls strategy is never

the best.

We pose given that RoBERTa was not trained on the NSP task (see section 2.5.2), which employs the [CLS] token, then the corresponding hidden vector for it does not contain enough semantic information for the rest of the sentence. It might be possible that RoBERTa produces higher quality embeddings for the individual tokens, which might be the reason why the Mean strategy performs better on HDFS1-blocks in fig. 6.1a. However, it may not be as straightforward, as the same pooling is the worst for HDFS1-time in fig. 6.1b. We explore whether Mean pooling is better than Cls pooling for DistilRoBERTa models trained on any dataset in hypothesis[1] H_0^7.

From table 6.3a and table 6.3b we can also see that the Two Tower ICT architecture produces the highest ranking encoders when training on the HDFS1 datasets.

6.2.3 Mix datasets

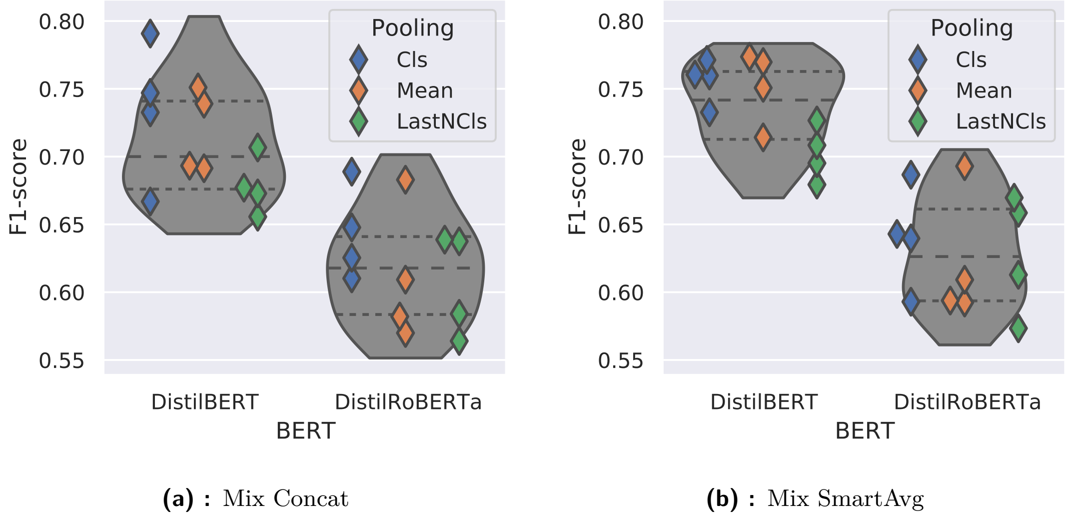
Our second set of experiments was done by training on our Mix datasets (section 4.3.2), which differ by using different context truncation strategies (see section 3.3.3). We trained all 12 of our possible Sentence Encoder types on these two datasets, each for 3 and 5 epochs, producing 48 total encoders. The overall results can be seen in fig. 6.2. The best three models can be seen in table 6.4a for Mix Concat, and in table 6.4b for Mix SmartAvg.

We see that encoders trained on these datasets typically perform better than most of those trained on HDFS1 datasets. Unfortunately, none of our models were able to match or surpass the baseline, with the Mix Concat trained DistilBERT Cls encoder reaching the highest F1-score, which is 93.12% of the baseline.

In fig. 6.2 we can compare the different truncation strategies on the datasets. Although the best encoder was trained on the Mix Concat dataset, it appears that training on Smart Average context truncated datasets results in slightly better encoders. We verify this in hypothesis H_0^3.

Generally we see the continuation of the trend that DistilBERT based encoders perform better than the DistilRoBERTa ones, although even those perform slightly better than most of the HDFS1 based encoders, with the exception of the HDFS1-time trained DistilBERT encoders.

[1] All hypotheses can be found in section 6.3



(a) : Mix Concat

(b) : Mix SmartAvg

Figure 6.2: F1-scores when trained on Mix datasets

Rank	BERT	Pooling	Towers	Epochs	Precision	Recall	F1	% of baseline
2	DistilBERT	Cls	1T	3	0.8882	**0.7126**	**0.7908**	93.12%
9	DistilBERT	Mean	1T	5	0.8822	0.6538	0.7510	88.44%
11	DistilBERT	Cls	1T	5	**0.8920**	0.6425	0.7470	87.96%

(a) : Mix Concat

Rank	BERT	Pooling	Towers	Epochs	Precision	Recall	F1	% of baseline
3	DistilBERT	Mean	1T	3	**0.9026**	0.6770	**0.7737**	91.10%
5	DistilBERT	Cls	1T	3	0.8604	**0.6989**	0.7713	90.82%
6	DistilBERT	Mean	1T	5	0.8972	0.6740	0.7698	90.64%

(b) : Mix SmartAvg

Table 6.4: Best Sentence Encoders for Mix datasets

6.2.4 Overall results

In this section we present selected results pertaining to various possible groupings of the Sentence Encoders.

In fig. 6.3 we see the comparison of F1-scores of sentence encoders trained on the Mix datasets and the HDFS1 datasets. Each experiment is plotted as a dot, with density estimation below it, with dashed horizontal lines dividing the quartiles of the distribution. This figure shows us that training on varied log datasets could lead to the encoders learning better language representations. We test the hypothesis that using Mix datasets is better in hypothesis H_0^1.

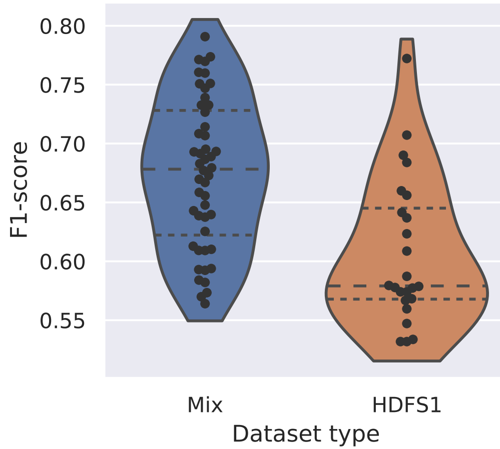


Figure 6.3: F1-score comparison of the dataset types

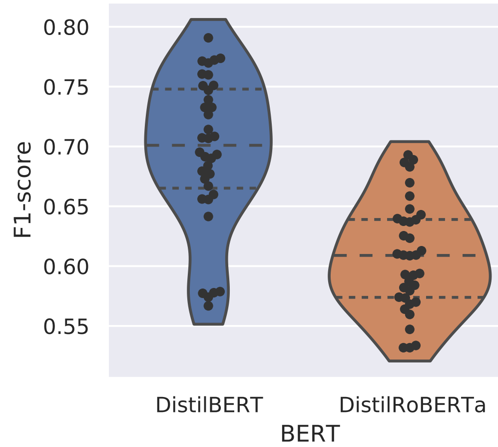
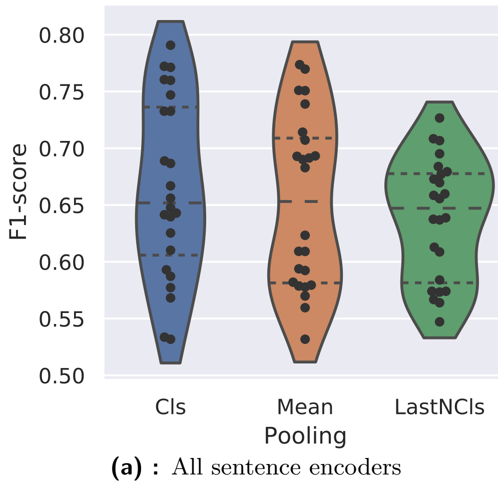
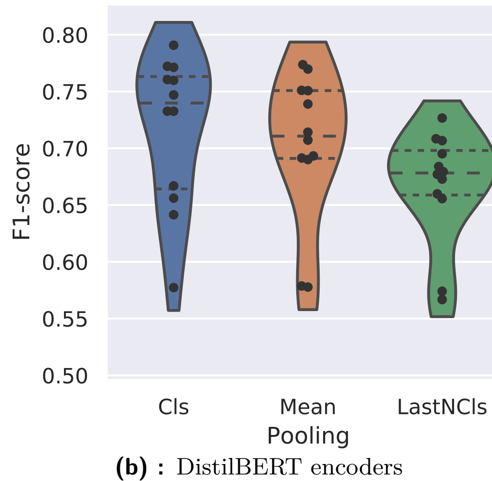


Figure 6.4: F1-score comparison of DistilBERT to DistilRoBERTa



(a) : All sentence encoders



(b) : DistilBERT encoders

Figure 6.5: F1-scores of different pooling strategies

In fig. 6.4 we see the comparison of sentence encoders using the different BERT base networks. It would appear that in general DistilBERT based encoders lead to better performance compared to DistilRoBERTa encoders. We explore this in hypothesis H_0^2.

Figure 6.5 shows the impact of using different pooling strategies. As we can see in fig. 6.5a, when taking all possible encoder configurations into account, there does not seem to be a marked difference in the F1-score distributions between the pooling strategies. We test for the presence of a difference in hypothesis H_0^4.

Figure 6.5b shows a different story. When we restrict our considerations to only the DistilBERT encoders, there seem to be differences in the performance when using different pooling strategies, with the trend appearing to be that

BERT	Pooling	Towers	Precision	Recall	F1-score
DistilBERT	Cls	2T	0.8269	**0.6418**	**0.7177**
		1T	0.8797	0.6166	0.7170
	Mean	1T	**0.8829**	0.6137	0.7170
		2T	0.8663	0.5813	0.6891
	LastNCls	1T	0.8809	0.5469	0.6697
		2T	0.8791	0.5405	0.6647
DistilRoBERTa	Cls	2T	0.8462	0.5165	0.6306
	LastNCls	1T	0.7674	0.5394	0.6172
	Mean	2T	0.8393	0.4922	0.6103
	LastNCls	2T	0.8592	0.4716	0.6004
	Cls	1T	0.8029	0.4868	0.5953
	Mean	1T	0.7313	0.5203	0.5942

Table 6.5: Averaged metrics for Sentence Encoder type comparison, ordered by F1-score

Cls is the best, followed by Mean, with LastNCls in the last place when using DistilBERT encoders. We test whether the difference between the pooling strategies is statistically significant for DistilBERT encoders in hypothesis H_0^5.

In fig. 6.6 we see the effect of One Tower compared to Two Tower ICT pretraining architecture, e.g., whether to use a single sentence encoder for embedding both targets and contexts, or to use a dedicated sentence encoder for each. We test whether there is a statistically significant difference between the training architectures in hypothesis H_0^6.

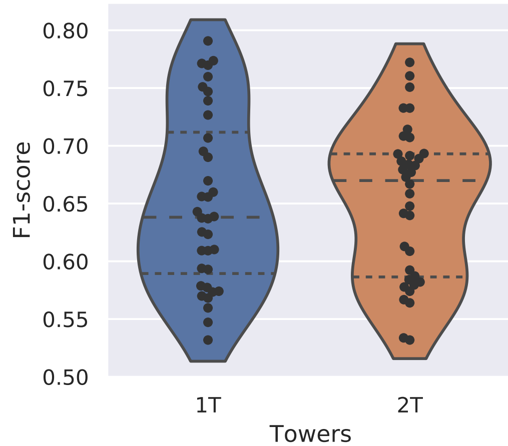


Figure 6.6: F1-scores of different ICT tower counts

Table 6.5 shows the average performance of Sentence Encoder types, where a type is a group of Sentence Encoders with the same BERT base network, pooling strategy, and ICT pretraining architecture. Each row in the table is an average of 6 sentence encoders trained with the same configuration. Encoders within a group differ by the dataset they were trained on.

From these aggregate results, we see that generally the best performing sentence encoder type is a DistilBERT with Cls pooling, trained using a Two Tower ICT architecture. It is closely followed by One Tower trained DistilBERT types, with Cls and Mean pooling.

6.3 Encoder hypotheses

In this section we present various hypotheses on the performance of using various Sentence Encoders. All hypotheses are tested at the significance level $\alpha = 0.05$.

Hypothesis $\mathbf{H_0^1}$: The median difference in F1-scores between the encoders trained on Mix datasets compared to those trained on HDFS1 datasets is negative.

Alternative $\mathbf{H_A^1}$: The median difference in F1-scores is positive.

Paraphrasing the null hypothesis H_0^1, it states that training on the Mix datasets leads to worse performing encoders compared to training on HDFS1. The distribution estimations are visualized in fig. 6.3.

We employed the non-parametric Mann-Whitney U test to test this hypothesis, which was designed to compare distributions of different populations, with possibly different sample sizes. Sample sizes differ in this case, as there are 48 encoders trained on the Mix datasets and 24 encoders trained on HDFS1 datasets.

Running the test, we obtained a p-value of $4.14 \cdot 10^{-5}$, which leads us to reject the null hypothesis H_0^1 in favor of the alternative H_A^1. We conclude that training on Mix datasets generally leads to better performing encoders.

Hypothesis $\mathbf{H_0^2}$: The median difference in F1-scores between the DistilBERT based encoders compared to the DistilRoBERTa based encoders is negative.

Alternative $\mathbf{H_A^2}$: The median difference in F1-scores is positive.

In other words, null hypothesis H_0^2 states that DistilBERT encoders perform worse than DistilRoBERTa encoders. The distribution estimates can be seen in fig. 6.4.

We employed the one-sided Wilcoxon matched Signed-rank test. We used this test because we could pair up each DistilBERT training configuration with the corresponding DistilRoBERTa configuration. There were 36 matched pairs of encoders.

Running the test resulted in a p-value of $2.47 \cdot 10^{-7}$, leading us to reject the null hypothesis H_0^2 in favor of the alternative H_A^2, meaning that DistilBERT generally leads to better performing encoders compared to DistilRoBERTa.

Hypothesis $\mathbf{H_0^3}$: The median difference in F1-scores between the encoders trained on Mix SmartAvg dataset compared to those trained on Mix Concat dataset is negative.

Alternative $\mathbf{H_A^3}$: The median difference in F1-scores is positive.

In other words, the null hypothesis H_0^3 states that training on Mix SmartAvg generally results in encoders with lower F1-scores, while the alternative says that it results in higher F1-scores compared to training the same configuration on Mix Concat.

To test this hypothesis we employed the one-sided Wilcoxon matched Signed-rank test. We decided to use the matched test, as we have trained the same encoder configurations on both kinds of datasets. We also have enough samples - 24 encoders trained on each dataset.

Running the test resulted in a p-value of 0.004, which leads us to reject the null hypothesis H_0^3 in favor of the alternative H_A^3. We conclude that using Smart Average context truncation generally results in slightly better encoders.

Hypothesis $\mathbf{H_0^4}$: The median F1-score is the same for each pooling method.

Alternative $\mathbf{H_A^4}$: The median F1-score differs between the pooling methods.

The null hypothesis H_0^4 states that all pooling methods (see section 3.1) result in encoders of similar quality, while the alternative says that some of the pooling methods results in differently performing encoders. The density estimation can be seen in fig. 6.5a

We have used the Friedman test, which is used when there are repeated measurements of the same individual across different groups. As our encoder configuration can be considered an individual, with the only difference being the pooling method, we were able to create matched triples for each possible Sentence Encoder configuration with each possible training configuration (dataset, epochs), differing only in the pooling method. There were 24 triples, e.g., 24 Sentence Encoders for each pooling type.

We have obtained p-value of 0.167, so we are unable to reject the null hypothesis H_0^4 that the F1-scores differ when using different pooling methods, e.g., we are unable to say whether any pooling method is different from the others.

Hypothesis $\mathbf{H_0^5}$: The median F1-score is the same for each pooling method when only using DistilBERT based Sentence Encoders.

Alternative $\mathbf{H_A^5}$: The median F1-score differs between the pooling methods for DistilBERT encoders.

The null hypothesis H_0^5 states that when using DistilBERT encoders all pooling methods are the same. The density estimations can be seen in fig. 6.5b. This hypothesis is the same as hypothesis H_0^4, except we restrict ourselves to only considering DistilBERT based encoders. As such, we have also used the Friedman test, only this time there were only 12 triples.

We have obtained p-value of 0.002, which leads us to reject the null hypothesis H_0^5 in favor of the alternative H_A^5, which means that the choice of pooling method is important for DistilBERT based encoders.

Hypothesis $\mathbf{H_0^6}$: The median F1-score is the same for encoders trained using One Tower architecture and those trained using the Two Tower architecture.

Alternative $\mathbf{H_A^6}$: The median F1-score differs depending on the training architecture.

Null hypothesis H_0^6 states that there is not a difference in the performance of the encoders trained using One Tower training compared to those using Two Towers. The density estimations can be seen in fig. 6.6.

This hypothesis was tested using the two-sided Wilcoxon matched Signed-rank test. We were able to pair each training configuration using One Tower architecture with its corresponding match using the Two Tower architecture. There were 36 pairs.

The test produced a p-value of 0.987, which does not allow us to reject the null hypothesis H_0^6. We conclude that, in general, there was not a difference in encoder quality based on the ICT training architecture.

Hypothesis $\mathbf{H_0^7}$: The median difference in F1-scores across all datasets between the DistilRoBERTa based encoders using the Mean pooling compared to corresponding encoders using Cls pooling is negative.

Alternative $\mathbf{H_A^7}$: The median difference in F1-scores is positive.

Null hypothesis H_0^7 states that for all DistilRoBERTa encoders using Mean pooling, trained on any datasets, the median F1-score is lower than for DistilRoBERTa encoders using Cls pooling, e.g., that using Mean generally leads to worse encoders, while the alternate hypothesis says that Mean pooling leads to better DistilRoBERTa based encoders.

We again used the one-sided Wilcoxon matched Signed-rank test, as we could create pairs of the same training configurations, with the only difference being the pooling method. There were 12 pairs.

The test resulted in a p-value of 0.909, which does allow us to reject the null hypothesis. This leads us to conclude that our initial guess in section 6.2.2 that Mean pooling is better than Cls for DistilRoBERTa is most likely incorrect.

Chapter 7

Exploration of model behaviour

In this chapter, we attempt to use current methods of explaining the decisions made by black-box models to gain insight into the inner workings of our Sentence Encoders. Specifically, in section 7.1, we explore why our Sentence Encoders performed poorly on the HDFS1-blocks dataset, and in section 7.2 we explore the predictions made by Anomaly Detection.

For both tasks, we use a technique called LIME (Local Interpretable Model-agnostic Explanations), introduced by Ribeiro, Singh and Guestrin [RSG16]. LIME is a method of creating explanations for machine learning classification models. It is a model-agnostic technique that works with any classification models which outputs class probabilities.

It provides explanations for single instances, which it does by searching the neighborhood of the instance to be explained by randomly perturbing it and examining how to output probabilities change. An example of how perturbing is typically achieved for text data is splitting the text by whitespace and leaving out words in the text to see how the class probabilities change.

7.1 LIME explanation of HDFS1-blocks model behaviour

As presented in section 6.2.2, the performance of models trained on HDFS1-blocks seems to be much worse compared to HDFS1-time, even though HDFS1-blocks should lead to creating targets and contexts with more related log-lines for ICT (see section 3.2).

We hypothesize that this low performance when training on HDFS1-blocks could be caused by the network paying too much attention to the block ID,

such that the block ID dominates the final contextual sentence embedding meaning. The cause of this may be the fact that both the *target* and the *context* texts contain the same block ID.

Setup

Exploring and explaining what has a Language Model (our Sentence Encoders) learned is a difficult task, as Language Modelling by itself is not a classification task, and such LIME is not applicable straight away. To employ LIME, we decided to explore how the Sentence Encoder assigns the probabilities of a target belonging to a particular context in the ICT task, as ICT can be seen as a classification task.

During regular ICT training, the network is given B targets and B contexts, with each target having its correct context. The network then tries to correctly pair up each target to its context, which is done by assigning probabilities of each context belonging to each target.

To employ LIME, we will restrict ourselves to exploring how the ICT assigns probabilities for each context given only a single target. We have taken data from the Anomaly Detection HDFS1 validation split, e.g., HDFS1 log-lines which the Sentence Encoder has not seen during training, and preprocessed them in the same fashion as HDFS1-blocks ICT pretraining dataset, e.g., that the chunks from which targets and contexts are created only contain log-lines with the same block ID.

We have then randomly selected $B = 32$ (*target*, *context*) pairs from this dataset. Now we can employ LIME, using the 32 contexts as the classification classes. We then used LIME to create explanations of how a target was assigned its correct context.

We explored the behaviour of the best performing Sentence Encoder trained on HDFS1-blocks, which was DistilBERT with Cls pooling trained using Two Tower architecture. Because of the Two Tower nature of this model, we used two Sentence Encoders, as this pretraining architecture produces two. One for embedding targets and one for embedding contexts. Only the one for embedding targets is used for the creation of contextual embeddings of log-lines for Anomaly Detection task, but for LIME, we used the corresponding encoders for the targets and contexts.

The class (context) probabilities are obtained similarly to the ICT pretraining illustrated in fig. 3.2. For a single target embedding $t \in \mathbb{R}^D$, where D is the embedding dimension, and the contexts embeddings $C \in \mathbb{R}^{B \times D}$, the context probabilities p are obtained as follows. Treating t as a row vector, we obtain logits $l = tC^\intercal$, $l \in \mathbb{R}^B$. We use l to create probabilities $p = \text{softmax}(l)$, $p \in [0, 1]^B$, with $\sum_{i=1}^{B} p_i = 1$.

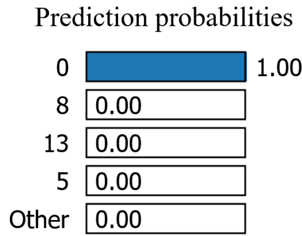


Figure 7.1: LIME prediction probabilities for different context indices for target 0

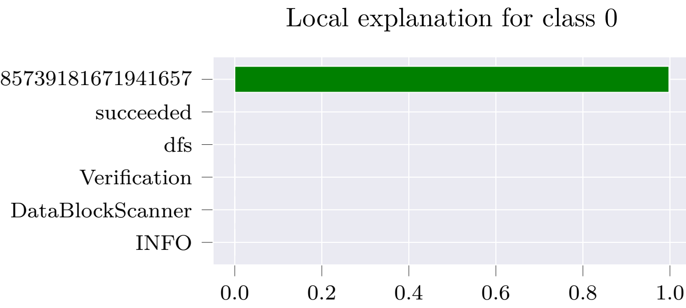


Figure 7.2: LIME explanation of the most influential words for assigning target 0 to it's corresponding context

LIME explanations

Here we present the LIME explanations for target with index 0 in our selected batch, which has a corresponding context with index 0. We have explored other indices, but the results were analogous. The target 0 can be seen in listing 7.1, and it's corresponding context 0 is seen in listing 7.2. Although timestamps are present in the raw log-lines, they are removed prior to both Sentence Encoder training and Anomaly Detection, for reasons described in section 6.1.

```
INFO dfs.DataBlockScanner: Verification succeeded for blk_1585739181671941657
```

Listing 7.1: Target log-line with index 0 in our explored batch

```
INFO dfs.DataNode$PacketResponder: PacketResponder 2 for block
      blk_1585739181671941657 terminating
INFO dfs.DataNode$PacketResponder: Received block blk_1585739181671941657 of size
      67108864 from /10.251.74.134
WARN dfs.DataNode$DataXceiver: 10.251.74.134:50010:Got exception while serving
      blk_1585739181671941657 to /10.251.38.53:
INFO dfs.DataNode$DataXceiver: 10.251.74.134:50010 Served block blk_1585739181671941657
       to /10.251.38.53
WARN dfs.DataNode$DataXceiver: 10.251.74.134:50010:Got exception while serving
      blk_1585739181671941657 to /10.251.74.134:
INFO dfs.FSNamesystem: BLOCK* NameSystem.delete: blk_1585739181671941657 is added
      to invalidSet of 10.251.106.214:50010
INFO dfs.FSNamesystem: BLOCK* NameSystem.delete: blk_1585739181671941657 is added
      to invalidSet of 10.251.39.64:50010
INFO dfs.FSNamesystem: BLOCK* NameSystem.delete: blk_1585739181671941657 is added
      to invalidSet of 10.251.74.134:50010
INFO dfs.FSDataset: Deleting block blk_1585739181671941657 file /mnt/hadoop/dfs/data/
      current/subdir49/blk_1585739181671941657
```

Listing 7.2: Context log-lines with index 0 in our explored batch

Figure 7.1 shows the LIME output for prediction probabilities of the most likely classes for target 0. In our case, the classes are the contexts, which are represented in the figure by their index. We see that the probability of the context with index 0 is 1, e.g., the network is certain that target 0 belongs to

context 0, which is correct, and shows 0 probability for the other classes. It being so sure is not a good sign, as we would expect the probabilities to be more "fuzzy" (given the assumption that the contexts are not significantly different).

Figure 7.2 gives us more insight into what words from target 0 influenced the probability for context 0. LIME produces similar graphs for each of the classes, e.g., for each context in the batch in our case. But except for this graph, where we see that the block ID dominates all other words from the target, other graphs had the effects of other words close to 0.

When trying to get explanations for the remaining targets i, the situation was analogous to figs. 7.1 and 7.2, with the context i having probability 1 or very close to it, and the explanation graphs showing the block ID being the only word with significant impact.

All of this supports our hypothesis that models trained on HDFS1-blocks do learn to only primarily represent the block IDs in their sentence embeddings.

7.2 LIME explanation of Anomaly Detection predictions

We wanted to see whether we could also use LIME to explain the decisions made by the Anomaly Detector. As the HDFS1 Anomaly Detection dataset has labels per block (a group of log-lines with the same block ID), we wanted to see whether LIME could be used to identify the most influential lines in a block.

Setup

To employ LIME, we need prediction probabilities for the Normal and Anomalous classes. The AETCN method we use unfortunately does not provide the probabilities, as it works by thresholding the reconstruction error and gives only binary output. To get around this, we decided to emulate prediction probabilities by using a sigmoid curve with its midpoint at the decision threshold, e.g., at the threshold, there is a 50% probability of the block of log-lines being Anomalous or Normal.

We devised a translated sigmoid function $\mathcal{T}_c^\alpha$, which can be seen in eq. (7.1), where c is a parameter, such that $\mathcal{T}_c^\alpha(c) = 0.5$, and α is the parameter such that $\mathcal{T}_c^\alpha(0) = \sigma(\alpha)$, where $\sigma(x) = (1 + e^{-x})^{-1}$ is the classic sigmoid function.

$$\mathcal{T}_c^{\alpha}(x) = \sigma\left(\frac{-\alpha \cdot (x - c)}{c}\right) \tag{7.1}$$

We then used $\mathcal{T}_c^{\alpha}$ to create probabilities for the Normal class (with the Anomalous probabilities being the complement) given the reconstruction error. Specifically, we used $\mathcal{T}_{th}^{7}$, where th is the found threshold associated with the currently examined AETCN detector, e.g., such as threshold which maximizes the F1-score on the validation dataset, as described in section 3.5. We used $\alpha = 7$, such that the probability of the Normal class given the reconstruction error is 0 is 99.9%. The function can be seen in fig. 7.3.

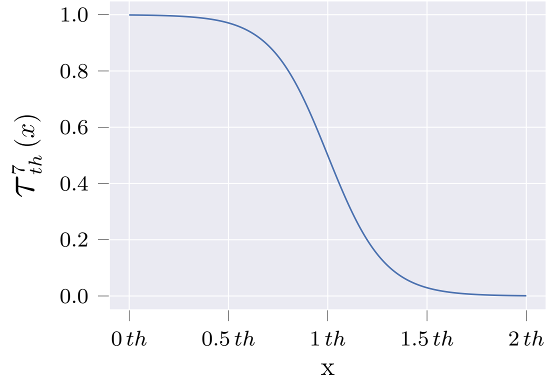


Figure 7.3: Translated Sigmoid for probability of the Normal class

We also had to modify LIME's text splitting algorithm, as usually it removes whole words to create a neighborhood for the instance it is trying to explain, but we wanted to examine the effect of whole log-lines. To achieve that, we set LIME to split the input instance by the newline symbol, so when passing it multiple log-lines as a single string separated by newlines, it would treat whole lines as the building blocks.

LIME explanation

Here we present the anomaly detection explanations using embeddings produced by the best Sentence Encoder, the DistilBERT with Cls pooling trained using One Tower ICT on the Mix Concat dataset.

We will show the LIME explanations in detail only for block 8367791625462110565 from the HDFS1-validation dataset, to illustrate how LIME could be used for understanding the anomaly detection prediction and identifying the important log-lines within a block.

Block 8367791625462110565 contains 33 lines and has the Anomalous label. We prefixed each line with its index for presentation purposes, but the model did not contain the indices when passed to the model. The model correctly identifies it as anomalous, with probability of 63%, when using our $\mathcal{T}_{th}^{7}$ function, e.g., the reconstruction error was slightly higher than the threshold.

Figure 7.4 shows the influence of specific lines on the classification. The

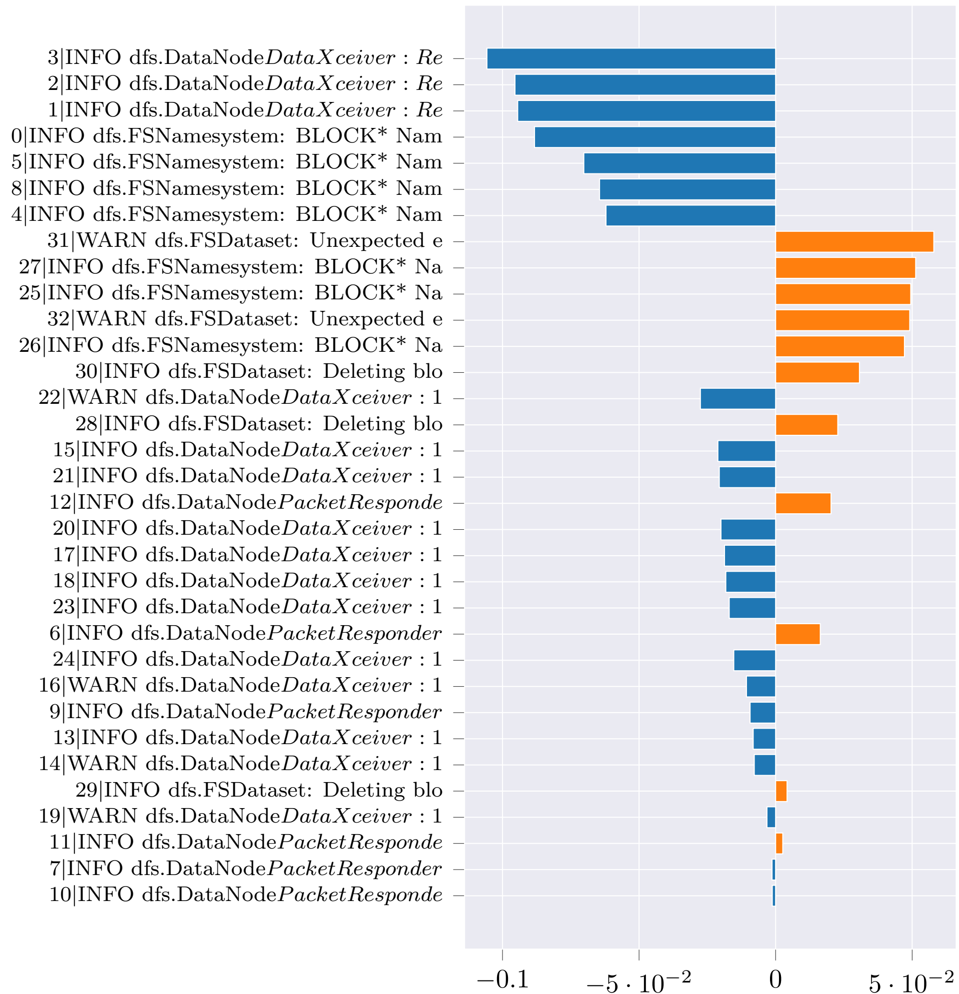


Figure 7.4: LIME explanation for block 8367791625462110565, showing the influence of it's lines on the Anomalous classification, with orange signifying positive influence and blue negative.

influences are not probabilities, as by purely examining the figure, it would seem that the block was classified as Normal. The numerical value of the influences is the weight of the given line in the internal LIME model and only represents the relative influence size of the different lines.

We can use these obtained weights from the internal LIME model to color the individual log-lines in the block, to quickly judge which lines might be causing the block to be Anomalous. This coloring can be seen in listing 7.3. A quick glance shows us that the model identifies several INFO level lines almost at the end, together with WARN lines at the end as the possible culprits of what Anomalous behaviour happened in this block.

We see that this usage of LIME can help with diagnosing anomalies and possibly save time for the engineers tending to the system. Nevertheless, the method is not without issues. We found out that for blocks comprised of only a few lines, LIME was not able to find any meaningful weights for them, as there was not a large enough neighborhood around the instance to find meaningful weights. A concrete example, given a block of 2 lines, there are only 4 possible "neighbor" instances, one empty instance, only the first line, only the second line, and the whole original block. This is not enough data to generate an explanation, but this should not be an issue in the real world, as a human can examine such a block themselves.

Another possible issue is that the weights need to be taken with a grain of salt. The Anomaly Detection model itself is most likely not perfect, and LIME works by trying to approximate it locally, e.g., caution must be taken when interpreting the LIME explanations, as they are not guaranteed to hold true.

Nevertheless, when taking the explanation uncertainty into account, we pose that this usage of LIME can be a beneficial helping tool when diagnosing anomaly detection predictions and whether they are trustworthy.

Listing 7.3: LIME coloring of each line in an anomalous block, which provides visual aid to finding which log-lines contributed most to either Normal (blue) or Anomalous (orange) classification

```
0|INFO dfs.FSNamesystem: BLOCK* NameSystem.allocateBlock: /user/root/randtxt/_tempora
ry/_task_200811092030_0003_m_000983_0/part-00983. blk_8367791625462110565
1|INFO dfs.DataNode$DataXceiver: Receiving block blk_8367791625462110565 src: /10.251.195.
33:57003 dest: /10.251.195.33:50010
2|INFO dfs.DataNode$DataXceiver: Receiving block blk_8367791625462110565 src: /10.251.195.
33:39521 dest: /10.251.195.33:50010
3|INFO dfs.DataNode$DataXceiver: Receiving block blk_8367791625462110565 src: /10.251.203.
166:32809 dest: /10.251.203.166:50010
4|INFO dfs.FSNamesystem: BLOCK* NameSystem.addStoredBlock: blockMap updated: 10.251.
203.166:50010 is added to blk_8367791625462110565 size 67108864
5|INFO dfs.FSNamesystem: BLOCK* NameSystem.addStoredBlock: blockMap updated: 10.251.
39.192:50010 is added to blk_8367791625462110565 size 67108864
6|INFO dfs.DataNode$PacketResponder: PacketResponder 0 for block blk_836779162546211056
5 terminating
7|INFO dfs.DataNode$PacketResponder: Received block blk_8367791625462110565 of size 67108
864 from /10.251.203.166
8|INFO dfs.FSNamesystem: BLOCK* NameSystem.addStoredBlock: blockMap updated: 10.251.
195.33:50010 is added to blk_8367791625462110565 size 67108864
9|INFO dfs.DataNode$PacketResponder: PacketResponder 1 for block blk_836779162546211056
5 terminating
10|INFO dfs.DataNode$PacketResponder: Received block blk_8367791625462110565 of size 6710
8864 from /10.251.195.33
11|INFO dfs.DataNode$PacketResponder: PacketResponder 2 for block blk_836779162546211056
5 terminating
12|INFO dfs.DataNode$PacketResponder: Received block blk_8367791625462110565 of size 6710
8864 from /10.251.195.33
```

13|INFO dfs.DataNode$DataXceiver: 10.251.39.192:50010 Served block blk_836779162546211056
5 to /10.251.39.192

14|WARN dfs.DataNode$DataXceiver: 10.251.195.33:50010:Got exception while serving blk_836
7791625462110565 to /10.251.214.112:

15|INFO dfs.DataNode$DataXceiver: 10.251.195.33:50010 Served block blk_836779162546211056
5 to /10.250.5.237

16|WARN dfs.DataNode$DataXceiver: 10.251.203.166:50010:Got exception while serving blk_83
67791625462110565 to /10.251.214.112:

17|INFO dfs.DataNode$DataXceiver: 10.251.203.166:50010 Served block blk_83677916254621105
65 to /10.251.203.166

18|INFO dfs.DataNode$DataXceiver: 10.251.195.33:50010 Served block blk_836779162546211056
5 to /10.251.195.33

19|WARN dfs.DataNode$DataXceiver: 10.251.203.166:50010:Got exception while serving blk_83
67791625462110565 to /10.251.214.112:

20|INFO dfs.DataNode$DataXceiver: 10.251.39.192:50010 Served block blk_836779162546211056
5 to /10.251.39.192

21|INFO dfs.DataNode$DataXceiver: 10.251.203.166:50010 Served block blk_83677916254621105
65 to /10.250.5.237

22|WARN dfs.DataNode$DataXceiver: 10.251.195.33:50010:Got exception while serving blk_836
7791625462110565 to /10.251.214.112:

23|INFO dfs.DataNode$DataXceiver: 10.251.203.166:50010 Served block blk_83677916254621105
65 to /10.251.203.166

24|INFO dfs.DataNode$DataXceiver: 10.251.195.33:50010 Served block blk_836779162546211056
5 to /10.250.5.237

25|INFO dfs.FSNamesystem: BLOCK* NameSystem.delete: blk_8367791625462110565 is added
to invalidSet of 10.251.195.33:50010

26|INFO dfs.FSNamesystem: BLOCK* NameSystem.delete: blk_8367791625462110565 is added
to invalidSet of 10.251.203.166:50010

27|INFO dfs.FSNamesystem: BLOCK* NameSystem.delete: blk_8367791625462110565 is added
to invalidSet of 10.251.39.192:50010

28|INFO dfs.FSDataset: Deleting block blk_8367791625462110565 file /mnt/hadoop/dfs/data/c
urrent/subdir34/blk_8367791625462110565

29|INFO dfs.FSDataset: Deleting block blk_8367791625462110565 file /mnt/hadoop/dfs/data/c
urrent/subdir49/blk_8367791625462110565

30|INFO dfs.FSDataset: Deleting block blk_8367791625462110565 file /mnt/hadoop/dfs/data/c
urrent/subdir30/blk_8367791625462110565

31|WARN dfs.FSDataset: Unexpected error trying to delete block blk_8367791625462110565. Bl
ockInfo not found in volumeMap.

32|WARN dfs.FSDataset: Unexpected error trying to delete block blk_8367791625462110565. Bl
ockInfo not found in volumeMap.

Discussion

Here we present the main takeaways from our thesis, mainly from our experiments (chapter 6) and hypotheses (section 6.3).

We have found out and statistically verified in hypothesis H_0^1 that training on Mix datasets produces better Sentence Encoders than training on HDFS1 datasets alone, showing the potential of BERT networks for learning semantics hidden in varied textual data, even log lines.

We also obtained a statistically significant finding in hypothesis H_0^2 that DistilBERT based encoders perform better compared to DistilRoBERTa encoders. We theorize that this difference could come from the different pretraining approaches the original models employed, with RoBERTa not having any pretraining task which would use the [CLS] token. Another factor could be that RoBERTa has double the amount of tokens in its vocabulary, which could mean that it may need to be trained for longer.

Another statistically significant finding stemming from hypothesis H_0^3 is that Smart Average context truncation strategy resulted in better encoders compared to using the Concatenation to max tokens strategy. This could point to the Sentence Encoder learning better by "seeing" the beginnings of more sentences in a context compared to having fewer full-length sentences in a context.

We have not been able to show in general whether using One Tower or Two Tower ICT pretraining architecture is better (hypothesis H_0^6), as usually the performance of the resulting encoders was similar, with one approach sometimes edging out the other in the same scenario and vice versa. We theorise that our log datasets are not varied enough for the contexts to differ enough semantically from the targets to fully utilise the flexibility offered by having separate encoders for each. It may be the case that using One Tower

may result in faster training of the Sentence Encoder, as it could be learning semantics from both target and the context at the same time.

Similarly to ICT architectures, we have not been able to show whether there is a difference between the pooling strategies (hypothesis H_0^4). However, we have been able to show that there is a difference in quality between the pooling strategies when only considering DistilBERT based encoders in hypothesis H_0^5. Together with fig. 6.5b we found that for DistilBERT Cls pooling is the best, followed by Mean, with LastNCls being the last.

None of our Sentence Encoders were able to beat the baseline fastText embeddings, which achieved an F1-score of 0.85, while the best configuration, DistilBERT with Cls pooling using One Tower ICT trained for 3 epochs on Mix Concat dataset, achieved F1 of 0.79, being 93% of the baseline. When averaging metrics over all training datasets in table 6.5, we found out that the best Sentence Encoder training configuration in general was DistilBERT with Cls pooling using Two Tower ICT.

One explanation of why our contextual embeddings have not improved the anomaly detection performance is that the HDFS1 dataset is not expressive enough, i.e., the natural language portions of the logs are not varied enough for contextual models to bring an advantage over high-quality global embedding techniques, such as fastText. But high-quality labeled log anomaly detection datasets with interesting text are hard to find, which leaves the exploration of this hypothesis as future work.

In section 7.1, we found out that the issue with sentence encoders trained on HDFS1-blocks dataset (section 4.3.1) is that they learn to only pay attention to the block ID present in each line. Our solution for preventing the network from only paying attention to this block ID was to create the Mix datasets (section 4.3.2), which contain log-lines from multiple different datasets. Another approach that may be worth exploring in future work would be to keep the HDFS1-blocks dataset, but to either remove the block IDs or replace them with random identifiers, such that the network would be forced to pay more attention to the semantics of the logs.

Conclusion

This thesis explored the applicability of Contextual Embedding methods for creating sentence embedding vectors from log-lines for use in the domain of Log Anomaly Detection. There have been successes in the past few years of applying Natural Language Processing techniques, particularly fastText, for the creation of vector embeddings of log-lines for use with anomaly detection methods. However, fastText only creates global embeddings, i.e., embeddings missing contextual information.

To explore the use of more advanced contextual embedding methods, we have created Sentence Encoders using BERT (section 2.5) models as their main component, which are the current state-of-the-art NLP neural network architectures. BERT models are capable of processing input strings of variable length, while producing a variable amount of vectors. We explored several methods for pooling the BERT output to reduce this variable amount of vectors into a single log-line embedding vector.

We have selected the Inverse Cloze Task as the training task for unsupervised training of our BERT-based Sentence encoders and created the necessary data preprocessing pipeline, which enabled us to use the large amount of publicly available raw unlabeled log datasets for training.

Our experiments took over 1824 combined GPU-compute hours, as we have trained 72 Sentence Encoders and evaluated the quality of their produced embeddings by using them to embed the labeled HDFS1 dataset to perform an Anomaly Detection task. The anomaly detection performance was then compared to baseline performance obtained by using fastText embeddings of the same dataset.

Although none of our Sentence Encoders were able to match the baseline, they still performed well, with the best achieving an F1-score 93% of the

fastText F1-score. This shows there is still promise for the use of contextual embeddings for anomaly detection, as the HDFS1 dataset is not expressive enough to reap the benefits of contextual models compared to global embedding models such as fastText. This result of approaching the baseline also verifies our choice of ICT as a valid unsupervised pretraining task.

We have also explored several ways of preparing the dataset for ICT pretraining and have found out that since the total length of the input BERT accepts is limited, it is better to use smart average context truncation (section 3.3.3), instead of just plain concatenation of the log-lines in the context and then truncating to the maximum BERT input length. This is a possible venue of future work, which could explore using different amount of log-lines in the context, and other ICT pretraining parameters which were not explored in this thesis, which could be done in conjunction with the usage of BERT models capable of processing longer inputs.

Interesting future work could be done on using larger BERT models as the base for the Sentence Encoders, which may have higher learning capacity than the distilled, less parametrized BERT models used in this thesis.

Lastly, outside the main topic of this thesis, we have done preliminary work on using techniques for explaining machine learning models for explaining Anomaly Detection predictions and have shown it to be another possible interesting venue for future work.

Appendix A

Bibliography

[BCB14] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio, *Neural machine translation by jointly learning to align and translate*, arXiv preprint arXiv:1409.0473 (2014).

[BGJM17] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov, *Enriching word vectors with subword information*, Transactions of the Association for Computational Linguistics **5** (2017), 135–146.

[CYC+20] Wei-Cheng Chang, Felix X Yu, Yin-Wen Chang, Yiming Yang, and Sanjiv Kumar, *Pre-training tasks for embedding-based large-scale retrieval*, arXiv preprint arXiv:2002.03932 (2020).

[CZL+04] Mike Chen, Alice X Zheng, Jim Lloyd, Michael I Jordan, and Eric Brewer, *Failure diagnosis using decision trees*, International Conference on Autonomic Computing, 2004. Proceedings., IEEE, 2004, pp. 36–43.

[DCLT18] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova, *Bert: Pre-training of deep bidirectional transformers for language understanding*, arXiv preprint arXiv:1810.04805 (2018).

[DLZS17] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar, *Deeplog: Anomaly detection and diagnosis from system logs through deep learning*, Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, 2017, pp. 1285–1298.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber, *Long short-term memory*, Neural computation **9** (1997), no. 8, 1735–1780.

[HVD15] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean, *Distilling the knowledge in a neural network*, arXiv preprint arXiv:1503.02531 (2015).

[HZHL20] Shilin He, Jieming Zhu, Pinjia He, and Michael R Lyu, *Loghub: a large collection of system log datasets towards automated log analytics*, arXiv preprint arXiv:2008.06448 (2020).

[Kor21] Martin Koryťák, *Anomaly detection methods for log files*, Master's thesis, Czech Technical University in Prague, 2021.

[LCT19] Kenton Lee, Ming-Wei Chang, and Kristina Toutanova, *Latent retrieval for weakly supervised open domain question answering*, arXiv preprint arXiv:1906.00300 (2019).

[LFV+16] Colin Lea, Michael D Flynn, René Vidal, Austin Reiter, and Gregory D Hager, *Temporal convolutional networks for action segmentation and detection. corr abs/1611.05267 (2016)*, arXiv preprint arXiv:1611.05267 (2016).

[LKB20] Qi Liu, Matt J Kusner, and Phil Blunsom, *A survey on contextual embeddings*, arXiv preprint arXiv:2003.07278 (2020).

[LOG+19] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov, *Roberta: A robustly optimized bert pretraining approach*, arXiv preprint arXiv:1907.11692 (2019).

[LTZ08] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou, *Isolation forest*, 2008 eighth ieee international conference on data mining, IEEE, 2008, pp. 413–422.

[LZL+16] Qingwei Lin, Hongyu Zhang, Jian-Guang Lou, Yu Zhang, and Xuewei Chen, *Log clustering based problem identification for online service systems*, 2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C), IEEE, 2016, pp. 102–111.

[LZXS07] Yinglung Liang, Yanyong Zhang, Hui Xiong, and Ramendra Sahoo, *Failure prediction in ibm bluegene/l event logs*, Seventh IEEE International Conference on Data Mining (ICDM 2007), IEEE, 2007, pp. 583–588.

[MCCD13] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean, *Efficient estimation of word representations in vector space*, arXiv preprint arXiv:1301.3781 (2013).

[MSC+13] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean, *Distributed representations of words and phrases and their compositionality*, arXiv preprint arXiv:1310.4546 (2013).

[OS07] Adam Oliner and Jon Stearley, *What supercomputers say: A study of five system logs*, 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07), IEEE, 2007, pp. 575–584.

[PGM+19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala, *Pytorch: An imperative style, high-performance deep learning library*, Advances in Neural Information Processing Systems 32 (H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, eds.), Curran Associates, Inc., 2019, pp. 8024–8035.

[PNI+18] Matthew E Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer, *Deep contextualized word representations*, arXiv preprint arXiv:1802.05365 (2018).

[RG19] Nils Reimers and Iryna Gurevych, *Sentence-bert: Sentence embeddings using siamese bert-networks*, arXiv preprint arXiv:1908.10084 (2019).

[RNSS18] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever, *Improving language understanding by generative pre-training*.

[RSG16] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin, *"why should I trust you?": Explaining the predictions of any classifier*, Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016, 2016, pp. 1135–1144.

[RWC+19] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever, *Language models are unsupervised multitask learners*, OpenAI blog **1** (2019), no. 8, 9.

[SDCW19] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf, *Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter*, arXiv preprint arXiv:1910.01108 (2019).

[SHB15] Rico Sennrich, Barry Haddow, and Alexandra Birch, *Neural machine translation of rare words with subword units*, arXiv preprint arXiv:1508.07909 (2015).

[Sou20] Marek Souček, *Log anomaly detection*, Master's thesis, Czech Technical University in Prague, 2020.

[SWS+99] Bernhard Schölkopf, Robert C Williamson, Alexander J Smola, John Shawe-Taylor, John C Platt, et al., *Support vector method for novelty detection.*, NIPS, vol. 12, Citeseer, 1999, pp. 582–588.

[Tay53] Wilson L Taylor, *"cloze procedure": A new tool for measuring readability*, Journalism quarterly **30** (1953), no. 4, 415–433.

[TRDG20] Nandan Thakur, Nils Reimers, Johannes Daxenberger, and Iryna Gurevych, *Augmented sbert: Data augmentation method for improving bi-encoders for pairwise sentence scoring tasks*, arXiv preprint arXiv:2010.08240 (2020).

[VRD09] Guido Van Rossum and Fred L. Drake, *Python 3 reference manual*, CreateSpace, Scotts Valley, CA, 2009.

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin, *Attention is all you need*, arXiv preprint arXiv:1706.03762 (2017).

[WDS+20] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush, *Transformers: State-of-the-art natural language processing*, Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations (Online), Association for Computational Linguistics, October 2020, pp. 38–45.

[WLvP+20] Thomas Wolf, Quentin Lhoest, Patrick von Platen, Yacine Jernite, Mariama Drame, Julien Plu, Julien Chaumond, Clement Delangue, Clara Ma, Abhishek Thakur, Suraj Patil, Joe Davison, Teven Le Scao, Victor Sanh, Canwen Xu, Nicolas Patry, Angie McMillan-Major, Simon Brandeis, Sylvain Gugger, François Lagunas, Lysandre Debut, Morgan Funtowicz, Anthony Moi, Sasha Rush, Philipp Schmidd, Pierric Cistac, Victor Muštar, Jeff Boudier, and Anna Tordjmann, *Datasets*, GitHub. Note: https://github.com/huggingface/datasets **1** (2020).

[WSC+16] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al., *Google's neural machine translation system: Bridging the gap between human and machine translation*, arXiv preprint arXiv:1609.08144 (2016).

[XHF+09] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I Jordan, *Detecting large-scale system problems by mining console logs*, Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, 2009, pp. 117–132.

[ZP17] Chong Zhou and Randy C Paffenroth, *Anomaly detection with robust deep autoencoders*, Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining, 2017, pp. 665–674.

Appendix B

All experiment results

Here we list all the raw Anomaly Detection results from chapter 6.

Rank	Model Type	Pooling	Towers	Epochs	Dataset	Precision	Recall	F1-score	% of baseline
1	fastText	-	-	-	HDFS1 train	0.7768	**0.9365**	**0.8492**	100.00%
2	DistilBERT	Cls	1T	3	Mix Concat	0.8882	0.7126	0.7908	93.12%
3	DistilBERT	Mean	1T	3	Mix SmartAvg	0.9026	0.6770	0.7737	91.10%
4	DistilBERT	Cls	2T	4	HDFS1 time	0.7418	0.8052	0.7722	90.93%
5	DistilBERT	Cls	1T	3	Mix SmartAvg	0.8604	0.6989	0.7713	90.82%
6	DistilBERT	Mean	1T	5	Mix SmartAvg	0.8972	0.6740	0.7698	90.64%
7	DistilBERT	Cls	2T	3	Mix SmartAvg	0.8755	0.6722	0.7605	89.55%
8	DistilBERT	Cls	1T	5	Mix SmartAvg	0.8747	0.6716	0.7598	89.47%
9	DistilBERT	Mean	1T	5	Mix Concat	0.8822	0.6538	0.7510	88.44%
10	DistilBERT	Mean	2T	3	Mix SmartAvg	0.8568	0.6681	0.7508	88.40%
11	DistilBERT	Cls	1T	5	Mix Concat	0.8920	0.6425	0.7470	87.96%
12	DistilBERT	Mean	1T	3	Mix Concat	0.8610	0.6473	0.7390	87.02%
13	DistilBERT	Cls	2T	5	Mix SmartAvg	0.8741	0.6306	0.7327	86.28%
14	DistilBERT	Cls	2T	5	Mix Concat	0.8379	0.6508	0.7326	86.27%
15	DistilBERT	LastNCls	1T	3	Mix SmartAvg	0.8753	0.6211	0.7266	85.57%
16	DistilBERT	Mean	2T	5	Mix SmartAvg	0.8557	0.6128	0.7142	84.10%

Rank	Model Type	Pooling	Towers	Epochs	Dataset	Precision	Recall	F1-score	% of baseline
17	DistilBERT	LastNCls	2T	3	Mix SmartAvg	0.8295	0.6182	0.7084	83.42%
18	DistilBERT	Mean	2T	4	HDFS1 time	0.8449	0.6081	0.7072	83.27%
19	DistilBERT	LastNCls	1T	3	Mix Concat	0.8518	0.6039	0.7067	83.22%
20	DistilBERT	LastNCls	1T	5	Mix SmartAvg	0.8807	0.5742	0.6952	81.86%
21	DistilBERT	Mean	2T	3	Mix Concat	0.7971	0.6134	0.6933	81.64%
22	DistilRoBERTa	Mean	2T	3	Mix SmartAvg	0.7912	0.6164	0.6929	81.60%
23	DistilBERT	Mean	2T	5	Mix Concat	0.8674	0.5748	0.6914	81.42%
24	DistilBERT	Mean	1T	4	HDFS1 time	0.7799	0.6188	0.6901	81.26%
25	DistilRoBERTa	Cls	2T	3	Mix Concat	0.8059	0.6015	0.6889	81.12%
26	DistilRoBERTa	Cls	2T	3	Mix SmartAvg	0.8448	0.5784	0.6866	80.86%
27	DistilBERT	LastNCls	2T	4	HDFS1 time	0.8329	0.5802	0.6839	80.54%
28	DistilRoBERTa	Mean	2T	3	Mix Concat	0.8101	0.5903	0.6829	80.42%
29	DistilBERT	LastNCls	2T	5	Mix SmartAvg	0.8871	0.5505	0.6794	80.00%
30	DistilBERT	LastNCls	2T	3	Mix Concat	0.8718	0.5534	0.6771	79.73%
31	DistilBERT	LastNCls	2T	5	Mix Concat	0.8996	0.5374	0.6729	79.23%
32	DistilRoBERTa	LastNCls	1T	5	Mix SmartAvg	0.7331	0.6164	0.6697	78.86%
33	DistilBERT	Cls	2T	3	Mix Concat	0.8403	0.5529	0.6669	78.53%
34	DistilBERT	LastNCls	1T	4	HDFS1 time	0.8093	0.5570	0.6599	77.70%
35	DistilRoBERTa	LastNCls	2T	3	Mix SmartAvg	0.7409	0.5926	0.6585	77.55%
36	DistilBERT	Cls	1T	4	HDFS1 time	0.7827	0.5647	0.6561	77.26%
37	DistilBERT	LastNCls	1T	5	Mix Concat	0.8900	0.5190	0.6557	77.21%
38	DistilRoBERTa	Cls	2T	5	Mix Concat	0.7285	0.5831	0.6478	76.28%
39	DistilRoBERTa	Cls	1T	3	Mix SmartAvg	0.6985	0.5956	0.6429	75.71%
40	DistilBERT	Cls	2T	4	HDFS1 blocks	0.7916	0.5392	0.6415	75.54%
41	DistilRoBERTa	Cls	2T	5	Mix SmartAvg	0.7850	0.5398	0.6397	75.33%
42	DistilRoBERTa	LastNCls	1T	3	Mix Concat	0.7114	0.5796	0.6387	75.22%
43	DistilRoBERTa	LastNCls	1T	5	Mix Concat	0.6780	0.6015	0.6375	75.07%

Rank	Model Type	Pooling	Towers	Epochs	Dataset	Precision	Recall	F1-score	% of baseline
44	DistilRoBERTa	LastNCls	1T	4	HDFS1 time	0.6432	0.6306	0.6369	75.00%
45	DistilRoBERTa	Cls	1T	3	Mix Concat	0.7609	0.5309	0.6254	73.64%
46	DistilRoBERTa	Mean	1T	4	HDFS1 blocks	0.5811	0.6722	0.6233	73.40%
47	DistilRoBERTa	LastNCls	2T	5	Mix SmartAvg	0.7494	0.5184	0.6128	72.17%
48	DistilRoBERTa	Cls	1T	5	Mix Concat	0.7068	0.5368	0.6102	71.85%
49	DistilRoBERTa	Mean	1T	3	Mix Concat	0.7423	0.5166	0.6092	71.74%
50	DistilRoBERTa	Mean	1T	5	Mix SmartAvg	0.8201	0.4846	0.6092	71.73%
51	DistilRoBERTa	LastNCls	2T	4	HDFS1 time	0.8252	0.4822	0.6087	71.68%
52	DistilRoBERTa	Mean	1T	3	Mix SmartAvg	0.6037	0.5843	0.5938	69.93%
53	DistilRoBERTa	Cls	1T	5	Mix SmartAvg	0.8333	0.4602	0.5930	69.82%
54	DistilRoBERTa	Mean	2T	5	Mix SmartAvg	0.7739	0.4798	0.5924	69.76%
55	DistilRoBERTa	Cls	2T	4	HDFS1 time	0.9179	0.4317	0.5872	69.15%
56	DistilRoBERTa	LastNCls	2T	5	Mix Concat	0.9750	0.4169	0.5840	68.77%
57	DistilRoBERTa	Mean	2T	5	Mix Concat	0.7314	0.4834	0.5821	68.54%
58	DistilRoBERTa	Mean	2T	4	HDFS1 time	0.9292	0.4210	0.5795	68.24%
59	DistilBERT	Mean	1T	4	HDFS1 blocks	0.9747	0.4115	0.5787	68.15%
60	DistilBERT	Mean	2T	4	HDFS1 blocks	0.9760	0.4103	0.5778	68.03%
61	DistilBERT	Cls	1T	4	HDFS1 blocks	0.9801	0.4091	0.5773	67.98%
62	DistilBERT	LastNCls	1T	4	HDFS1 blocks	0.9785	0.4062	0.5741	67.60%
63	DistilRoBERTa	LastNCls	2T	4	HDFS1 blocks	0.9717	0.4074	0.5741	67.60%
64	DistilRoBERTa	LastNCls	1T	3	Mix SmartAvg	0.8776	0.4258	0.5734	67.52%
65	DistilRoBERTa	Mean	1T	5	Mix Concat	0.7766	0.4501	0.5699	67.11%
66	DistilRoBERTa	Cls	1T	4	HDFS1 time	0.8181	0.4353	0.5682	66.91%
67	DistilBERT	LastNCls	2T	4	HDFS1 blocks	0.9537	0.4032	0.5668	66.74%
68	DistilRoBERTa	LastNCls	2T	3	Mix Concat	0.8932	0.4121	0.5640	66.41%
69	DistilRoBERTa	Mean	1T	4	HDFS1 time	0.8637	0.4139	0.5596	65.90%
70	DistilRoBERTa	LastNCls	1T	4	HDFS1 blocks	0.9612	0.3824	0.5472	64.43%

Rank	Model Type	Pooling	Towers	Epochs	Dataset	Precision	Recall	F1-score	% of baseline
71	DistilRoBERTa	Cls	2T	4	HDFS1 blocks	0.9951	0.3646	0.5337	62.84%
72	DistilRoBERTa	Mean	2T	4	HDFS1 blocks	**1.0000**	0.3622	0.5318	62.62%
73	DistilRoBERTa	Cls	1T	4	HDFS1 blocks	**1.0000**	0.3622	0.5318	62.62%

Appendix C

List of Attachements

- Source code for the work done in this thesis.