Master Thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

Contextual Embeddings for Predictions Based on Log Files

Bc. Petra Vaňková

Supervisor: Ing. Jan Drchal, Ph.D.
Field of study: Open Informatics
Subfield: Cybersecurity
August 2021

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Vaňková** Jméno: **Petra** Osobní číslo: **492583**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Specializace: **Kybernetická bezpečnost**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Kontextové reprezentace pro predikce založené souborech logů

Název diplomové práce anglicky:

Contextual Embeddings for Predictions Based on 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 downstream prediction 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) Evaluate the method on a downstream task of manufacturing process duration prediction based on a dataset by Škoda a.s. (will be supplied by the supervisor).

Seznam doporučené literatury:

[1] Wang, Jin, et al. "LogEvent2vec: LogEvent-to-Vector Based Anomaly Detection for Large-Scale Logs in Internet of Things." Sensors 20.9 (2020): 2451.
[2] Devlin, Jacob, et al. "Bert: Pre-training of deep bidirectional transformers for language understanding." arXiv preprint arXiv:1810.04805 (2018).
[3] Marek Souček, "Log Anomaly Detection", master thesis, supervisor Jan Drchal, FEE CTU, 2020.
[4] 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.
[5] 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).

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: **11.02.2021** Termín odevzdání diplomové práce: **13.08.2021**

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

Diplomantka bere na vědomí, že je povinna 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 studentky

Acknowledgements

I would like to express my sincere gratitude to my supervisor Ing. Jan Drchal, Ph.D., for his guidance, support and patience, to the CTU in Prague for allowing me to pursue education in a relatively new field for me and lastly to my family for always being supportive and tolerant during my studies.

Declaration

I declare that I elaborated this thesis on my own and that I mentioned all the information sources that have been used in accordance with the Guideline for adhering to ethical principles in the course of elaborating an academic final thesis.

In Prague, 13. August 2021

Abstract

Contextual embeddings have become an significant part of Natural Language Processing tasks, being able to capture information of a word in a context of a whole text sequence.

Current state-of-the-art techniques of contextual embeddings are studied in this work in terms of log analysis. One of such techniques - a model based on BERT - is used to derive contextual embeddings of log lines included in a custom dataset provided. To evaluate the quality of embeddings created, several regression analysis models were chosen - Temporal Convolutional Network, the Longformer - also a BERT-based model and *fastText* technique as a baseline approach. The aim of the models is to be able to predict certain values in the dataset. Unfortunately, none of these models was able to learn on the dataset.

Anomaly detection problem was chosen as a simplification of the regression task and the TCN model was chosen for the evaluation. As neither this approach produced better results, leveraging of another dataset with similar features could verify whether the original dataset is a suitable choice for contextual embeddings evaluation.

Keywords: contextual embeddings, NLP, logs, BERT, regression analysis

Supervisor: Ing. Jan Drchal, Ph.D. Karlovo náměstí 13, E-406, Praha 2

Abstrakt

Kontextové reprezentace (embeddingy) se staly důležitou součástí přirozeného zpracování jazyka (NLP), neboť umožňují zachycení informace o daném slově v kontextu celé sekvence textu.

Současně nejmodernější technologie na vytváření kontextových reprezentací budou předmětem studia této práce se zaměřením na analýzu logů. Jedna z těchto technologií - model postavený na architektuře BERT - bude použita k vytvoření kontextových reprezentací logů z datasetu poskytnutého výhradně pro tuto práci. K vyhodnocení kvality těchto reprezentací bude použito několik regresních modelů s cílem predikce určitých hodnot v datasetu - konvoluční síť TCN, Longformer - další model postavený na architektuře BERT a jako výchozí bod kvality byla použita technologie *fastText*. Bohužel, žádný z těchto modelů se nebyl schopen na předložených datech natrénovat.

Detekce anomálií s použitím konvoluční sítě TCN byla zvolena jako jednodušší přístup oproti regresi. Ani tento přístup bohužel nepřinesl lepší výsledky. Použití jiného datasetu by mohlo pomoci odhalit, jestli ten původní byl vhodně vybrán pro vyhodnocení kontextových reprezentací logů.

Klíčová slova: kontextové embeddingy, NLP, logy, BERT, regresní analýza

Překlad názvu: Kontextové reprezentace pro predikce založené souborech logů

Contents

Figures

Tables

Chapter 1

Introduction

In recent years the demand for automation in industry has increased significantly. Companies rely on software solutions to accelerate their processes, an essential part of which is logging. Well-structured logs provide necessary information about the behaviour of the software system itself, but also about the industrial process and are therefore a crucial tool for troubleshooting. As computation resources have become more affordable, simply storing logs as historical data no longer fulfils its potential. The collected data are widely reused in various machine learning tasks such as regression and classification to discover patterns or predict values, which can bring further improvements.

Since logs are the favourite method of holding all required information a developer or a system administrator needs to analyse and solve a potential problem, the memory consumption of generated log files notably increases with the complexity of the system. To reduce the memory consumed by logs, complete sentences are often truncated to contain only keywords or parameters. Nevertheless, despite the absence of proper syntax, logs are still textual structures. When working with texts, word embeddings - numerical representations of words, values of which encode semantic relations between the words - are often used. Recent state-of-the-art machine learning techniques for Natural Language Processing tasks have shown significant improvement in such embeddings creation, now outstandingly handling context of sentences or longer texts.

Whilst previous research on contextual embeddings has focused mainly on existing languages, the aim of this work is to illustrate that embeddings of logs learned by the same techniques are able to encode the context as well. One of these techniques - the BERT architecture - will be described and evaluated on a custom dataset provided for this work.

Since contextual embeddings of log lines have been previously studied and evaluated on anomaly detection models, a more complex approach will be analysed in this work - several regression analysis models used for prediction of certain values in the dataset will be used to evaluate the contextual embeddings created.

Chapter 2

Related work

2.1 Log analysis

This thesis is a part of a larger project focusing on log analysis. It follows the work of Marek Souček [24], and is parallel to the theses of Martin Koryťák [13], which focuses on anomaly detection task with an exhaustive research on autoencoder-based models, and of Prokop Černý [29], who analyses contextual embeddings of logs and evaluates them on anomaly detection experiments. The objective of this work is to extend the pool by leveraging of contextual embeddings in regression analysis. Unlike anomaly detection, regression analysis in log analysis is not widely studied.

2.1.1 Log parsing

In general approach to log analysis, log parsing is the initial step. It is a process of transforming of raw log messages into a sequence of structured events [10]. Traditional approaches rely on regular expressions, however this approach is not ideal as the amount of unique log messages is unlimited. That led to automated log parsing, which are data-driven approaches that are able to learn the patterns from the log data and automatically generate log templates . Such approaches include frequent pattern mining (LogCluster [25]), an online streaming method Spell, which utilizes longest common subsequence computation [7] and parsing tree (Drain [11]). [28]

The step following log parsing is feature extraction, which is a process of encoding information from structured data to numerical vectors that can be later fed into machine learning models. [24]

2.1.2 Log anomaly detection

The task of anomaly detection in logs is a commonly used approach in log analysis. With the volume of log lines generated by systems, automated anomaly detection can possibly save great amount of time.

General anomaly detection methods are either supervised or unsupervised. The supervised method where labels are provided is nothing else than a binary classification. The most common methods are Logistic Regression, Support Vector Machines (SVM) and Decision Tree. Among the unsupervised methods are Log Clustering, PCA and Invariant Mining. [12]

An Online Evolving SVM algorithm is proposed in [9] for log-based anomaly detection. Another approaches to log anomaly detection use Recurrent Neural Networks, such the LSTM network in DeepLog [8], or Bi-LSTM network used in LogRobust tool [27].

2.2 Transformer

The Transformer [26] is a model that utilizes attention mechanism ([1] and [16]) to speed the learning process instead of relying on recurrence. It follows the encoder-decoder architecture, its structure is depicted in Figure 2.1.

The stack of encoders are on the left side, while the the stack of decoder on the right. Before entering an encoder or decoder, the input embeddings pass through a positional encoding operation that addresses the order of the words in a sequence. Each encoder consists a self-attention layer and a feed-forward neural network, both layers are followed by a normalization layer.

The architecture of decoder is very similar apart from an additional attention layer the input of which is the output of the encoder. The outputs of the decoder stack are then transformed to a vector of next-token probabilities with linear layer and a softmax function.

2.3 BERT

BERT (Bidirectional Encoder Representations from Transformers) [6] is the current state-of-the-art model for Natural Language Processing tasks. It derives from multiple technologies such as the Transformer introduced in previous section, ELMo [18] - deep word representations that allow words to be handled differently according to their context, semi-supervised sequence learning [5] and OpenAI Generative Pre-trained Transformer [19].

BERT model derives the encoder-decoder structure from the Transformer. Like ELMo, BERT is a bi-directional model, which means it handles both left-to-right and also right-to-left context, which the OpenAI GPT does not - it is uni-directional. This is done by so-called "masked language modelling" (MLM) where a portion of the input tokens - randomly chosen - are masked and the model is trying to predict the word only by its context. An advantage that BERT derived from OpenAI GPT is that by replacing the encoder stack with the OpenAI Transformer a fine-tunable pre-trained model is created.

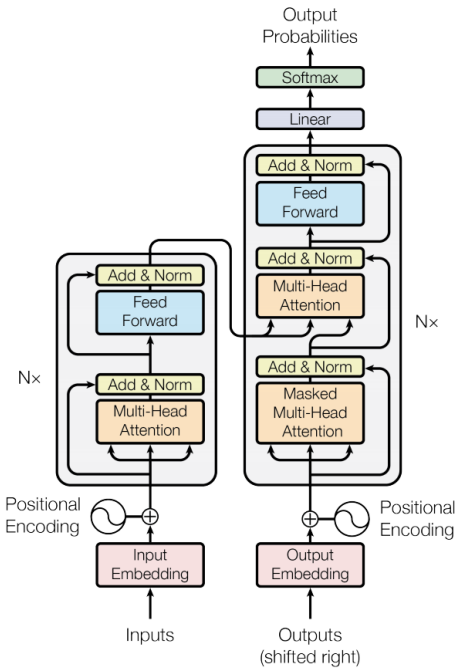


Figure 2.1: The Transformer encoder-decoder structure. Image from [26]

It can be used for variety of tasks such as sentence classification, question answering or only to create contextualized word embeddings.

When using BERT as a model pre-trained for specific tasks, there are several special tokens, that the model expects on the input based on the task selected. The first one is `[CLS]`, which stands for classification and is prepended in front of every input, the `[SEP]` token serves as a separator in two-sentence tasks. For MLM the masked tokens are replaced with the `[MASK]` token.

2.4 Word embeddings

2.4.1 fastText

FastText is one of the most common techniques of word embeddings creation. It was introduced in [4] as an extension to a skip-gram flavour of word2vec algorithm [17] where also subword information is considered.

Each word is split into character groups called n-grams of a specified length n, minimum and maximum of which can be set as a parameter. The resulting substrings are stored in a vocabulary. Characters < and > are added to the beginning and to the end of the word respectively, thus inner parts of words are stored as is and prefixes and suffixes are distinguishable by the special characters.

2.4.2 Contextual embeddings

Simple word embeddings take into account only words as they are, therefore words like "car" and "vehicle" will be embedded as ultimately different words despite having very similar meaning. This led to a new approach to numerical word representation - the contextual embeddings.

ELMo (Embeddings from Language Models) [18] introduces a new approach, that each token is represented by a function of the whole input sequence. ELMo derives from a bi-directional LSTM, thus is able to handle context in both directions.

BERT word representations are deeply bi-directional (see section 2.3), which unlike with ELMo means that each tranformer layer of the BERT model creates a contextualized representation of each token. BERT is the current state-of-the-art in contextual word representation.

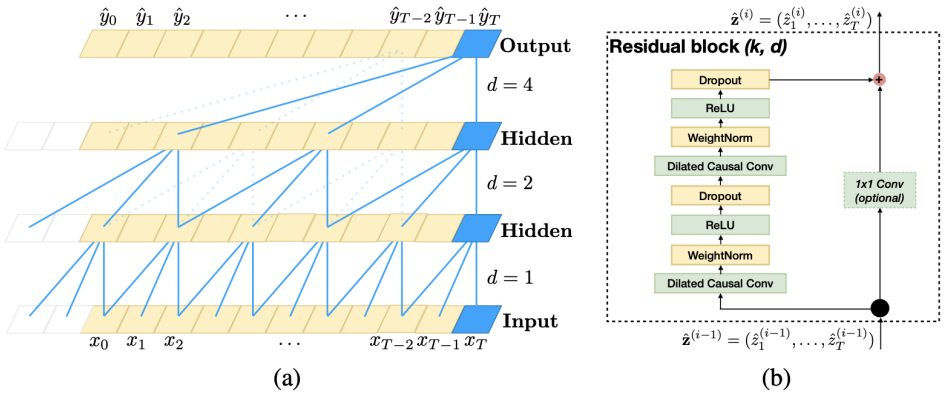


Figure 2.2: Architecture of TCN. Image from [2]

2.5 Temporal Convolutional Network

Temporal Convolutional Network [2] introduces a new approach to sequence-to-sequence modeling. Instead of using recurrent networks a simple convolutional architecture is presented. The key characteristics of the TCN are causality of convolutions, meaning that the direction of information is never from future to the past and that the TCN can map a sequence of an arbitrary length to a sequence of the same length.

TCN is also able to build long effective history sizes by dilated convolution and very deep networks. The dilation convolution with dilation factors $d = 1, 2, 4$ and filter size $k = 3$ is shown in 2.2(a) This way An example of a generic residual block that the TCN consists of is shown in Figure .

Chapter 3

Problem analysis

This chapter proposes the goal of this thesis. Initially, section 3.1 describes the datasets used. With the initial knowledge of the data provided the actual aim of this work is introduced in section 3.2 and section 3.3 analyzes the datasets and their features with respect to the goal stated earlier that are fundamental for further experiments.

3.1 Dataset introduction

Altogether three datasets were provided by Milan Suchý (Trask solutions a.s., ŠKODA AUTO a.s.). The first and the largest one, which will be referred to as the *CAKL* dataset, consists of a collection of logs generated by an internal software system of the company. Data included in this dataset will be the main source in later work. The other two datasets consist of logs generated by Automatic Workload Repository[1], which is a "automatic performance statistics data warehouse" and is a feature of Oracle Database[2]. Each of the AWR datasets provided contains different database statistics. Comparison of sizes of all datasets is displayed in Table 3.1.

Query	Size	No. of log lines
CAKL	20.4 GB	44248555
AWR DB STATS	92.8 MB	693621
AWR QUERY STATS	123.5 MB	211684

Table 3.1: Datasets comparison

[1] `https://www.oracle.com/technetwork/database/manageability/diag-pack-ow09-133950.pdf`

[2] `https://www.oracle.com/database/`

3.1.1 CAKL dataset

The logs included in this dataset were collected within the span of 21 days - from 29th November 2020 until 19th December 2020. The total number of log lines provided is 44248555. Figure 3.1 shows the number of log lines generated each day. For *2020-12-05* and *2020-12-19* the numbers are 6565 and 410 respectively. The periodicity of lower numbers can be attributed to the fact that *2020-11-29* was a Sunday, thus higher numbers occur on workdays.

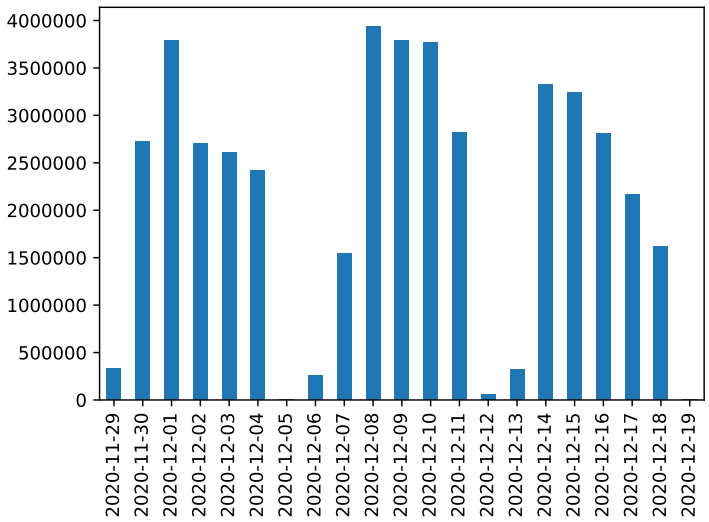


Figure 3.1: Number of log lines per day

The format of each log line of this dataset is the same:

```
<timestamp> <host> <log message>
```

The *timestamp* part contains a timestamp in a human-readable format. The *host* part contains one of two hosts *SKDAMBSCAKL1N01* and *SKDAMBSCAKL1N02*. Finally the log messages is what differs significantly. The only common part is the beginning - each log message starts with a time information in a local timezone. The rest of the raw message can be divided into two groups. The first group which includes the majority of log lines starts with an unexplained number, the log origin - a file with a line number and a path to class method of origin. These are each enclosed in square brackets and followed by a custom log message:

```
[<number>][][<log origin>] <custom message>
```

The second group consists of much fewer lines supposedly called telegrams. These do not contain any readable text and can be recognized by the majority of characters being an asterisk (*). Examples of both groups are shown below:

```
11:32:26.210 [1][][StdClassDB.pas:3128, StdClassDB.TStdClassUniDac.
    StartTransaction] Transaction started

11:32:25.819 000268 : MTU03120405119564***************4147*********
**************************253*0060000000******************B2**B2-
OUT-SCALE********B2SC03216237********B2-R301************B2-R301****
*********B2CV************************************************************
*****************************************I00014093431.0001.0001***
*********************************************************************
*********************************************************************0
000386000000000000000000000000000000000000000000000000000000*****
***************P23869*************************************************
****************************0001000000EE
```

The most important log lines for the purpose of this thesis are members of the first group and contain SQL queries. The custom message of each such log line is of the following format:

```
SQL-Dauer=<time> ms, SQL="<query>", Params=[<params>]
```

The SQL-Dauer value, which translates to "SQL duration", stores information of how long the respective SQL process took in milliseconds. For simplicity, any such value will be later referred to as *SQL-Dauer* value.

3.1.2 AWR datasets

This section introduces both AWR datasets at once. They both serve a supporting purpose.

The first is the *AWR DB STATS* dataset, which consists of statistics of the database metrics measured periodically. The format of each log line is as follows:

```
<timestamp>, SNAP_ID=<snap_id>, END_TIME=<timestamp>, METRIC_NAME=<
    name>, VALUE=<value>
```

The *AWR QUERY STATS* dataset contains information about certain SQL queries that were run on the system. The format of each log line of this dataset is shown below:

```
<timestamp>, SNAP_ID=<snap_id>, END_INTERVAL_TIME=<timestamp>,
    SQL_ID_HASH=<hash>, MODULE=<module file>, EXECUTIONS_DELTA=<delta
    >, BUFFER_GETS_DELTA=<delta>, ROWS_PROCESSED_DELTA=<delta>,
    CPU_TIME_DELTA_MS=<delta>, ELAPSED_TIME_DELTA_MS=<delta>,
    IOWAIT_DELTA_MS=<delta>, PHYSICAL_READ_REQUESTS_DELTA=<delta>,
    PHYSICAL_READ_BYTES_DELTA_KB=<delta>,
    PHYSICAL_WRITE_REQUESTS_DELTA=<delta>,
    PHYSICAL_WRITE_BYTES_DELTA_KB=<delta>, SQLTEXT=<sql_query>
```

Examples of log lines of each dataset are listed below:

```
2020-12-19 23:59:42.000, SNAP_ID="75762", END_TIME="2020-12-19
    23:59:42.0", METRIC_NAME="Host CPU Usage Per Sec", VALUE
    ="524.429261789702"

2020-12-19 23:45:14.825, SNAP_ID="75761", END_INTERVAL_TIME
    ="2020-12-19 23:45:14.825", SQL_ID_HASH="450f0g08ssct3_1287658384
    ", MODULE="PMS_WMS_ALLOC.exe", EXECUTIONS_DELTA="22500",
    BUFFER_GETS_DELTA="33660", ROWS_PROCESSED_DELTA="22500",
    CPU_TIME_DELTA_MS="317.562", ELAPSED_TIME_DELTA_MS="775.236",
    IOWAIT_DELTA_MS="0", PHYSICAL_READ_REQUESTS_DELTA="0",
    PHYSICAL_READ_BYTES_DELTA_KB="0", PHYSICAL_WRITE_REQUESTS_DELTA
    ="0", PHYSICAL_WRITE_BYTES_DELTA_KB="0", SQLTEXT="SELECT COUNT(*)
     FROM tblWMS_ARTICLES WHERE WART_IDENT = :AArticleIdent"
```

3.2 Problem statement

Previous section introduced the CAKL dataset and the log lines containing SQL queries. The general objective of this work is a regression analysis that would yield and hopefully be able to predict the individual *SQL-Dauer* values based on previous log lines.

3.3 Exploratory analysis

In this section a thorough analysis of all three datasets with respect to the *SQL-Dauer* value will be provided.

3.3.1 CAKL dataset

Initially, Figure 3.2 illustrates the daily distribution of *SQL-Dauer* value in the whole dataset. Clearly the amount corresponds to the overall number of log lines in the dataset. The total amount of all *SQL-Dauer* values in the dataset is *3397650*.

There are altogether *114* different queries present - *BEGIN*, *SELECT*, *UPDATE*, *INSERT* and also *DELETE* statements are represented. Table 3.2 contains concrete figures of frequency of each statement category.

For simplicity only *SELECT* statements were chosen to be analyzed. All log lines containing a particular query are considered identical even though the parameters of the query differ. The most frequently occurred queries over the whole dataset with respective amounts are shown in Figure 3.3 for illustration.

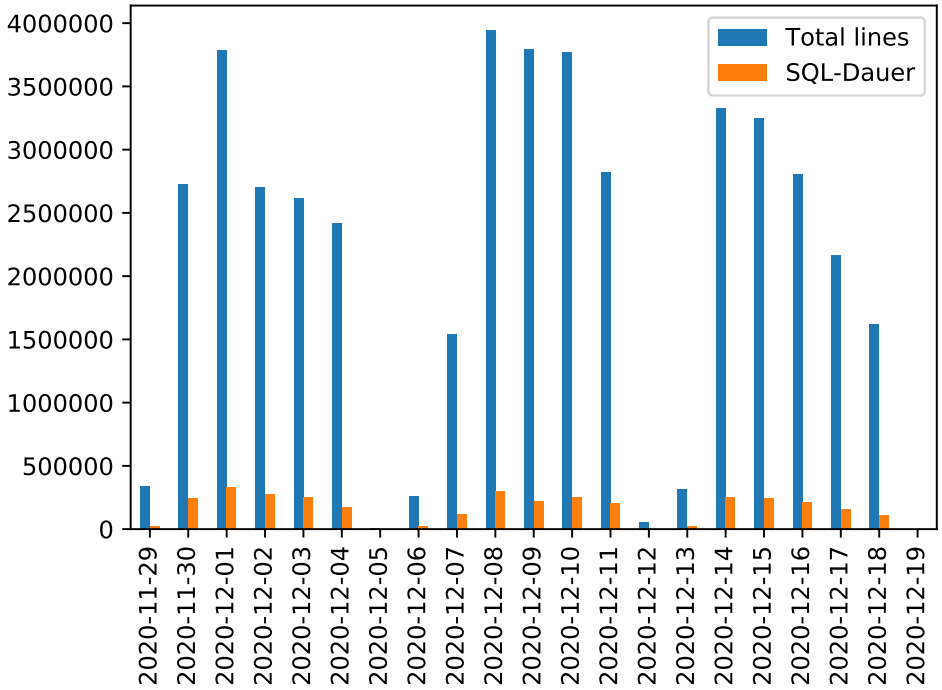


Figure 3.2: Number of *SQL-Dauer* values per day

Query	No. of queries	No. of log lines
SELECT	57	2894683
UPDATE	46	317458
INSERT	9	185505
DELETE	1	2
BEGIN	1	2

Table 3.2: Number of log lines per query category

	Query	Lines
a	`select * from tblWMS_ARTICLES where WART_NUMBER = :refWART_NUMBER`	520555
b	`SELECT * FROM TBLWMS_TASK_ITEMS WHERE WTIT_TASK = :TaskItem ORDER BY WTIT_TASK, WTIT_POS`	496802
c	`SELECT * FROM TBLWMS_TASK_SPLITS WHERE WTSP_TASK = :TaskItem ORDER BY WTSP_TASK_POS, WTSP_POS_SEQ`	496796
d	`SELECT lu.*, luty.*, artl.* FROM tblWMS_LU lu INNER JOIN tblWMS_LU_TYPES luty ON (lu.WLOU_TYPE_IDENT = luty.WLTY_IDENT) LEFT JOIN tblWMS_ARTICLES artl ON (lu.WLOU_ARTL_IDENT = artl.WART_IDENT) WHERE WLOU_HU = :AParentIdent ORDER BY WLOU_IDENT, WLOU_MASTER, WLOU_GRID_POS`	237292
e	`SELECT * FROM TBLWMS_TASKS WHERE WTAS_IDENT= :par_WTAS_IDENT`	233153

Table 3.3: Number of log lines per query

The top three statements were selected to be further studied. They will be later referred to as *Query 1*, *Query 2* and *Query 3* in the same order as in the table. Obviously the second and third statements are very similar, this fact is also supported by the respective numbers of lines.

Figure 3.3 shows the daily distribution of each of the three statements throughout the dataset. Apparently, calling of *Query 2* and *Query 3* must always occur simultaneously as their daily frequencies appear to be almost identical. The distributions correspond to the overall distribution of the SQL queries in the dataset, thus apart from the weekend days with general lack of data, other parts of the dataset seem suitable for experiments.

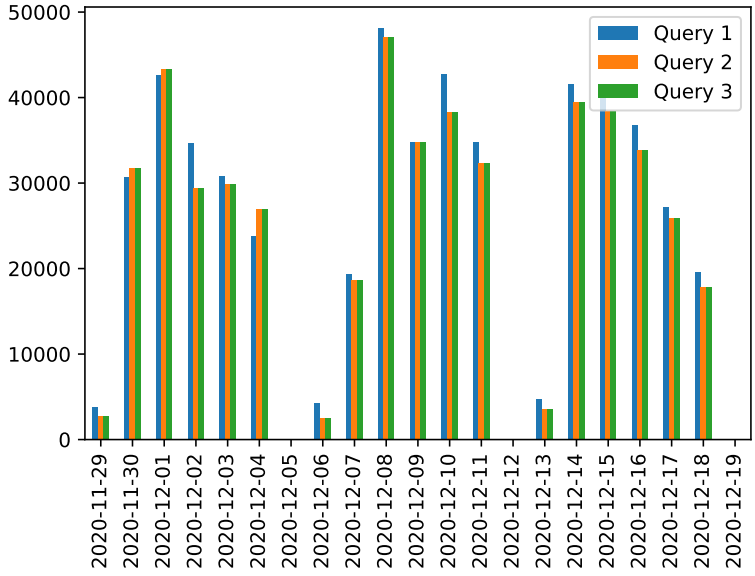


Figure 3.3: Distribution of *Query 1*, *Query 2* and *Query 3* over the dataset, the y-axis represents the total number of occurrences of each query per day

Apart from the frequencies of each of these queries shown in Table 3.3 it is crucial to explore individual *SQL-Dauer* values. Not only the range of values were inspected, but also the count of lines between each log line containing the same SQL query. Each column in the histograms in Figure 3.4 represents one of the three queries. Histograms in the first row show the distribution of the *SQL-Dauer* values over the dataset. The x-axis represents individual *SQL-Dauer* values of the respective query and y-axis represents the total number of occurrences of each value. The second row histograms illustrate the distribution of the number of log lines between each *SQL-Dauer* occurrence, again individual amounts of log lines in-between are on the x-axis, while the total amount of the each number of lines is represented by the y-axis. Finally, the last row represents a relationship between the two.

Another feature worth evaluation is a distribution in one day. The day of 30/11/20 was chosen as it is the first workday, thus containing a reasonable amount of log lines. The distribution is shown in Figure 3.4. As this date will

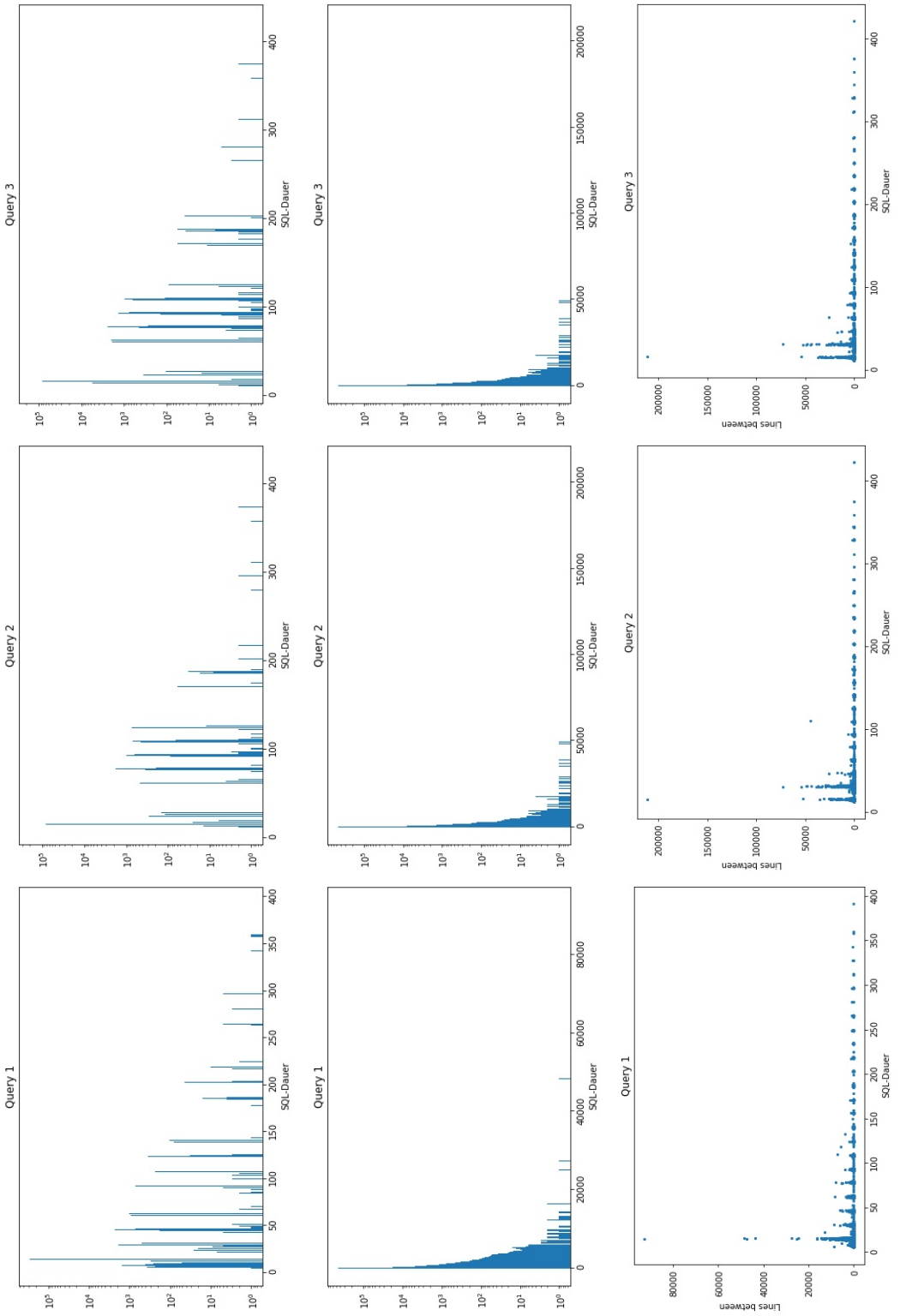


Table 3.4: Number of occurrences of *SQL-Dauer* value; number of log lines in-between each query occurrence and their relationship

have been already analyzed, this time span will be later used in experiments as a 24 hour sample when needed.

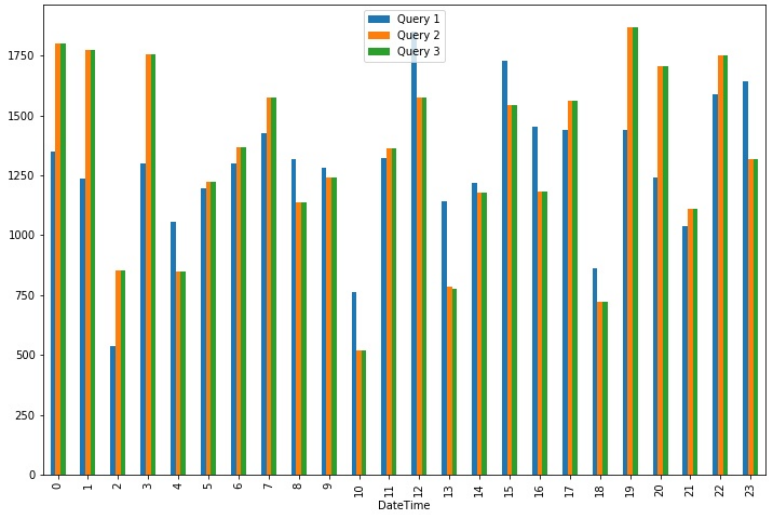


Figure 3.4: Number of *SQL-Dauer* values of *Query 1*, *Query 2*, *Query 3* per day

The last feature studied was unfortunately forgotten in the initial work on this section, which might have implications on the course of this thesis. The feature is the distribution of the actual value of the *SQL-Dauer* variable. Histograms were created for each of the *SELECT* and *UPDATE* query containing more than 1000 such values. To capture as much information possible, two histograms were generated for each query. Examples in Figure 3.5 were chosen to illustrate certain patterns. Each row consists of two histograms generated for one specific query. While the first column contains histograms generated with the whole dataset, the second one illustrates a shorter portion of the data - only one day - again of 30/11/20.

The first query represents those where the *SQL-Dauer* values form at least two groups, in this case the most frequent SQL process duration values are `15ms` and `30ms`. This leads to a thought that rather than predicting a concrete *SQL-Dauer* value, it might be easier to predict a category which the value falls in.

The second row demonstrates a category where a portion of the data is missing. The query in the second row could be used for experiments only when a shorter dataset is considered such as the day of 30/11/20. In both this and the third row, the *SQL-Dauer* values form two groups, one containing almost exclusively a value of 15 milliseconds (with several 14ms and 16ms) and the other one consisting of different other values - anomalies. Queries in this category would therefore be suitable for classification experiments, in this case anomaly detection.

This criterion is not met by the *Query 2* and *Query 3*, these form at least three such categories; therefore, different queries had to be selected. Apart from this feature, a number of *SQL-Dauer* values was also taken into

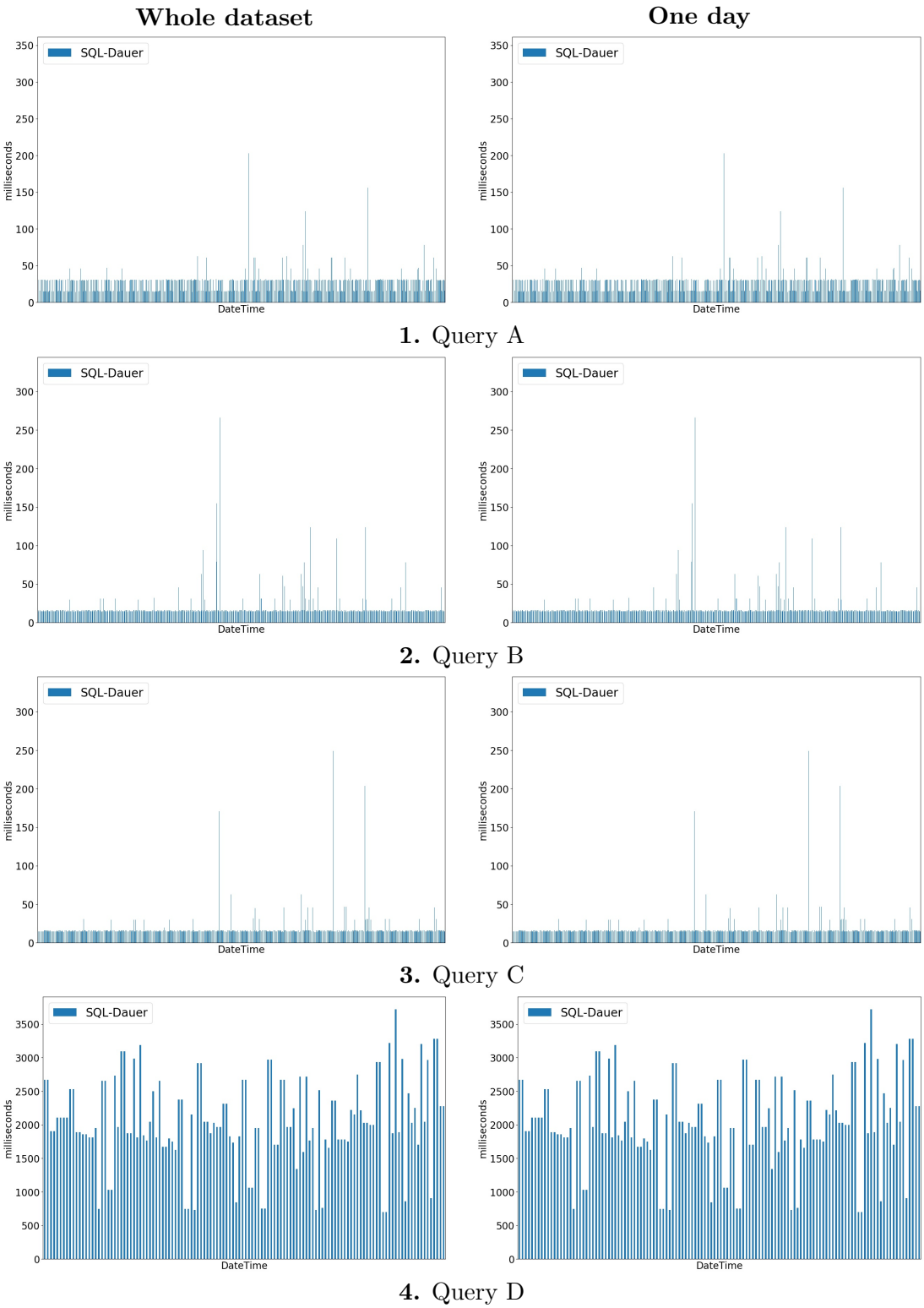


Figure 3.5: Distribution of *SQL-Dauer* values in time of four chosen queries - throughout the whole dataset in the left column and throughout the day of 30/11/20 in the right column; the x-axis represents a time of the occurrences and the y-axis the actual *SQL-Dauer* values

account. Thus *Query 1*, which is the same as the one used previously, hence the identical label, remains. Two other queries presented in Figure 3.5 with their respective number of log lines, which will be referred to as *Query 4* and *Query 5*, were selected.

	Query	Lines
1	`select * from tblWMS_ARTICLES where WART_NUMBER = :refWART_NUMBER`	520555
4	`SELECT * FROM tblWMS_TASK_EXT WHERE WTAE_TASK = :ATaskIdent`	204965
5	`SELECT * FROM TBLWMS_TASK_SPLITS WHERE WTSP_IDENT= :par_WTSP_IDENT`	197624

Table 3.5: Queries selected for anomaly detection

Table 3.6 illustrates number of anomalies of each query with a percentage in the data of the day 30/11/20 and certain time spans, all beginning at midnight at ending after the number of hours in the first column

	Time interval	Log lines	Anomalies	Percentage
Query 1	24 hrs	30718	1751	5,7%
	12 hrs	14082	698	4,96%
	4 hrs	4424	162	3,7%
	1 hr	1348	25	1,9%
Query 4	24 hrs	17785	1298	7,3%
	12 hrs	8153	466	5,7%
	4 hrs	2079	126	6,0%
	1 hr	840	35	4,2%
Query 5	24 hrs	15369	1025	6,7%
	12 hrs	7314	399	5,5%
	4 hrs	2844	81	2,8%
	1 hr	836	8	0,96%

Table 3.6: Number and percentage of anomalies of each query per time spans of 30/12/20; all time spans begin at midnight

The *SQL-Dauer* values in the last row of Figure 3.5 are more randomly distributed than in the previous cases. Such queries would be more appropriate for prediction of specific *SQL-Dauer* values. Unfortunately none of the three queries selected at the beginning of this chapter belong to this group. Actually, there are only two queries that do, altogether with 3084 and 3358 *SQL-Dauer* values over the whole dataset - these will be later referred to as *Query 6* and *Query 7*.

3.3.2 AWR datasets

Initially, the *AWR DB STATS* dataset includes periodical statistics of the database. There are 23 unique log lines generated every minute over the entire dataset.

The second *AWR QUERY STATS* dataset consists of statistics of certain queries called throughout the whole time interval. As there are only 211684 values compared to 520555 of the *Query 1* alone, only a small segment of queries is actually included in this dataset. In fact, for example *Query 1* is not present at all. Figure 3.6 shows the distribution of both AWR datasets. It is clear that in contrast to the dynamically generated data in the CAKL dataset, the AWR statistical data frequency remains almost constant each day. Indeed the minimal frequency of the query statistics is 6772 log lines on 12th December, while the maximum is 9331 on 7th December.

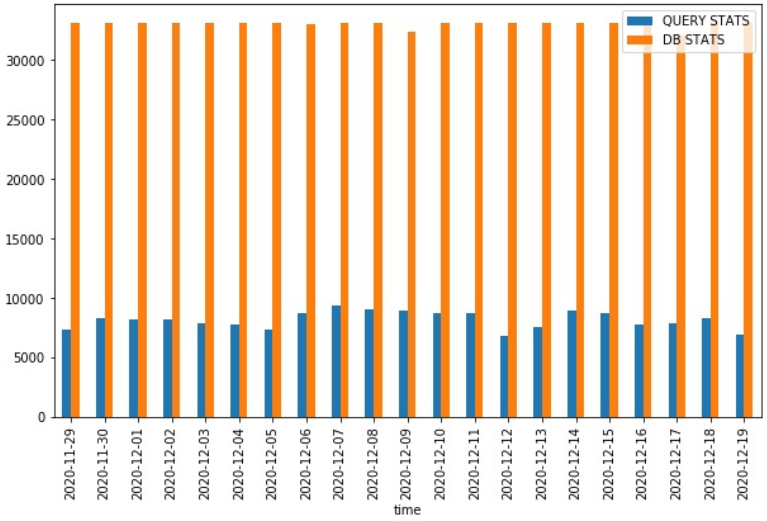


Figure 3.6: Distribution of AWR datasets

For some experiments all three datasets will be combined after sorting the timestamps of log lines.

Chapter 4

Methodology

The goal of this chapter is to describe the approaches and architectures used throughout the work in order to reach the goal stated in section 3.2, which a regression analysis for prediction of *SQL-Dauer* values of individual queries based on previous log lines. Initially the log lines need to be transformed into a numerical representation suitable for training. For this task a BERT architecture will be used as it is currently the state of the art language model for Natural Language Processing tasks.

4.1 Log preprocessing

A general approach used in most works focusing on log analysis is a framework of *log collection*, *log parsing* and *feature extraction* introduced in [12].

As mentioned in section 3.1, the datasets for this work were provided, thus the log collection is done. There are two approaches to log parsing in this work. In the first one each log line in the CAKL dataset is split into three parts - the *timestamp*, the *host* and the raw *log message* and only the raw log message part is then transformed into embeddings. In the second approach only timestamps are extracted from the log lines from both CAKL and AWR datasets, but they serve only for sorting and combination of the datasets.

Considering feature extractions as a process of conversion of text to a set of features, than it is the contextual embeddings creation in this work.

4.2 Contextual embeddings

4.2.1 Tokenization

Before being fed into a model, a raw text must be first tokenized, i.e. split into tokens, which are groups of characters that form the original text. Each token can be a single character, a whole word or a subword - e.g. a root

or a prefix of a word. A new vocabulary of a tokenizer can be build from scratch with a custom dataset, which is especially useful when a pre-trained model of the dataset language does not yet exist. Even though the language of the datasets provided for the experiments could not be considered proper English, building of a new tokenizer is out the scope of this thesis. Thus already existing tokenizers with pre-trained models will be used.

There are three different tokenizers used in the Transformers[1] library that will be introduced in section 5.1 - Byte-Pair Encoding [23], WordPiece [22], and SentencePiece [14]. Each of them is used for different models. Related to this work are the WordPiece tokenizer for *BERT* model [6], and also its distilled version *DistilBERT* model [21], and the Byte-Pair Encoding tokenizer used for *RoBERTa* [15].

Figure 4.1 demonstrates a shortened example of a tokenized log line with the DistilBERT model which uses WordPiece. The total number of tokens generated from this particular line is 112.

```
11:32:20.663 [2][][StdClassDB.pas:0902,StdClassDB.TStdClassCommonDAC.
TimeLog] SQL-Dauer=000030 ms, SQL="SELECT * FROM TBLWMS_TASK_SPLITS
WHERE WTSP_TASK = :TaskItem ORDER BY WTSP_TASK_POS, WTSP_POS_SEQ ",
Params=["I00014094666"]

['11',':','32',':','20','.','66','##3','[','2',']','[',']','[','st',
'##dc','##lass','##db','.','pas',':','09','##0','##2',',',',','st','##dc
','##lass','##db','.','ts','##t','##dc','##lass','##com','##mond','#
#ac','.','time','##log',']','sql','-','da','##uer','=','000','##0',
'##30','ms',',',',','sql','=','"','select','*','from','tb','##l','##w',
'##ms',...]
```

Figure 4.1: Tokenization example

Apparently, words such as `time` and `select` are present in the tokenizer's vocabulary, but the word `dauer` is not, therefore it is split into a word `da` from the vocabulary and a token `##uer`, where the two `##` denote that it is a suffix and should be attached to the previous token.

4.2.2 Embeddings creation

After tokenization of log lines, the next step is the contextual embeddings creation. The model chosen for this specific task is called the *DistilBERT*. While the general aspects stay the same as the architecture of BERT, it is a smaller, faster, cheaper and lighter - or *distilled* - version. To achieve this, the token-type embeddings and the pooler were removed, and the number of layers was reduced by a factor of 2. Also, most of the operations used in the Transformer architecture (linear layer and layer normalisation) were highly optimized in modern linear algebra frameworks. As a result, the DistilBERT

[1] `https://huggingface.co/transformers/index.html`

has 40% fewer parameters than the general *bert-base-uncased*[2] model, runs 60% faster while preserving over 95% of BERT's performances as measured on the GLUE language understanding benchmark. [21] This model is particularly beneficial when computational resources are shared among researchers and it is desired to reduce the time spent on training.

Again the model for embeddings creation can be trained from scratch, but for simplicity a pre-trained model will be used. Regarding that, further enhancement to the BERT architecture was adopted - the *Sentence-BERT*[3] (SBERT). It is a modification of BERT that improves its ability to derive semantically meaningful sentence embeddings by adding a pooling operation to the output of BERT, which enables derivation of fixed-sized embeddings. Semantically meaningful is understood as the fact that semantically similar sentences are close in vector space. To achieve that the BERT model must compare each pair of sentences, which becomes computationally too complex. By using a *twin* network architecture, which means that two identical networks are used for the two sentences in the pair and the outputs are then compared by a specific objective function.[20] The default `MEAN` pooling strategy was applied in this case, the output of which are the desired embeddings.

4.3 Regression analysis

There are three approaches chosen for regression analysis. In section 4.3.1 the Temporal Convolutional Network is described. It is the only model that will be fed with the contextual embeddings created as introduced in 4.2.2. The second approach is a BERT model called *Longformer* [3], which will be used both for embeddings creation and for regression analysis. In the last approach neither embeddings creation nor regression analysis is based on any BERT architecture.

4.3.1 Temporal Convolutional Network

A sequence of log lines previous to the log line containing an SQL query is used to predict the value individual *SQL-Dauer* value. Each sequence of lines forms a *window* that is then fed into a neural network. Windows can partially overlap as there is no minimal distance between log lines containing identical queries. Temporal Convolutional Network (TCN) architecture is chosen for this task as it is proven to outperform architectures such as LSTM on a variery of tasks and datasets [2]. The windows fed into the TCN will consist of contextual embeddings of individual log lines.

[2] `https://huggingface.co/bert-base-uncased`
[3] `https://sbert.net/index.html`

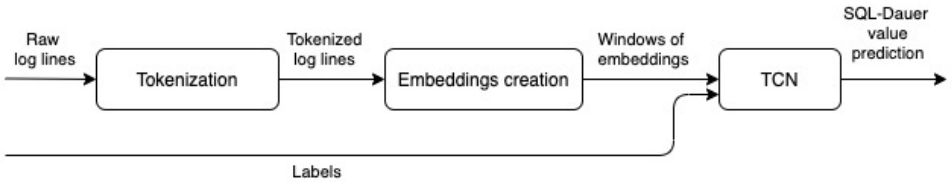


Figure 4.2: Data flow

4.3.2 Longformer

The BERT models are trained for tasks such as sentence classification, translation or question answering. The input of all of these tasks is usually a few sentences long and for one input, there is also one output - a class of a sentence or an answer to a question. Even in tasks such as sentence similarity the two strings are concatenated with tokens in-between mentioned in 2.

In this case, all log lines in a selected window act as a single input, therefore they are chained into one string and fed into the network. As mentioned in Section 5.1, the maximum length of the input sequence the *DistilBERT* model can handle is 512. However, as Figure 4.1 demonstrated, such length is not sufficient even for a window of 10 log lines. To overcome such obstacle it was necessary to find a different architecture - the *Longformer*. *Longformer* is also a BERT architecture with the ability to manage long input sequences. The original transformer-based architectures are not suitable for such tasks as the attention mechanism memory consumption scales quadratically with the input length. The *Longformer* introduces an improvement in the self-attention operation the memory consumption of which is linear.[3]

The most significant difference between *Longformer* and the model used in Section 5.1 is that the *Longformer* is derived from *RoBERTa* model and not the *DistilBERT*. *RoBERTa* or the robustly optimized BERT, the size of which is the same as the original *BERT*, but it is trained on 10 times more data and outperforms the original *BERT*.[15]

The *Longformer* model will thus be used for both embeddings creation and regression analysis.

4.3.3 fastText

Creating embeddings with fastText unsupervised model is chosen as a baseline approach to BERT architecture since it is a commonly used tool in log analysis and is verified to yield interesting results [24]. Unlike BERT tokenizers introduced in section 4.2.1, fastText splits text into n-grams, where n represents number of characters The regression analysis model will be based on the LSTM recurrent neural network architecture.

4.3.4 Loss function

The most common regression loss function will used all regression models in this work, its formula is shown in equation 4.1, where n represents a size of a test set, y_i is the observed/actual value and $\hat{y}_i$ is the predicted value.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{4.1}$$

4.4 Anomaly detection

4.4.1 Motivation

Despite the overall aim of this thesis was a regression analysis, it is demonstrated in section 6, that it did not yield satisfactory outcomes. Therefore it was decided to scale down in terms of difficulty and instead of attempting to predict a certain real value, only anomalies will be detected instead.

4.4.2 Data preparation

The embeddings - or the input - created for value prediction can be reused in this approach, however the output values need to be updated. As already described in section 3.3.1, there are queries where the value of *SQL-Dauer* variable visually form two categories - *normal* and *anomalous*. The normal category contains a value of 15ms, while the rest is considered anomalous. Since the visual separation of categories is not precise enough, a further analysis of the values is done to properly distinguish them. Values 14 and 16 will also be considered *normal* because their frequency is too high to label them anomalous. Several events occurred throughout the time span of the whole dataset, each lasting from a few minutes to a couple of hours, when the *SQL-Dauer* values gradually decrease below 15 ms, at most up to 6ms, and then return back to the *normal* values. Such events will not be regarded anomalous. To simplify the experiments all windows with the output value less than 14 are skipped.

4.4.3 Loss function

For anomaly detection, the Temporal Convolutional Network architecture described in section 4.3.1 with the only difference, that the loss function used will be the *binary cross entropy function* which is the default loss function for binary classification. Its formula is illustrated in equation 4.2, where p is a target distribution and q is an approximation of the target distribution,

thus q_i is the prediction of one output value and p_i is the actual value.

$$BCE(p, q) = -\sum_{i} q_i \log p_i \tag{4.2}$$

Chapter 5

Implementation

In this chapter a detailed description of implementation will be provided. Initially the generation of contextual embeddings itself will be described, while Section 5.2.1 and Section 5.2.2 focus on different approaches to evaluation of the embeddings.

5.1 Embeddings

This section focuses on the implementation of the contextual embeddings creation. As explained in Section 4, the Transformer-based technique called BERT was applied. In all steps of the process the *Transformers*[1] by *Hugging Face* and the native *PyTorch*[2] libraries were used.

Due to a significant memory consumption of the dataset, the data is first loaded into CPU and later processed by individual batches on GPU. As the resulting embeddings will be analyzed and evaluated on a regression task with the objective of predicting the *SQL-Dauer* values, a *window* of 100 log lines preceding each value are included. For each query, those lines of which the number of log lines between respective *SQL-Dauer* value and the previous one seems anomalously too high are skipped. Each batch consists of 10 of such windows.

The log lines in each batch are then tokenized, i.e. divided into string tokens. To obtain the tokens, there is a specific tokenizer class designated for this task in the library. For BERT models the *BertTokenizer* class was implemented. This class inherits from a more abstract PreTrainedTokenizer. According to the official documentation[3], the PreTrainedTokenizer handles raw input data tokenization by splitting log lines into token strings, converting token strings to ids and back, and encoding/decoding (i.e., tokenizing and converting to integers). The resulting tokens are then trained by a BERT

[1] `https://huggingface.co/transformers/index.html`

[2] `https://pytorch.org`

[3] `https://huggingface.co/transformers/main_classes/tokenizer.html#transformers.PreTrainedTokenizer`

model, and a pooling operation is applied on the output that yields the desired fixed-sized embeddings.

Because pre-trained models are used with both tokenizers and embeddings creation, the *AutoModel* and *AutoTokenizer* classes were used instead, since they are able to reliably select the desired BERT architecture according to the pre-trained model used.

As for the tokenizer, the *DistilBERTTokenizerFast* class was selected. The *Fast* version of every Tokenizer class within the library is a Python wrapper of the *tokenizers* library implemented in Rust, which as the title suggests reduces the computational time even further. Since the input sequence length of the *DistilBERT* model can not handle sequences longer than 512 tokens, each log line is embedded separately. It was decided to use each raw log message as a whole so that it contains the maximum information possible. Any line longer than the maximum size is automatically truncated by the algorithm during the preprocessing phase.

As the SBERT is built on top of BERT, it is possible to create the sentence embeddings using only the libraries mentioned earlier. The SBERT part is done by loading the appropriate pre-trained model - the *distilbert-base-nli-stsb-mean-tokens*[4] model provided in the *sentence-transformers* framework.

All embeddings in the PyTorch tensor format are then concatenated and saved to a file to be later analyzed.

5.2 Value prediction

5.2.1 Temporal Convolution Network

Firstly the data needs to be loaded into memory. The GPU, on which the training is run, is provided with significantly limited resources, thus the whole data is loaded into CPU first, and later during the training it is processed by individual batches in GPU. For each *SQL-Dauer* value a window of n embedded log lines prior is selected - the exact number can be selected as a parameter. All data is stored as PyTorch 3-D *tensors*[5]. As it was described in the previous section, the dimension of one embedded log line is 768 and the shape of the respective tensor is then `[1,1,768]`. Considering the *window size* used is 20 and so is the sequence length, the final input tensor shape is `[1,20,768]`.

The next step is to standardize both the inputs and predicted values with the *standard score* method shown in Figure 5.1

$$z = \frac{x - \mu}{\sigma}, \tag{5.1}$$

[4] `https://huggingface.co/sentence-transformers/distilbert-base-nli-stsb-mean-tokens`
[5] `https://pytorch.org/docs/stable/tensors.html`

where x is the value of a tensor element, μ is the mean of the data and σ represents the standard deviation of the data. The standardization is computed for each data feature individually. The dataset is then split into training and validation part. Each window is then randomly placed in either group according to the selected ratio. The data preparation is thus finished.

Layer (type)	Output Shape	Param #
Conv1d-1	[1, 768, 21]	1,180,416
Chomp1d-2	[1, 768, 20]	0
ReLU-3	[1, 768, 20]	0
Conv1d-4	[1, 768, 21]	1,180,416
Chomp1d-5	[1, 768, 20]	0
ReLU-6	[1, 768, 20]	0
TemporalBlock-7	[1, 768, 20]	0
Conv1d-8	[1, 768, 22]	1,180,416
Chomp1d-9	[1, 768, 20]	0
ReLU-10	[1, 768, 20]	0
Conv1d-11	[1, 768, 22]	1,180,416
Chomp1d-12	[1, 768, 20]	0
ReLU-13	[1, 768, 20]	0
TemporalBlock-14	[1, 768, 20]	0
TemporalConvNet-15	[1, 768, 20]	0
Linear-16	[1, 1]	769
TCNModel-17	[1, 1]	0

Figure 5.1: TCN tensor shape

An already existing implementation of TCN architecture by *locuslab*[6] was used with several adjustments. The network consists of two layers. Before entering the first layer of TCN network, the tensor axes are permuted according to *(0,2,1)*-ordering, hence the tensor shape is transposed to `[1,768,20]`. Figure 5.1 illustrates the output tensor shapes after each step of the algorithm. Each layer of the TCN network (called *TemporalBlock* in the figure) consists of 3 steps, repeated twice. Initially a 1D convolution is applied to each tensor, which pads the tensor from both sides. To ensure causality, the *Chomp1d* function removes relevant elements from the right. Finally, the *ReLU* activation function is used.

Since one of the TCN principles is mapping a sequence of an arbitrary length to a sequence of the same length, a linear layer was added as the output layer to obtain a classification model. An overview of the model is illustrated in Figure 5.2.

Several parameters needed to be set for training. The size of the kernel

[6] `https://github.com/locuslab/TCN`

```
TCNModel(
  (tcn): TemporalConvNet(
    (network): Sequential(
      (0): TemporalBlock(
        (net): Sequential(
          (0): Conv1d(768, 768, kernel_size=(2,), stride=(1,),
          padding=(1,))
          (1): Chomp1d()
          (2): ReLU()
          (3): Conv1d(768, 768, kernel_size=(2,), stride=(1,),
          padding=(1,))
          (4): Chomp1d()
          (5): ReLU()
        )
      )
      (1): TemporalBlock(
        (net): Sequential(
          (0): Conv1d(768, 768, kernel_size=(2,), stride=(1,),
          padding=(2,), dilation=(2,))
          (1): Chomp1d()
          (2): ReLU()
          (3): Conv1d(768, 768, kernel_size=(2,), stride=(1,),
          padding=(2,), dilation=(2,))
          (4): Chomp1d()
          (5): ReLU()
        )
      )
    )
  )
  (linear): Linear(in_features=768, out_features=1, bias=True)
)
```

Figure 5.2: Summary of TCN structure

was set to 2, while dropout rate was 0.2.

5.2.2 Longformer

In this approach the contextual embeddings creation and *SQL-Dauer* value prediction is implemented as one process. Similarly to the Section 5.1 the *Transformers* by *Hugging Face* and the native *PyTorch* libraries were used.

First of all the data is loaded into memory, however only certain number of log lines prior to each *SQL-Dauer* value are joined in a string and stored in an array. The corresponding target values - or *labels* - are saved in another. These labels are then standardized the same way as in the Temporal Convolutional Network demonstrated in equation 5.1. The resulting arrays are randomly divided into a training set and a validation set.

As the *Hugging Face* documentation of *Longformer*[7] states the *Longformer* model is still a work in progress, nevertheless it is the easiest BERT approach for long sequences of the few available.

To simplify the process, a pre-trained *Longformer* model was used - the *allenai/longformer-base-4096*[8]. According to its documentation site it is "a BERT-like model started from the RoBERTa checkpoint and pre-trained for MLM on long documents. It supports sequences of length up to 4,096 (tokens)". Usage of *Longformer* is the same as of other models included in the *Transformers* library by *Hugging Face*. Similarly for tokenization a *LongformerTokenizerFast*[9] is used, which is again a faster implementation of the original *LongformerTokenizer* class. The training and validation part are tokenized separately and the output is then fed into a *Trainer*[10] alongside training parameters and a model. Again the pre-trained model *allenai/longformer-base-4096* is chosen and handled by a *LongformerForSequenceClassification* class. Another parameter set is the number of label which in this case is 1, since the target value is an integer. The last step is then fine-tuning the model.

The *Transformers* library itself deals with CPU/GPU usage, therefore it is not necessary to manually manage data loading and training memory consumption differences.

5.2.3 FastText

This approach differs from the previous ones in terms of both embeddings creation and value prediction. The reason it was chosen is that no *Transformer*-based architecture is used.

To create the embeddings the *fastText*[11] library and its text representation feature is used instead. An existing implementation[12] of library function wrappers as well as a *Model* and a *Trainer* classes used later for value prediction.

Initially a new model is trained in unsupervised mode using a log dataset as an input. Windows of log lines preceding a *SQL-Dauer* value of a specific query are then fed into the model to create embeddings. The embeddings are concatenated with time delta values, which is a time difference between each log line and the one prior. This feature adds additional information to the embeddings about the relationship between the logged processes. Also labels are saved into a separate file. After the preprocessing part the next step is the training itself.

[7] https://huggingface.co/transformers/model_doc/longformer.html
[8] https://huggingface.co/allenai/longformer-base-4096
[9] https://huggingface.co/transformers/model_doc/longformer.html#transformers.LongformerTokenizerFast
[10] https://huggingface.co/transformers/main_classes/trainer.html
[11] https://fasttext.cc/
[12] https://github.com/LogAnalysisTeam/ml4logs/tree/master/src/ml4logs

The *Seq2SeqModel* and *Seq2Se2ModelTrainer* were used. Initially a multi-layer long short-term memory (LSTM) regular neural network is applied to the input. It is followed by a linear transformation and a leaky ReLU activation function. Then a combination of dense layer and again a leaky ReLU activation function is applied for each hidden layer. Finally, another linear transformation is added as the output layer. Mean squared error is used as the model estimator.

5.3 Anomaly detection

The same Transformer-based approach - *TCN model* was used for anomaly detection as for the regression analysis with the difference in loss function. Since the anomaly detection is actually a binary classification, the *binary cross-entropy* loss function was chosen. Items in the *anomaly* category will be represented by number 1, while the *normal* windows are set to 0.

Unfortunately, during the late stages of experiments, the linear activation function of the output layer was changed to *sigmoid* function which is much more suitable for binary classification tasks.

Chapter 6

Experiments and evaluation

In this section the outcomes of experiments will be thoroughly described. Initially, section 6.1 evaluates the contextual embeddings creation. Section 6.2 describes different approaches to value prediction and their results and finally section 6.3 evaluates the problem of anomaly detection.

The experiments were run on the computational cluster of the Research Center for Informatics[1].

6.1 Contextual embeddings

A set of 100 consecutive log lines was chosen to help visually illustrate the embeddings created. All log messages are mentioned exactly as in the dataset with possible typos. As it was analyzed in section 3.1 the formats of the custom log messages cannot be easily divided into a small number of categories. However there are certain patterns that can be observed. In this sample, fourteen different categories were distinguished. For later reference, five of them consist of log lines identical to the category name. Those are:

1. `Partially loaded task will load all data now.  this is SLOW!`
2. `Bearbeite Telegramm mit Satzart "0002" von SourceProcess "PMS_COM_MQ"`
3. `Added data to list at position [0000]`
4. `No destinationsolver-bo configured, using default`
5. `Transaction started`

Another four contain a placeholder * for specific parameters, that can be different for each log line:

6. `Next subsystem for moving with split <*> from <*> to <*> is <*>`
7. `Booking finished.  Splitstate :  <*>.  LUState :  <*>`

[1] http://rci.cvut.cz/

8. `Found reservations to resolve for HU <*>`
9. `Received BO <*> for Destination <*>`

The *IPC_DB* category consists of 10 identical lines with the content of `IPC_DB->Data successful inserted` and one line `IPC_DB->Record update successfull`, due to the visual similarity it was assigned the same category.

The *Telegram*, and *SQL-Dauer* categories have already been described in section 3.1.

The last two categories are rather mutually interlinked and the separation is not unambiguous. One of them is the category *Processing*, which consists of several slightly different log lines:

- `Start processing telegram TT02 for current number 297693`
- `Start processing Telegram :  0002`
- `End processing Telegram :  0002`
- `Finished processing telegram <>`

Nevertheless each of them include the words *processing* and *telegram*.

The last category is simply referred to as *Other*. It contains the rest of log lines, that do not match any of the previous categories. There is a minimal overlap with the *Processing* category in log lines such as `Start processing <PrjClassWmsHostHandlerLogis.TPrjClassWmsHostHandlerLogis.SendHostRequestTelegram>`, but the lack of the word *telegram* and the length of the parameter is assumed to be long enough for the embedding to differ from the `Processing` category.

A *t-SNE* technique for dimensional reduction was used to visualize the computed embeddings and to examine if the same categories can be distinguished. Each graph in Figure 6.1 contains a *t-SNE* visualization with different perplexity. Each iteration was run in 1000 steps.

It is evident from a visual evaluation, that members of each category (apart from the *Other* one) create a cluster. Members of the *Other* category are intentionally numbered to analyze their position as well. For example numbers *13* and *15* can be found close to each other in each graph, and indeed the log lines are `end proccessing article add/edit 5E3809514` and `proccessing article add/edit 5E3809514`. Similarly, numbers *5*, *7* correspond to a template `End processing <*> with result <*>` and *11* to `End processing <*>.`

Apparently, the embeddings created by the BERT technique from each log line are distinguishable according to the words they contain.

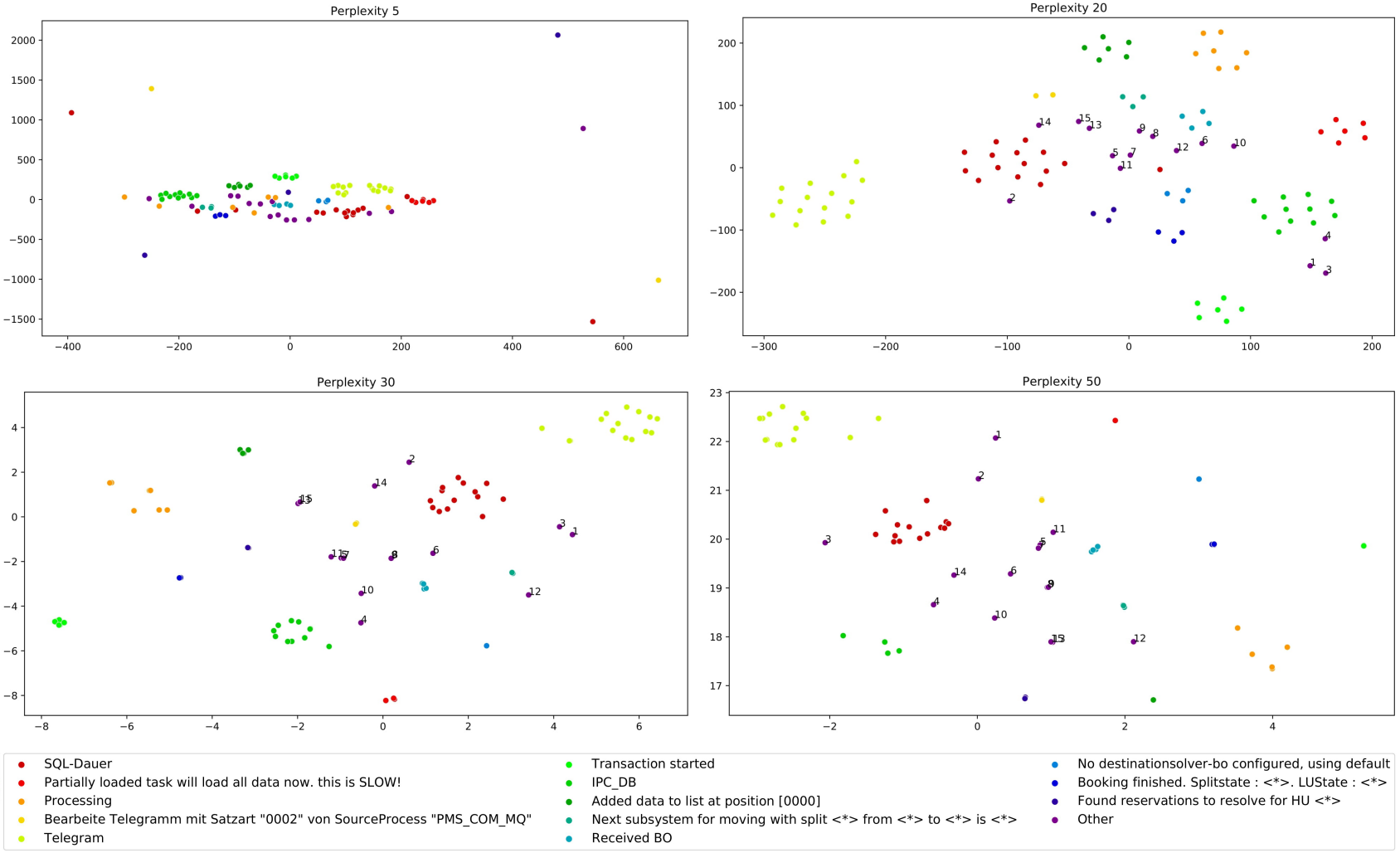


Table 6.1: t-SNE visualization with different perplexity

6.2 Value prediction

6.2.1 Temporal Convolutional Network

The first approach of *SQL-Dauer* value prediction was the TCN model. An already implemented model as described in Section 5.2.1 was used with several minor adjustments made throughout the experiments. Only the CAKL dataset containing the *SQL-Dauer* values was used with this model.

Several experiments were performed with the model. The parameters were an *SQL query*, a *time span* limiting the total number of *SQL-Dauer* values to be predicted by the time of generation of respective log lines and a *window size* i.e. the number of log lines prior to each predicted value. The train/validation split ratio used was 9:1. Training and validation losses were measured in each of these experiments. Since these metrics showed that the model was not learning, the testing part was skipped.

The first SQL query used was the *Query 1*. The first several experiments were performed with a batch size of 64, learning rate 10^{-2} and a *window size* of 100 log lines. Due to RAM limits, the dataset was truncated to 2/3 of the original size. The results were not satisfactory as after 25 epochs the training loss decreased only insignificantly, its value remained above 1 and the validation loss oscillated above 1 as well. The minima for training and validation losses were 1.115 and 1.011 respectively, with the latter being a value of the very first epoch. It implies that the model was not able to train on the dataset at all.

Changes were made first to the learning rate - it was decreased to 10^{-3}, the batch size was reduced initially to 10 and later to 1 log line, so that the potential loss reduction was apparent faster and the *window* size was reduced to 20. The number of epochs was increased to 50. Only certain time spans were used, all within one day - 30/11/20. This date was selected because the analysis of this part of the dataset it was subjected to in section 3.3.1 showed that there is enough data and the *SQL-Dauer* values are sufficiently distributed. All time spans used start at midnight and end after respective amount of hours.

This time the experiments were run for *Query 1*, *Query 2* and *Query 3*. Windows of log lines prior to each *SQL-Dauer* value of each query contain different log lines; therefore, the chance of successful training is bigger and the results can also provide more information crucial for further development. For each query individually, a number of log lines between each occurrence of the query was measured and those values having this number greater than 20000 were skipped. The measured training and validation losses of individual experiments are shown in Table 6.2.

Even though the minimal train loss values are rather low, it is clear from the validation loss minima and graphs in Figure 6.1 - only showing *Query 1* results - that the model is over-fitting on all time spans used. The graphs of *Query*

	Start	End	No of values	Min train loss	Min val loss
Query 1	00:00:00	00:59:59	1350	0.164	0.873
	00:00:00	01:59:59	2585	0.145	0.7
	00:00:00	03:59:59	4424	0.155	0.67
	00:00:00	11:59:59	14090	0.204	0.831
	00:00:00	23:59:59	30732	0.381	0.774
Query 2	00:00:00	00:59:59	1803	0.089	0.432
	00:00:00	01:59:59	3576	0.096	0.346
	00:00:00	03:59:59	6186	0.112	0.419
	00:00:00	11:59:59	15467	0.079	1.254
	00:00:00	23:59:59	31772	0.231	0.401
Query 3	00:00:00	00:59:59	1803	0.087	0.353
	00:00:00	01:59:59	3576	0.099	0.43
	00:00:00	03:59:59	6186	0.101	0.386
	00:00:00	11:59:59	15467	0.079	1.254
	00:00:00	23:59:59	31764	0.23	0.441

Table 6.2: Query 1 - 3 experiments - *30/11/20*

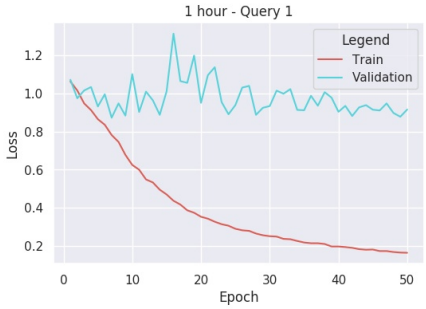
2 and *Query 3* results look very similar. The facts that the amount of data used with this model is not low and over-fitting prevention techniques such as dropouts were used suggest that there might not be enough information in the data itself, or the model is too complex for regression analysis. To confirm or deny this assumption, another approaches were tested in following sections of this chapter.

As explained in section 3.3 none of the queries used in these experiments are not in fact suitable for value prediction; therefore, different ones were chosen and the experiments were rerun. As mentioned, the frequency of both queries is low, only little over 3000 values, per the whole dataset. Splitting the dataset into smaller parts was rather unreasonable, hence only one experiment was conducted for each query with the whole dataset as an input. In contrast to the previous queries, the threshold of 20000 log lines between *SQL-Dauer* values is too small, it was changed to 500000. All other parameters were reused, the batch size equaled 1, the window size stayed 20, the learning rate used was 10^{-3}.

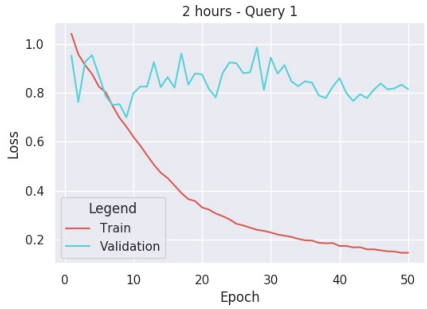
	No of values	Min train loss	Min val loss
Query 6	3351	0.091	0.707
Query 7	3076	0.07	0.478

Table 6.3: Value prediction per whole CAKL dataset - Query 6 & Query 7

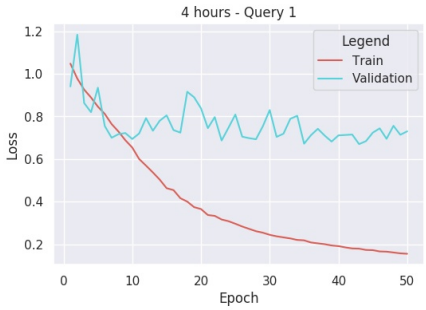
Unfortunately, likely due to the lack of sufficient amount of data the



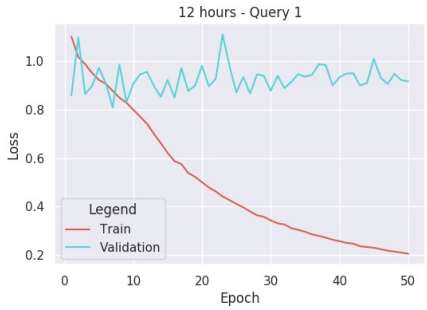
(a) : 1 hour - 00:00:00 - 00:59:59



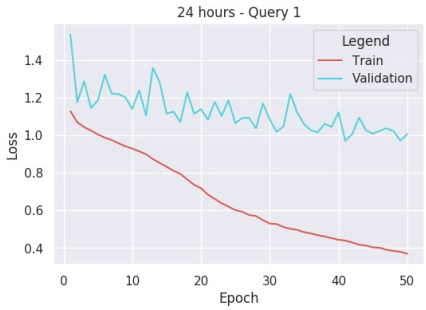
(b) : 2 hours - 00:00:00 - 01:59:59



(c) : 4 hours - 00:00:00 - 03:59:59



(d) : 12 hours - 00:00:00 - 11:59:59



(e) : 1 day - 00:00:00 - 23:59:59

Figure 6.1: Train/Validation loss per different time spans of the day of 30/11/20, with the x-axis representing number of epochs

results did not bring any improvement and the testing part was also skipped. Respective minimal training and validation losses are shown in Figure 6.3.

At the latest stage of this work an implementation error in loading of data that are fed into the TCN model was found; therefore concrete outcomes in this section are invalid.

6.2.2 Longformer

The second approach, which was chosen to be simpler than the TCN model and was potentially able to solve the over-fitting, is also a transformer-based architecture - the *Longformer*.

All three datasets - the CAKL one with the two AWR datasets - were

combined for these experiments and *Query 1, 2* and *3* were selected. The number of log lines in each *window* prior the *SQL-Dauer* value was chosen with respect to the limitations of the model used. As explained in the Section 5.2.2, these log lines are concatenated into one string, and thus the length of 10 log lines appeared to be suitable. Despite it decelerating the process, even a batch of size 10 proved to be too high due to RAM limits, therefore a batch size of 1 was set. The train/validation ratio used was 8:2.

Experiments were again conducted on a portion of the data each, the same time spans of 30 November 2020 as for the TCN model were used - 1 hour, 2 hours, 4 hours, 12 hours and 1 day (24 hours). Again all time spans start at midnight and end after respective amount of hours. Initially the default value of the learning rate - 5×10^{-5} - was used. According to training and validation losses measured, the model was again not training at all.

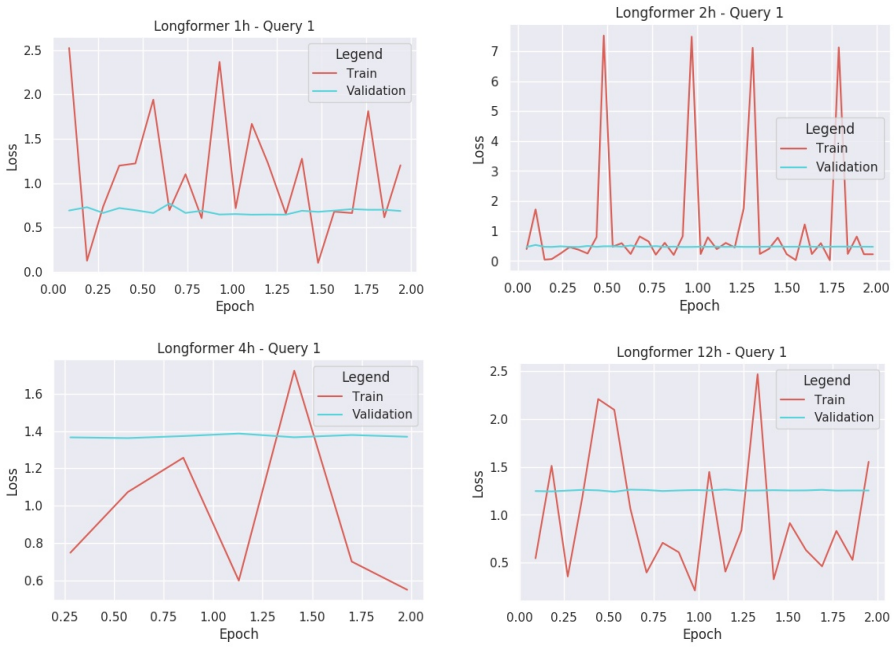


Table 6.4: Train loss per different time spans - Longformer

After that the learning rate was decreased to 10^{-3} and the experiments were run again. Figure 6.4 shows the resulting training loss over two learning epochs for 1-hour to 12-hour time spans of *Query 1* experiments. It is apparent from the graphs that the model is still not learning, the training loss oscillates around the value of 1 over the two epochs. Experiments with *Query 2* and *Query 3* did not bring any significant change.

The *Longformer* approach was proven to be even less suitable for regression analysis than the TCN architecture.

6.2.3 FastText

The results of previous approaches suggest that the data might be too noisy and therefore not hold enough information to be suitable for the regression analysis. For that reason an entirely different approach in terms of value prediction and also embeddings generation was chosen.

Again only a section of the original dataset was used - first 12 hours of the day of 30 November 2020. The experiments were run for the same three queries already used in the approaches before. As for parameters of the fastText model used for word vector creation, several combinations of the dimension or the size of the word vector, minimum count of word to be included and minimal and maximal length of character n-gram were tested, as shown in Table 6.5.

	Dimension	Minimum count	Minimum n	Maximum n
1	100	5000	1	1
2	100	5000	3	6
3	300	5000	1	1
4	300	5000	3	6
5	100	10000	1	1
6	100	10000	3	6

Table 6.5: fastText model parameters combinations

The regression analysis *Seq2Seq* model consists of one LSTM layer, two hidden layers with the width of 300, and the learning rate was set to 10^{-3}. All experiments were run for 20 epochs with batch size of 64.

Experiments were run for *Query 1*, *Query 2* and *Query 3*. Training and validation losses were measured during each run, the results are demonstrated in table 6.6.

	Combination	Min train loss	Min val loss
Query 1	1	0.0613	0.0652
	2	0.0616	0.0643
	3	0.0625	0.0648
	4	0.0622	0.0642
	5	0.0616	0.0649
	6	0.0612	0.0624

Table 6.6: fastText experiments - Query 1

Only results of *Query 1* experiments are shown as the results of *Query 2* and *Query 3* runs yielded much worse numbers. Neither training nor validation losses decreased below 0.5, rather oscillated between 0.5 and 0.6.

The minimal losses of *Query 1* experiments are rather low, but they do not change much over the epochs.

6.2.4 Data split

As none of the models was not able to train on the dataset, instead of a proper three-way data split, only training and validation sets were used.

6.3 Anomaly detection

The anomaly detection experiments were run for three queries again, however they were chosen according to a different criteria. As demonstrated in Figure 3.5 only certain queries are suitable for anomaly detection, the *SQL-Dauer* of which form only two categories - *normal* and *anomalous*. Apart from *Query 1*, also *Query 4* and *Query 5* introduced in section 3.3.1 meet this requirement. When generating the embeddings for *Query 4* and *Query 5* the threshold for log lines in-between was set to 100000.

Initially, the Temporal Convolutional Network approach experiments were run. As with the value prediction experiments, for each query there were five different time spans selected - 1 hour, 2 hours, 4 hours, 12 hours and 24 hours. All these time spans were again taken from the day of 30/11/20 and overlap as they all start at midnight and end after respective number of hours.

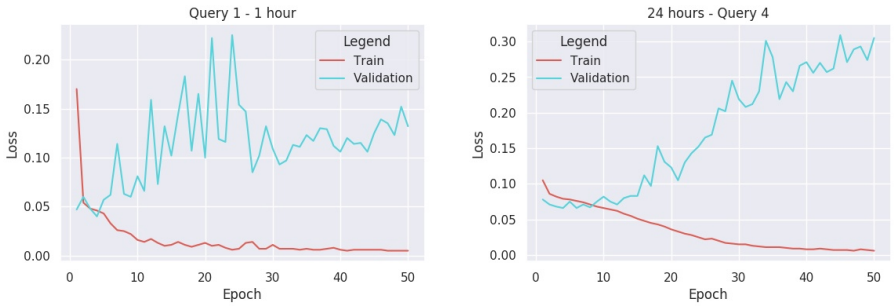


Figure 6.2: Train/Validation loss of anomaly detection experiments

The number of epochs was set to 50, learning rate stayed at 10^{-3}. For each run a train and a validation loss were measured per epoch. Results are shown in Table 6.7.

Both train and validation losses significantly decreased compared to the regression experiments. However, the minimum values for the time spans other than 12 hours occur at the earlier epochs and the validation loss either oscillates or increases throughout the 50 epochs. Examples of one-hour time span of *Query 1* and 24-hour time span *Query 4* is shown in Figure 6.2. The only promising results for all queries were of the experiments run with the

	Start	End	No of values	Min train loss	Min val loss
Query 1	00:00:00	00:59:59	1348	0.005	0.012
	00:00:00	01:59:59	2583	0.003	0.049
	00:00:00	03:59:59	4419	0.004	0.059
	00:00:00	11:59:59	14082	0.011	0.006
	00:00:00	23:59:59	30718	0.015	0.055
Query 4	00:00:00	00:59:59	840	0.002	0.072
	00:00:00	01:59:59	1604	0.001	0.104
	00:00:00	03:59:59	2709	0.003	0.047
	00:00:00	11:59:59	8153	0.004	0.002
	00:00:00	23:59:59	17785	0.006	0.066
Query 5	00:00:00	00:59:59	836	0	0.002
	00:00:00	01:59:59	1675	0.001	0.077
	00:00:00	03:59:59	2844	0.003	0.036
	00:00:00	11:59:59	7314	0.005	0.001
	00:00:00	23:59:59	15369	0.006	0.062

Table 6.7: Query 1, 4, 5 anomaly experiments - *30/11/2020*; initial 1h, 2h, 4h, 12h and 24h time spans per query

	Start	End	No of values	Min train loss	Min val loss
Query 1	00:00:00	09:59:59	11994	0.009	0.005
	00:00:00	10:59:59	12757	0.011	0.005
	00:00:00	12:59:59	15931	0.011	0.005
	00:00:00	13:59:59	17070	0.011	0.005
Query 4	00:00:00	09:59:59	6937	0.009	0.005
	00:00:00	10:59:59	7367	0.011	0.005
	00:00:00	12:59:59	9143	0.011	0.005
	00:00:00	13:59:59	9717	0.004	0.002
Query 5	00:00:00	09:59:59	6364	0.002	0.001
	00:00:00	10:59:59	6600	0.003	0.001
	00:00:00	12:59:59	8109	0.004	0.002
	00:00:00	13:59:59	8516	0.003	0.002

Table 6.8: Query 1, 4, 5 anomaly experiments - *30/11/2020*; additional 10h, 11h, 13h and 14h time spans per query

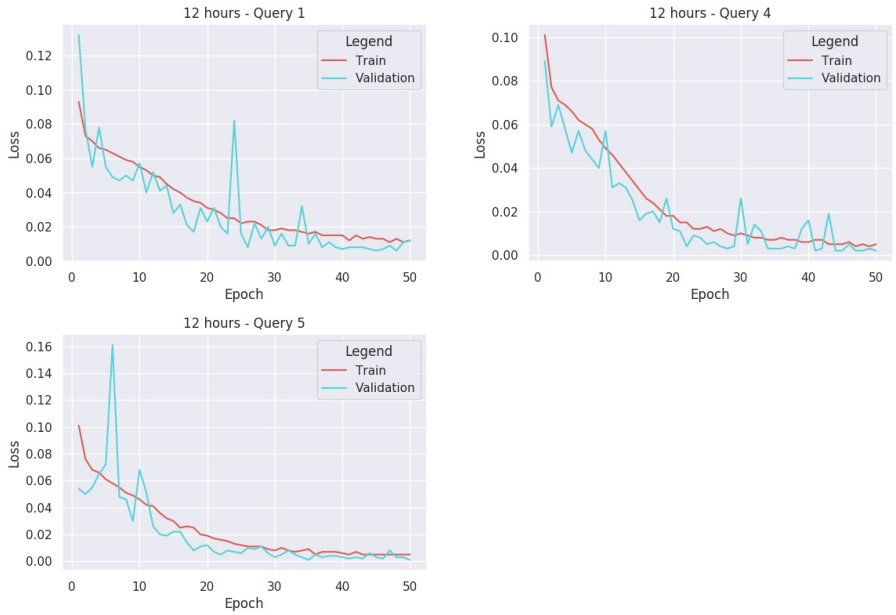


Figure 6.3: Train/Validation loss per 12-hour time span of the day of 30/11/2020

12-hour time span as shown in Figure 6.3, where both training and validation losses seem to reasonably converge to a small number.

Further analysis needed to be performed, since such difference between the time spans, but also such consistency between the queries is slightly surprising. To find out why exactly 12-hour time span is better than the others, for each of the queries further experiments were run - time spans around 12 hours were used, specifically 10, 11, 13 and 14 hours. Corresponding minimum train and validation losses per query per time span are demonstrated in Figure 6.8. The graphs of these experiments look similar to those in Figure 6.3. This suggests that the data might be too noisy in the early hours after midnight and the model learns to deal with it with longer time spans. It does not explain though, why the 24-hour time span yield unsatisfactory results.

Until now, all anomaly detection experiments were conducted with the input data being a part of the 30/11/20 day only and all time spans overlap as they all start at midnight and finish after respective number of hours. Three other days were selected - 1/12/20, 8/12/20, and 14/12/20 - and the data of each day were split into two disjunctive 12-hour-long intervals. The results of these experiments are illustrated in Table 6.9. Values of both the training and validation losses are again reasonable, and graphs of their development through epochs show similar trend as those in Figure 6.3.

To test the assumption that the data might be noisy in the early hours after midnight of 30/11/20, another test was run for *Query 1* with the 12-hour time span shifted to start at 04:00:00 and end at 16:00:00. It apparent from the graph in Figure 6.4 that the validation loss curve is less smooth than in the previous graphs.

	Start	End	No of values	Min train loss	Min val loss
01/12	00:00:00	11:59:59	13615	0.012	0.007
	12:00:00	23:59:59	15413	0.012	0.006
08/12	00:00:00	11:59:59	31988	0.012	0.007
	12:00:00	23:59:59	16189	0.01	0.005
14/12	00:00:00	11:59:59	20068	0.002	0.001
	12:00:00	23:59:59	21468	0.002	0.001

Table 6.9: Query 1 anomaly experiments - 12-hour time spans per days 1/12/20, 8/12/20 and 14/12/20

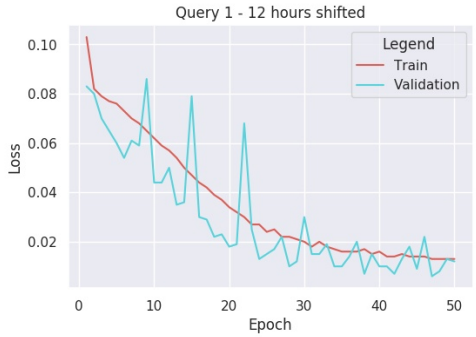


Figure 6.4: Train/Validation loss of shifted time span

Another approach to discover where the problem might be is in reusing a model that was rather successfully trained in one of previous experiments. It was decided that one of the very first runs with *Query 1* and 12-hour time span will be used. When running the following experiments, a previously saved state of the trained model was first loaded from memory and then fine-tuned during the experiments.

Initially, the 12-hour time span was split into three disjunctive 4-hour-long intervals. Neither of these tests brought any progress.

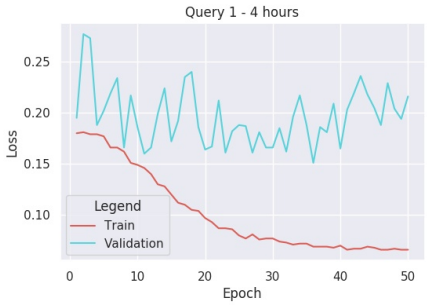
6.3.1 Error correction

At this stage of work an implementation error in loading of data that are fed into the TCN model was found. Unfortunately this had implications to the results of all experiments in this section. The error was corrected and it was necessary to run the experiments again. Also at this time the activation function of the output layer was changed from linear to sigmoid function.

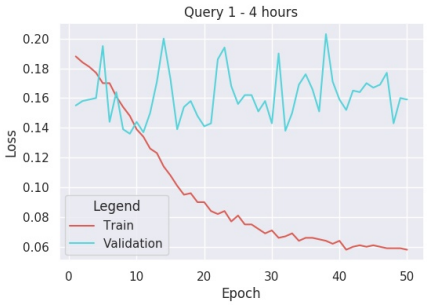
With the adjustments set, new experiments were run for each query (*Query 1*, *Query 4* and *Query 5*) with the same set of time spans of the day 30/11/20 - 1 hour, 2 hours, 4 hours, 12 hours and 24 hours. Each time span starts at midnight and ends after respective number of hours. Number of epochs was 50, window size 20 and learning rate 10^{-3}.

	Start	End	No of values	Min train loss	Min val loss
Query 1	00:00:00	00:59:59	1348	0.034	0.016
	00:00:00	01:59:59	2583	0.075	0.13
	00:00:00	03:59:59	4419	0.066	0.151
	00:00:00	11:59:59	14082	0.051	0.185
	00:00:00	23:59:59	30718	0.011	0.214
Query 4	00:00:00	00:59:59	840	0.059	0.16
	00:00:00	01:59:59	1604	0.094	0.174
	00:00:00	03:59:59	2709	0.051	0.219
	00:00:00	11:59:59	8153	0.103	0.23
	00:00:00	23:59:59	17785	0.035	0.235
Query 5	00:00:00	00:59:59	836	0.021	0.001
	00:00:00	01:59:59	1675	0.047	0.209
	00:00:00	03:59:59	2844	0.059	0.17
	00:00:00	11:59:59	7314	0.066	0.226
	00:00:00	23:59:59	15369	0.066	0.269

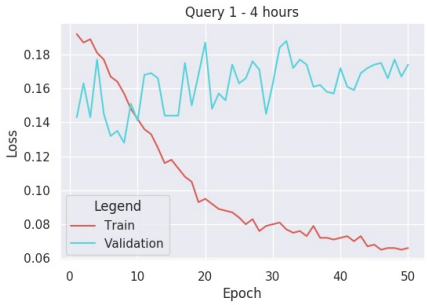
Table 6.10: Query 1, 4, 5 anomaly experiments after error correction - *30/11/2020*



(a) : Window size 20



(b) : Window size 50



(c) : Window size 100

Figure 6.5: Train/Validation loss per 4-hour time span of the day of 30/11/2020 with different window sizes

Results of these tests are demonstrated in Table 6.10. The trend of training losses was decreasing even in the later epochs, hence with higher number of epochs the number could possible drop even further, but the validation loss trend suggests that the model is not training.

Other experiments were run for *Query 1* only. The first set was to try different window sizes, if it influences the result. A 4-hour time span was used in these experiments. Graphs in Figure 6.5 illustrate training and validation loss behaviour over the 50 epochs. Clearly the window size in this case does not affect the ability of the model to learn on this portion of data.

Also another test for *Query 1* was conducted, training on a two-day-long dataset, which consists of data of the days 30/11/20 and 1/12/20. Neither this experiment brought any improvement as the validation loss oscillated between 0.2 and 0.3.

As with the regression analysis, as the validation jobs were not successful, further testing on previously unused data was skipped.

6.4 Discussion

Initially, the experiments with BERT architecture proved to be a suitable tool for creation of contextual embeddings of logs. Clusters formed in the embedding space contain semantically similar log lines.

The experiments with regression analysis the aim of which was to predict a certain value that occur repeatedly in the provided dataset were not satisfactory. None of the three approaches used (TCN, Longformer, fastText) verifies that the contextual embeddings provide enough information to the model so that it is able to learn on the data. It is possible that either the regression task is too complicated or the dataset itself is noisy.

A reduction in terms of complexity of the task was applied. Instead of predicting a certain value, experiments were run to train the TCN model to classify the values as either normal or anomalous. Unfortunately an implementation error in the TCN model data loading meant a significant setback. After a correction only few experiments were run, that did not yield any improvement as the model was not able to train on the data.

Chapter 7

Conclusions

This thesis studied the usage of contextual embeddings in log analysis, with focus on regression analysis. It extends previous work on numerical representations of log lines and their leveraging in anomaly detection. The research shows that evaluation results of non-contextual embeddings depend on the dataset used and also on a model used for evaluation. Nevertheless, there are approaches showing outstanding results in anomaly detection experiments. Significantly less work was done on contextual embeddings of logs, but the initial results yielded on an anomaly detection task are promising and worth further research.

Three custom not publicly available datasets were provided for this work. CAKL - the largest one was mainly used in experiments. As there was no prior information about the data it contains, an extensive analysis of the dataset had to be conducted. It revealed several important features and the overall complexity of the dataset. The initial aim of predicting of a certain value in any SQL query occurring throughout the dataset was reduced to few query representatives that were subjected to further inspection.

Initially, the contextual representations of log lines based on *BERT* architecture were created. A visual evaluation proved that this approach is a suitable method for contextual embeddings creation of log lines.

Several regression analysis models were chosen to properly evaluate this assumption. The first one was Temporal Convolutional Network model using the previously created contextual embeddings as input. The second one was the Longformer sequence classification model, a BERT architecture suitable for long sequences. None of these models were able to train on the dataset. A baseline approach using *fastText* technique and its unsupervised model for embeddings creation was then used. However, the model did not show any remarkable results, as it was not able to train on the dataset either.

Finally, to simplify the task of regression analysis, anomaly detection experiments were conducted using an updated TCN model. Seemingly, this model was able to train on a certain portion of the dataset and further experiments were run to clarify the reason why other parts of the dataset are not suitable. While working on this task, an implementation error in

the TCN model data loading was found which unfortunately invalidated the experiments and results conducted earlier with this technique. After the correction several experiments were repeated, but neither this approach was successful.

The next step would be evaluation of the proposed architectures on a different dataset to learn if the provided one was suitable. A search of a dataset that would have similar features such as a value that was worth predicting was made during the first regression analysis experiments, unfortunately no such publicly available dataset was found.

Bibliography

[1] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate, 2016.

[2] S. Bai, J. Z. Kolter, and V. Koltun. An empirical evaluation of generic convolutional and recurrent networks for sequence modeling. *CoRR*, abs/1803.01271, 2018.

[3] I. Beltagy, M. E. Peters, and A. Cohan. Longformer: The long-document transformer. *CoRR*, abs/2004.05150, 2020.

[4] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov. Enriching Word Vectors with Subword Information. *Transactions of the Association for Computational Linguistics*, 5:135–146, 06 2017.

[5] A. M. Dai and Q. V. Le. Semi-supervised sequence learning, 2015.

[6] J. Devlin, M. Chang, K. Lee, and K. Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[7] M. Du and F. Li. Spell: Streaming parsing of system event logs. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*, pages 859–864, 2016.

[8] M. Du, F. Li, G. Zheng, and V. Srikumar. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. CCS '17, page 1285–1298, New York, NY, USA, 2017. Association for Computing Machinery.

[9] S. Han, Q. Wu, H. Zhang, B. Qin, J. Hu, X. Shi, L. Liu, and X. Yin. Log-based anomaly detection with robust feature extraction and online learning. *IEEE Transactions on Information Forensics and Security*, 16:2300–2311, 2021.

[10] P. He, J. Zhu, S. He, J. Li, and M. R. Lyu. An evaluation study on log parsing and its use in log mining. In *2016 46th Annual IEEE/IFIP*

International Conference on Dependable Systems and Networks (DSN), pages 654–661, 2016.

[11] P. He, J. Zhu, Z. Zheng, and M. R. Lyu. Drain: An online log parsing approach with fixed depth tree. In *2017 IEEE International Conference on Web Services (ICWS)*, pages 33–40, 2017.

[12] S. He, J. Zhu, P. He, and M. R. Lyu. Experience report: System log analysis for anomaly detection. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, pages 207–218, 2016.

[13] M. Koryťák. Anomaly Detection Methods for Log Files. Master's thesis, Czech Technical University in Prague, 2021.

[14] T. Kudo and J. Richardson. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. *CoRR*, abs/1808.06226, 2018.

[15] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov. Roberta: A robustly optimized BERT pretraining approach. *CoRR*, abs/1907.11692, 2019.

[16] M.-T. Luong, H. Pham, and C. D. Manning. Effective approaches to attention-based neural machine translation, 2015.

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

[18] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer. Deep contextualized word representations, 2018.

[19] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever. Improving language understanding with unsupervised learning. 2018.

[20] N. Reimers and I. Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. *CoRR*, abs/1908.10084, 2019.

[21] V. Sanh, L. Debut, J. Chaumond, and T. Wolf. Distilbert, a distilled version of BERT: smaller, faster, cheaper and lighter. *CoRR*, abs/1910.01108, 2019.

[22] M. Schuster and K. Nakajima. Japanese and korean voice search. In *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5149–5152, 2012.

[23] R. Sennrich, B. Haddow, and A. Birch. Neural machine translation of rare words with subword units. *CoRR*, abs/1508.07909, 2015.

[24] M. Souček. Log Anomaly Detection. Master's thesis, Czech Technical University in Prague, 2020.

[25] R. Vaarandi and M. Pihelgas. Logcluster-a data clustering and pattern mining algorithm for event logs. In *2015 11th International conference on network and service management (CNSM)*, pages 1–7. IEEE, 2015.

[26] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[27] X. Zhang, Y. Xu, Q. Lin, B. Qiao, H. Zhang, Y. Dang, C. Xie, X. Yang, Q. Cheng, Z. Li, J. Chen, X. He, R. Yao, J.-G. Lou, M. Chintalapati, F. Shen, and D. Zhang. Robust log-based anomaly detection on unstable log data. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 807–817, New York, NY, USA, 2019. Association for Computing Machinery.

[28] J. Zhu, S. He, J. Liu, P. He, Q. Xie, Z. Zheng, and M. Lyu. Tools and benchmarks for automated log parsing. 11 2018.

[29] P. Černý. Contextual Embeddings for Anomaly Detection in Log Files. Master's thesis, Czech Technical University in Prague, 2021.