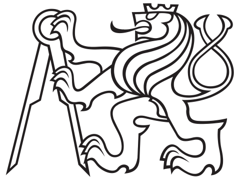
Master Thesis



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

Log Anomaly Detection

Marek Souček

Supervisor: Ing. Jan Drchal, Ph.D.
Field of study: Open Informatics
Subfield: Cybersecurity
August 2020

CTU CZECH TECHNICAL UNIVERSITY IN PRAGUE

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Souček Marek** Personal ID number: **457106**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Computer Science**

Study program: **Open Informatics**

Specialisation: **Cyber Security**

II. Master’s thesis details

Master’s thesis title in English:

Log Anomaly Detection

Master’s thesis title in Czech:

Detekce anomálií z logů

Guidelines:

The task is to develop, implement, and evaluate methods of anomaly detection in log file data.
1) Familiarize yourself with methods of anomaly detection with a focus on processing log files.
2) Start with DeepLog [1] algorithm as a baseline.
3) Explore possibilities of the end-to-end differentiable models and design such a model.
4) The model will most likely involve text embedding, such as FastText [2].
5) Evaluate the model on public log datasets such as HDFS or OpenStack.

Bibliography / sources:

[1] 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.
[2] Bojanowski, Piotr, et al. "Enriching word vectors with subword information." Transactions of the Association for Computational Linguistics 5 (2017): 135-146.

Name and workplace of master’s thesis supervisor:

Ing. Jan Drchal, Ph.D., Artificial Intelligence Center, FEE

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **05.02.2020** Deadline for master's thesis submission: **22.05.2020**

Assignment valid until: **30.09.2021**

Ing. Jan Drchal, Ph.D. Supervisor’s signature	Head of department’s signature	prof. Mgr. Petr Páta, Ph.D. Dean’s signature

III. Assignment receipt

The student acknowledges that the master’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master’s thesis, the author must state the names of consultants and include a list of references.

.

Date of assignment receipt Student’s signature

 © ČVUT v Praze, Design: ČVUT v Praze, VIC

Acknowledgements

I wish to express my sincere appreciation to my supervisor Ing. Jan Drchal, Ph.D. for his patience and helpful guidance. I appreciate technical support of Research Center for Informatics at CTU, which provided computation cluster, where my experiments were hosted. Special thanks then belong to Jakub Hejret, for editorial corrections of this thesis. I also wish to thank my family and friends for support through all 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.

Prague, 14. August 2020

Abstract

This thesis explores possibilities of applying recent advancements in NLP domain to log anomaly detection. More specifically it tests whether fastText, as advanced NLP embedding approach, can be used to model logs, which do not contain typical natural language, but they are unstructured or semi-structured human readable text. Proposed log representation was used as input for supervised and unsupervised LSTM based anomaly detection models. These models were evaluated in multiple experiments and compared witch anomaly detection method on two publicly available datasets. Supervised approach showed some really good results and placed among the best methods in benchmark.

Keywords: anomaly detection, logs, NLP, LSTM

Supervisor: Ing. Jan Drchal, Ph.D.

Abstrakt

Tato diplomová práce se zabývá možností aplikovat nedávné pokroky v oblasti zpracování přirozeného jazyka (NLP) na problém detekce anomálií z logů. Konkrétně zkouší, zda lze použít fastText, jakož to pokročilou metodu NLP embeddingu, k reprezentaci logů, jejichž text neobsahuje přirozený jazyk, ale je to stále nestrukturovaná nebo jen částečně strukturovaná informace ve formě čitelného textu. Navrhnutá reprezentace logů je použita jako vstup pro detekci anomálií se supervizovanými i nesupervizovanými modely založenými na LSTM neuronových sítích. Výslendé modely byly vyhodnoceny a porovnány s dalšími metodami detekce anomálií na dvou veřejně dostupných datasetech. Supervizované modely dosáhly velmi dobrých výsledků a v pozovnání se umístili mezi nejepšími metodami.

Klíčová slova: detekce anomálií, logy, NLP, LSTM

Překlad názvu: Detekce anomálií z logů

Contents

1 Introduction 1

2 Related work 3

2.1 Log collection 4

2.2 Log parsing 4

2.3 Feature extraction 6

2.3.1 Aggregating representations . . 6

2.3.2 Per line representations 7

2.3.3 Word2vec and fastText 8

2.4 Anomaly detection 9

3 Problem analysis 11

4 Proposed architecture 13

4.1 Embedding 13

4.2 Unsupervised model 15

4.3 Supervised model 17

5 Implementation 19

5.1 Preprocessing and benchmarks . 20

5.2 Models 21

5.3 Embedding and data management 24

6 Experiments and evaluation 27

6.1 Datasets 27

6.1.1 DHFS 28

6.1.2 BGL 29

6.2 Embedding analysis 30

6.3 Anomaly detection 34

6.3.1 Supervised anomaly detection 35

6.3.2 Unsupervised anomaly detection 39

6.4 Experiments summary 42

7 Conclusion 43

A Bibliography 45

B Command line interface 49

C Content of enclosed CD 51

Figures

2.1 Log anomaly detection framework 3
2.2 Example of log parsing 5

4.1 Data flow in proposed architecture 13
4.2 Data flow in embedding 14
4.3 Structure of prediction model . . 16
4.4 Structure of classification model 17

5.1 Process of data preprocessing . . 20
5.2 Input and output data for models 23
5.3 Data loading and padding 24

6.1 Window length distribution HDFS 29
6.2 Window length distribution BGL 30
6.3 Embedding visualization across datasets (t-SNE reduction) 31
6.4 Embedding visualization HDFS . 32
6.5 Embedding visualization BGL . . 33
6.6 Supervised training 37
6.7 Supervised output distribution (HDFS data) 37
6.8 Supervised thresholds 38
6.9 Prediction errors 40
6.10 Prediction errors distribution (HDFS data) 41
6.11 Unsupervised thresholds 41

Tables

6.1 Summary of datasets 28
6.2 Benchmark results on HDFS . . . 35
6.3 Benchmark results on BGL 35
6.4 Results of supervised models . . . 36
6.5 Results of unsupervised models with different embeddings 39

Introduction

Software systems produce logs to record events and system current state. Logging has been commonly adopted in practice, because of its simplicity and effectiveness. Logs are essential and valuable information source for developers and operators which can examine recorded logs to understand the system state when troubleshooting, by detecting system anomalies and locate the root causes. And in era of cloud computation even every day tasks as billing can be based on logs which recorded use of service by customer.

Modern systems are scaling up and moving to distributed computation in cloud. These large scale systems are supporting online services as search engines, social networks or e-commerce and computation heavy application such as whether forecasts. Many of these systems are designed to operate 24/7 serving millions of users globally. Any quality degradation or even outage of such services are very costly, so timely discovery of any changes and ability to quickly find out root cause of problem is important. But these systems generate large amount of logs with rate of tens of gigabytes per hour. Such volume is difficult if not impossible to manually analyze, even with search and filtering tools.

In reaction to growing amount of logs, automated log analysis and anomaly detection have become important research topic, in last years. Automation promises online monitoring and timely anomaly alerting, which allow developers and operators to focus their effort just on solving the problems. But that is still in the future. With the amount of logs from these systems, even small ratio of false positive alerts can still overwhelm operators. And log based anomaly detection have proven to be challenging tasks. Most of the important information is hidden in log messages, which are usually unstructured or semi-structured text strings, and as such hard to process by algorithms. Also log based anomaly detection have to often deal with constantly changing environment, caused by frequent system updates.

This thesis propose use of recent advances in NLP (Natural Language Processing) in combination with machine learning to make log based anomaly detection more independent on type of logs it process. That should make

it more robust to gradual changes caused by system updates, and easier to deploy on different systems.

Chapter 2

Related work

Logs are valuable information source in many scenarios from debugging in development to monitoring in production. Logs record events and current system state. And these data were examined manually at first. But specialized tools for log processing an analysis were developed over time. When amount of logs increased as applications scaled up and become more complex. First tools included basic text processing utilities as keyword search and regular expressions. Then more complex filters and rule based alerting came to reduce amount of logs for manual examination. Then applications moved to cloud and distributive processing, and amount of false positive alerts from rule based alerting become too large for manual processing. Now more sophisticated tools, which often utilize machine learning, are starting to be used, to deal with the flood of data. There are two general areas on which new features are now focusing. Machine learning is used to improve alerting by further reducing false positive rate. Then more advanced visualization and organization tools are developed, with functionalities as grouping similar logs or logs caused by the same event, to make manual processing of alerts more efficient.

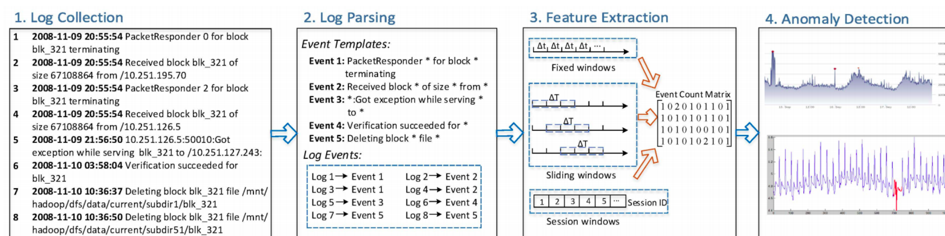


Figure 2.1: General log anomaly detection framework. Picture taken from: *Experience Report: System Log Analysis for Anomaly Detection*[1]

Figure 2.1 illustrates that log based anomaly detection can be split into several general steps, namely *log collection*, *log parsing*, *feature extraction* and *anomaly detection*. This general framework was used and described, among other, in report [1] where several anomaly detection methods were compared. Most publications recognize these steps as more or less separate tasks and

often focus only on some of them. Brief goal description and review of related work for each step will follow in next sections.

2.1 Log collection

Nowadays large-scale systems generate large amount of logs distributed over many machines potentially even across multiple data centers on different continents. Logs record system states and run-time information which should be saved for future use. Logs are not used only for anomaly detection when monitoring system performance or security but also billing, auditing etc. On the other hand all these use-cases expect to be able to access all relevant logs, but that might not be so simple task with distributed nature of current systems.

Collecting, storing logs and providing when needed is important step that need to be kept in mind when building system. There is several options. Smaller non-distributed application can use simple log files. Larger system might use protocols like *syslog*[1] for centralized monitoring and storage. And each big cloud provider offer some solution within their cloud as *Amazon CloudWatch*[2] or *Google Cloud Logging*[3].

But log collection is out of scope for this thesis, which is focused more on their processing. Publicly available log datasets from other publication on log processing and anomaly detection are used for experiments in this thesis.

2.2 Log parsing

Raw logs are unstructured and contain free form text. Generally it is hard to process unstructured data in automated fashion. So goal of log parsing is to get some structured representation of information contained in raw data, so it can be more easily processed algorithms.

But firstly some definitions of basic terms about logs and their parts. There is no universally accepted terminology, common terms as log template, log event or log key are often interchangeable but not always. In this thesis following terms will be used: log statement, log header, log message, log template, log key and parameters.

Illustrative example in Figure 2.2 shows how logs are created, stored in raw text and parsed to structured data. One log statement, usually one line of raw text, records information abut current system state or event that happened. Log statement consists of two parts log header and log message.

[1] https://tools.ietf.org/html/rfc5424
[2] https://docs.aws.amazon.com/cloudwatch/index.html
[3] https://cloud.google.com/logging

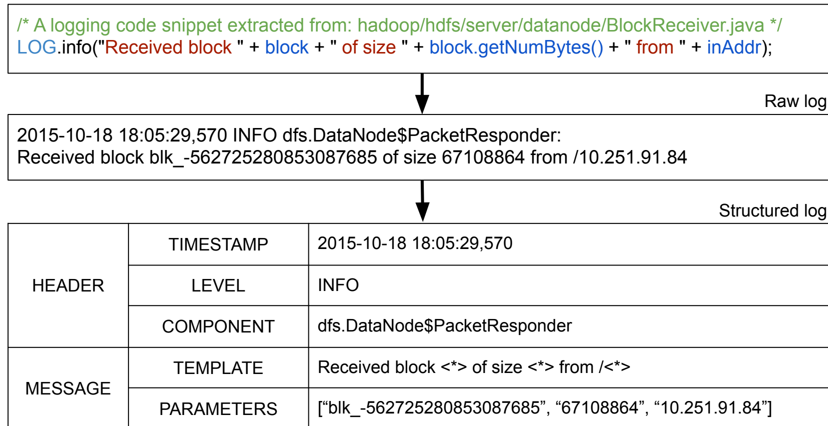


Figure 2.2: Example of log parsing

Log headers are created by logging frameworks and contain basic common contextual information related to each log statement. Header usually include timestamp, severity level (e.a. INFO, ERROR...) and source component, but can also include additional information as process identifiers etc. Headers are relatively easy to parse, since they are generated by logging framework and the same formatting is used for all logs, at least within one application. On the other hand log messages are free form text supplied by developer. They contain more specific and rich information about system state, but are also much harder to parse. Log message is composition of string constants and variable values. The constant parts define log template which remains the same for all occurrences of given event. This is why in some publication log template is sometimes called log event. Log key is also very similar term and in some cases interchangeable with the other two, but in this thesis log key represents enumerative id of given log template. And finally log parameters are the variable part of log message.

Comparison and benchmark of different parsing tools is presented in [2]. There are three general approaches to log parsing. Traditional log parsing relies on handcrafted regular expressions to extract template and parameters. This is straight forward but time-consuming and error-prone for systems with larger number of templates (e.g. over 70k templates in Android framework according to [2]). Modern software systems also change templates frequently as they are updated (even hundreds of templates per month [3]). Static source code analysis was proposed in several publications including [3]. But this approach is also limited because it is common practice to use third-party software without access to source code. And finally many datadriven approaches have been proposed, including frequent pattern mining (LogCluster [4]), iterative partitioning (IPLoM [5]), hierarchical clustering (LKE [6]), longest common sub-sequence computation (Spell [7]), parsing tree (Drain

[8]).

Benchmark results from [2] show that parsing tolls are improving but parsing is hard task and no tool is perfect. Architecture proposed in this thesis in chapter 4 does not require parsed templates on its input and much simpler tools for header parsing would be sufficient. But other methods used in experiments for comparison require it so Drain is used for logs parsing in this thesis, because it came as currently the best tool in benchmark comparison provided by [2].

2.3 Feature extraction

Structured logs, obtained from parsing, allow easier manipulation and automated analysis. But this thesis focuses on machine learning approaches which require numerical values as input and output. Goal of feature extraction is to encode information from structured, but still textual, data to numeric vectors, on which machine learning models can be applied.

Feature extraction is key step in any machine learning task. Because even the most sophisticated models cannot decide meaningfully, if relevant information is missing in chosen numeric representation. Feature extraction need to find measurable features that will keep as many information, as possible to inform decisions. Choosing right value encoding is also important. For some models, as random forests, numeric id of type might be ok, but one-hot encoding of the same value will be required for efficient use of neural networks.

Depending on chosen features, extraction is often closely connected to parsing. Or in same cases it can even partially replace the parsing. This is especially true for some natural language processing methods (NLP) which have lately stated to be used for processing of free form text in log message.

Logs are chronological series of separate log statements. Modern anomaly detection have to consider context and at least some history to overcome rule based alerting systems, that considers only one log statement. There are two types of machine learning models. Models capable of processing series and more common traditional ones, which process one input at a time independently. Both types are described more in Section 2.4. Features and resulting feature vectors have to provide information about last log statement and its context in representations suitable for chosen model. So features can be also divided into two categories.

2.3.1 Aggregating representations

All contextual information, including history, has to be encoded into one feature vector, when model consider each input independently. For that reason

many aggregating features were presented and used in different systems.

First step, before aggregating features can be used, is to select subset of log statements which will be aggregated. Common approaches are fixed and sliding windows over time or number of log statements. But other also exists, especially if solution is build for specific application. Concept of sessions can be used in some applications. In HDFS dataset, which is also used in this thesis, logs are grouped and labeled by session. It is important to realize that anomalies are detected on window or session level and not log statement level, with aggregating representations.

Aggregating representations often refer to NLP methods, even though they usually do not process text in logs directly. But NLP is about extracting semantic information from series of data (words) and log history is series of data. So general ideas or methods about information retrieval as *bag-of-words* or *TF-IDF* are used on some aspects of logs.

The most straightforward method of constructing the feature vector is the *bag-of-words* algorithm used in NLP. *Bag-of-words* is simple count vector of occurrences and can be calculated over log keys in window [1]. Some statistical features as event ratio, mean inter-arrival time, mean inter-arrival distance, severity spread and time-interval spread are proposed in [9]. Prefix, from [10], focuses on the patterns in template sequences and propose to extract four features named sequence, frequency, surge and seasonality. Another approach presented in [11, 12] used *word2vec* and considered whole log line as one word when learning embedding vectors. Embeddings of whole log file or time window are then aggregated.

2.3.2 Per line representations

The feature vector represents one log statement, when anomaly detection models capable of processing series are used. It is expected that information hidden in history is extracted by model directly and does not have to be explicitly encoded in feature vector. Instead there is focus on rich representation of one log statement. Resulting feature vector should provide as much information as possible including semantic of message, appropriately encoded parameters or contextual information, that is not included or hard to obtain from series of previously seen logs.

Most of the important information exploited by human during log analysis is in free form text in log message. That is why many NLP methods were explored lately to help capture information hidden in log massage. Recurrent neural network (RNN) language models for both word and character level tokenizations were build in [13] to represent log lines.

Then there have been breakthrough in NLP embedding field caused by *word2vec* followed by *fastText*. These methods and ideas behind them are briefly described in Section 2.3.3, because *fastText* is used in this thesis. These

NLP embedding methods and their ideas were already used few times in context of log processing.

Template2vec presented in [17] is based on *word2vec*. It takes log key and tries to enrich it with semantic information extracted from log template using *word2vec* embedding. Thanks to modification it can handle unknown log keys online, but recommends to periodically retrain, to incorporate new log keys properly. Very similar approach is taken by [18], where off-the-shelf *FastText* model pre-trained on large general text corpus dataset is used on log templates to create template feature vector.

Basic string similarity coupled with clustering methods is presented in [19]. And Deeplog presented in [20] is one of really few experiments which try to take into account log parameters when detecting anomalies, and it do so by training several separated models. One for analysis of log keys sequence and another one for each log key which estimates likelihood of given parameters. These do not create one feature vectors, but they are mentioned here as original approaches to utilize information hidden in single log statement.

2.3.3 Word2vec and fastText

Word2vec from [14] caused breakthrough for word embedding in both quality and computation complexity. It is a two-layer neural net that processes text by turning words into embedding vectors. Its input is a text corpus and its output is a set of vectors that represent words in that corpus. While *word2vec* is not a deep neural network, it turns text into a numerical form that deep neural networks can understand. Popularity of *word2vec* comes from its power to extract semantic information and encode it into embedding vector in a way that can be exploited by simple arithmetic operations. Famous examples used in original article are word relation questions like: What is the word that is similar to *grandson* in the same sense as *brother* is similar to *sister*? Surprisingly correct answer *granddaughter* can be found as closes word vector to result of simple expression $brother - sister + grandson$.

One of the very few imperfections of *word2vec* is limitation to only represent words included in training corpus. As result of following research *fastText* was presented [15, 16]. *FastText* embedding is trained not only on words but also on character n-grams, which allow it to handle previously unseen words and also better incorporate some syntactic structure of word into embedding.

An n-gram is a contiguous sequence of n items from a given sequence. N-grams are commonly used in many NLP methods as well as other domains processing sequences as DNA or protein sequencing.

FastText works with idea that there is some semantic information hidden in syntax of word. Embedding, for word not contained in training corpus, can then be created from n-grams or shorter words. For example imagine there is no embedding for word *going* but word "go" is known and also 3-gram *ing*

was learned from other words. Then aggregation of embeddings for *go* and *ing* should be good approximation of *going*, if original embeddings contained correct semantic information.

Another common task in NLP is to create embedding for larger sentence or other longer parts of text. Depending on used language model it could be tricky task to appropriately aggregate embeddings of multiple words. But it is relatively simple for *fastText* since it inherits *word2vec* arithmetic friendly encoding of information into vectors. Library provided by *fastText* also provide some additional methods for common tasks like this. Method get_sentence_vector simply divide each word vector by its L2 norm and then use average for aggregation.

2.4 Anomaly detection

After feature extraction, several machine learning models can be used for anomaly detection. There are two main parameters that can split models to categories. Capability, to directly process and learn from sequential data, have already been mentioned above. And since logs are inherently sequential these models can efficiently used as alternative to more traditional anomaly detection methods. But much more important and generally used categorization of machine learning method is separation to supervised and unsupervised methods.

Supervised methods require labeled data for training. It can be time consuming and expensive to obtain training dataset of sufficient size and quality. But in return supervised method are more robust since they learn required concept directly thanks to feedback obtained from labels. Anomaly detection with labels is essentially problem of binary classification. *Logistic Regression*, *Decision Tree* and *Support Vector Machine* (SVM) where compared with other unsupervised methods in [1].

Unsupervised methods cannot relay on labels and it is challenging to find how to learn required concept. In anomaly detection common approach is to learn normal state from provided data and detect anomalies as deviations or outliers. In [1] *Clustering*, *PCA* and *Invariant Mining were compared. Recurrent Neural Networks* (RNN) have been used in [13, 18, 20]. And several other types of neural networks might be used. *Time convolution networks* are examined in [21] as promising alternative RNN for sequence processing.

Problem analysis

Existing anomaly detection algorithms used on logs have many limitations. Most of them relay on preprocessing logs to log keys. This approach brings simple and quite successful way how to transform text information in logs to numerical values which are required by all sorts of anomaly detection algorithms. But it also implies several limitations.

Each log key actually creates class of logs and as result algorithms are trained on predefined set of classes. But this set is changing as systems are updated over time and most algorithms are not capable to work with unknown classes. Some solutions for this situation were proposed in [17] but it is still recommended to retrain model periodically to properly incorporate new log keys.

Another problem can rise from errors introduces in preprocessing stage. While idea of log key is simple, their extraction form raw text is not. Algorithms which can parse them are complex and have their limitations in precision of detecting log keys. Comparison of parsing tools in [2] shows that available algorithms are improving, but no of them is perfect.

Log key parsers usually return rich information about given log statement. Log template and extracted parameters are provided in addition to enumerative id of log key. But most of existing anomaly detection algorithms do not use this information. Many even aggregate data from several logs in time or session windows and losing sequential information which is part of original logs. It is surprising how good results can be achieved with aggregated information. And it rises a question how much better results could be achieved if more information is used. Parameters in logs often provide key insight to system state when examined by human. Complex processing of parameters is presented in [20], where separate model is trained for each parameter of each log key. And additional semantic information are extracted from log template in [17]. In both cases additional information improved performance.

Also several other features and characteristics have to be considered when comparing anomaly detection algorithms. These characteristics do not affect precision or reliability but they are important aspects when deployed to

production.

Some use cases can benefit from online algorithms. This is especially true in some security domains where such behavior might be even required to stop ongoing attacks in time. Online algorithm is able to provide results on per log statement basis and have to be fast enough to process incoming log. Note that required bandwidth can vary extensively depending on application.

Modern detection systems usually include some form of supervised machine learning and need some training data which is another common obstacle for deployment. Obtaining good labeled dataset for training require lot of time and domain specific knowledge. Unsupervised methods removes this need for labeled training data. This can significantly reduce time and effort needed for deployment, even thought some small labeled data might still be required for validation.

This thesis explore new possibilities of applying recent advancements in NLP domain to log anomaly detection. More specifically it tests whether advanced NLP embedding approaches can be used to model logs which do not contain typical natural language but it is human readable text. And if such representation is suitable for building unsupervised online log anomaly detection system which use rich information provided in logs.

Chapter 4

Proposed architecture

First task is to create log representation suitable for further processing by machine learning. This representation uses NLP embedding to keep information contained in log message and parameters. Structure of representation and reasons why NLP embedding is used are described in 4.1.

Then unsupervised anomaly detection model based on LSTM is presented in 4.2. Model is trained to predict next log embedding on loges generated by normal system operation. Distance between predictions and real logs is then measured and compared with threshold to detect anomalies. Overview of whole system is shown in Fig 4.1.

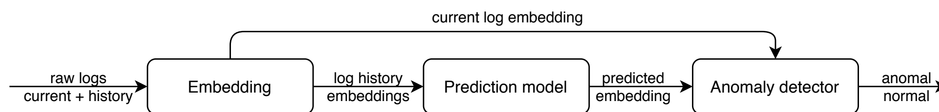


Figure 4.1: Data flow in proposed architecture

Supervised anomaly detection model was also build to check if information needed to distinguish anomalies are included in created representation. Labels allow to train model directly on anomaly detection and model can learn which features are important for it. This make supervised model easier and more reliable to train. Supervised model is described in detail in 4.3.

4.1 Embedding

Embedding encodes information contained in text to numerical values usually vectors. Different embedding types are used in different domains and applications. Pre-trained word embeddings as word2vec from [14] are common in NLP domains. Log keys are often used in log analysis and anomaly detection.

In this thesis combination of NLP sentence embedding of log line and additional handpicked features is used to create final embedding. Enriching embedding with additional custom features is needed, because some contextual

information might not be included in single log line. One such feature is time delta from previous log, which provide significant information when detecting performance anomalies, as shown in [20].

Log keys and some other NLP embeddings consider only fixed and relatively small vocabulary. This is fine for many applications in natural language if we add embedding value for "unknown" word. It is expected that words outside of vocabulary are rare so their occasional appearance will have small effect on application. But this is not entirely true for logs. Software updates cause changes in set of log keys, which discussed more in [3] and then there is another problem with parameters.

Some parameters are numeric and can be simply added as feature to embedding vector. But rest of them is part of text usually with given syntax and semantic analogical to words in natural language. Common parameter types are ip address, file path, time or date. Such parameters are usually not included in vocabulary, because it is simply not possible to have pre-computed embedding for each file path or ip address.

Interesting idea for solving the "unknown" words is presented in *fastText* [15] and already described in Section 2.3.3. Idea behind *fastText* allow not only solving the "unknown" words, but also tries to exploit semantic information hidden in syntax and word structure. Such approach might work well also on already mentioned log parameters as ip addresses or paths, which have clear syntactic structure and semantic.

Even when some numerical representation of parameters is obtained adding them as new features to embedding is also problematic. Most machine learning algorithms expects inputs with fixed dimension. But each log key can have different number of parameters of different types. Baseline approach for parameter representation, presented in [20], showed that it is possible to include all features from each log key and leave unused features empty. Such approach, similar to one-hot vector, results in large sparse inputs and consider fixed set of log keys.

Need of fixed size representation for variable length text is not new in NLP. Common approach uses pre-computed word embedding and aggregation function. Depending on characteristic of selected embedding, aggregation can be as simple as average or in NLP domain it's often TF-IDF, which is variant of weighted average. *FastText* API include build-in method for sentence embedding, which firstly divides each word vector by its norm and then use simple average as aggregation.

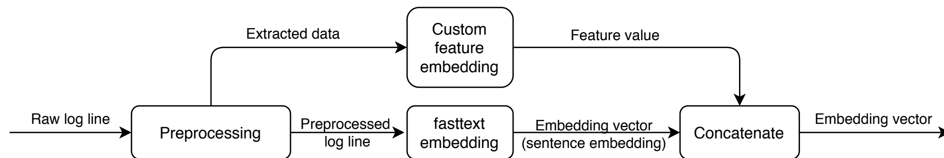


Figure 4.2: Data flow in embedding

Final embedding is concatenation of *fastText* sentence embedding and custom handpicked features as shown in Fig 4.2. Raw logs firstly go through preprocessing phase where data for *fastText* and custom features are extracted. It is obvious that custom features do not need whole log as input data. But *fastText* input is also modified. Some parts of raw log, especially in header, are stripped to reduce noise because they are irrelevant or hard for NLP methods to interpret. Irrelevant can be pid or other identifiers since this thesis focus only on processing of event sequences from one source. Example of value difficult to interpret is already mentioned time-stamp which have many different formats across different applications. But it is same within one dataset (application) and it is present for each log. So it can be easily parsed and added to embedding in much more meaningful representation as custom feature.

preprocessed data continue to *fastText* and custom feature to be transformed to numerical vectors in the next step. *FastText* part is straight forward. Build-in sentence embedding method is used with custom trained *fastText* model to compute embedding of preprocessed log line. Implementation custom features can vary based on characteristic of given feature. Common steps will probably include parsing extracted text data to appropriate format, obtain contextual information saved from previously seen logs (time delta) or from external knowledge about system and then use current value and contextual information to compute some meaningful numeric representation.

Last step is concatenation of *fastText* embedding and outputs of all custom features to one vector. Final embedding have fixed dimension for all logs because all custom features should be general and able to provide values for each log, not only for some subset e.g. one log key.

4.2 Unsupervised model

The goal is to create unsupervised anomaly detection. Embedding works on per log basis thumbs model needs to accept sequential data on input. Common approach in sequence analysis is to train model on normal data to predict next item of sequence. Prediction is then compared with real values to determine if current value is anomalous. This high level approach was used in [20, 17, 22].

Let $S = (s_0, s_2, \ldots, s_t)$ be sequence of embeddings corresponding to log lines indexed by time, where s_t represent embedding value of last received log. Then input for prediction model is sequence $H_{n,t} = (s_{t-n}, \ldots, s_{t-1})$, which represents history of length n logs preceding s_t. And output of model $\hat{s}_t$ is prediction of s_t.

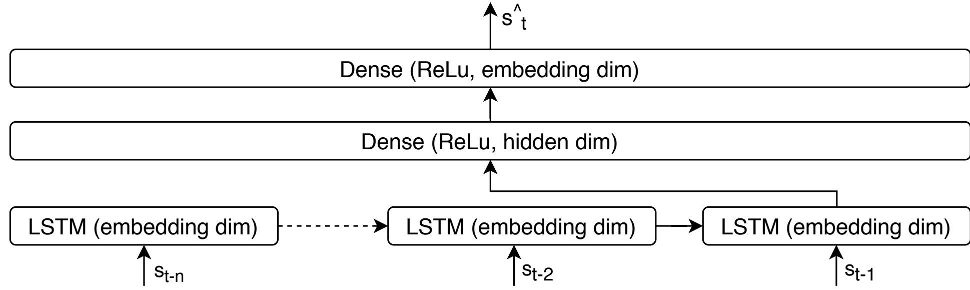


Figure 4.3: Structure of LSTM based prediction model

Model, shown in Fig 4.3, is composed of LSTM layer for sequence processing, and additional dense layers on top of it. LSTM is widely used recurrent neural network (RNN) architecture that has been proven to robustly process sequential data. Idea is that LSTM layer will extract relevant features from sequence. While following dense layers with ReLU activation function provide additional non-linearity and regression when computing prediction. Forward networks, as dense layers, are easier to train then RNN (LSTM), which include some non-linearity itself but they are expensive to train. So similar approaches with smaller RNN for sequence processing and following forward layers are common in practice.

Experiments with different number of layers and sizes of hidden dimension are described in Section 5.2. But dimension of input and hidden state for LSTM as well as output dimension of last layer are equal to embedding size.

Resulting model is trained on log sequences from normal system behavior to minimize distance between prediction $\hat{s}_t$ and real value s_t. But it is important to note that $\hat{s}_t$ and s_t are vectors of relatively large dimension. So selecting suitable distance metric might be problematic. MSE is commonly used in machine learning for smaller dimension. And angular based metrics as cosine distance are considered to be more stable in higher dimensions. And L1 norm or so called fractional distance metrics are proposed in [23] where problem of distance metrics in high dimensional space is explored more in detail. However MSE have proven to work best, after few experiments with above mentioned metrics. Even in our case with relatively high dimensions ranging from 100 to 300.

To detect anomalies prediction model is supplied with $H_{n,t}$ and distance between its output $\hat{s}_t$ and s_t is measured using same metric as in training phase. Current log is label as anomalous if distance is greater then threshold. Optimal setting of correct threshold is a hard problem and it is beyond scope of this thesis. Some sophisticated solutions, like dynamic shareholding from [22] exists, but no out of box implementation is available. So simple threshold, computed as confidence interval from errors on training data, is used in this thesis.

4.3 Supervised model

There is one significant difference between this thesis and other papers which uses unsupervised learning with the prediction model. In most cases prediction of class is considered, which means prediction is limited to finite and relatively small number of options. But in this thesis prediction is point in space with high dimension, which makes prediction significantly harder. This is reason why supervised anomaly detection model was also designed. Supervised model uses labels to directly learn problem of anomaly detection and so it is used to prove, that information needed to distinguish normal and anomalous logs is included in embedding. Anomaly detection with labels is problem of sequence classification with two classes.

Let $L = (l_0, l_1, \ldots, l_t)$ be sequence of labels corresponding to S. Where l_i is 1 for anomaly and 0 for normal. Then input for classification model is sequence $S_{n,t} = (s_{t-n}, \ldots, s_t)$, which represents s_t and history of length n logs preceding it. And output $\hat{l}_t$ is probability of s_t being anomaly.

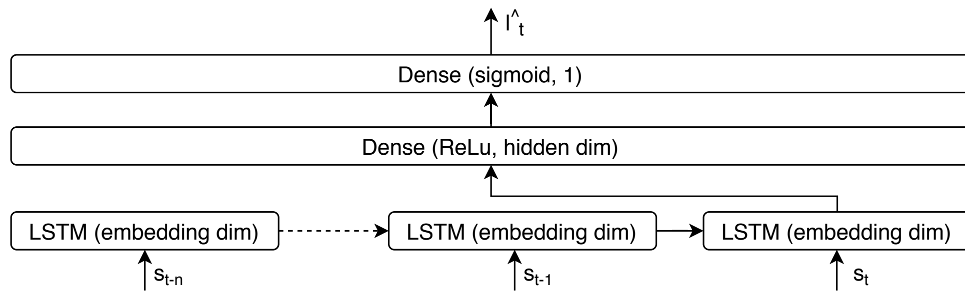


Figure 4.4: Structure of LSTM based classification model

Classification model shown in Fig 4.4 is very similar to prediction model described in Section 4.2. Only difference is that output dimension of last dense layer is 1 while in prediction model it is matching embedding dimension. And sigmoid activation function for this last layer is used.

Binary cross entropy loss is used to train this model because it is standard and proven to be robust when training binary classification models.

Chapter 5

Implementation

Architecture proposed in Chapter 4 was implemented in Python[1] 3.7 language. There are several reasons why Python has been chosen. It is popular language for machine learning and data analysis, so many libraries exist for common tasks and method in this domain. *PyTorch*[2] is used for building LSTM based anomaly detection models. PyTorch is an machine learning library used for applications such as computer vision and natural language processing. It is free and open-source software released under the Modified BSD license. Also fastText[3], as open-source library for efficient learning of word representations and sentence classification, provides Python binding, so it can be comfortably called from Python. Finally there are log base anomaly detection methods already implemented and open sourced, which allow for easy benchmarks.

Comparison of six anomaly detection methods is presented in [1] and implementation of these methods, as well as benchmark scripts, are publicly available in *Loglizer*[4] project on GitHub under MIT License. Loglizer is used to compare experimental results with other anomaly detection methods.

Implementation include some deviations from originally described architecture. Changes have been made to make implementation more efficient in experiment setting, or simpler to compare with other methods. All such changes will be explicitly stated in following sections, which describe separate parts of implementation and challenges they posed.

[1] https://www.python.org/
[2] https://pytorch.org/
[3] https://fasttext.cc/
[4] https://github.com/logpai/loglizer

5.1 Preprocessing and benchmarks

First change of architecture is in the very first step of preprocessing. Loglizer and its benchmark script implements, not only anomaly detection methods and their evaluation, but also whole pipeline of data loading, preprocessing and splitting to training and testing datasets. Log preprocessing for proposed models is implemented as modification of data loader in Loglizer, to simplify implementation and also ensure that exactly the same datasets will be used in experiments. This implementation extract and save required preprocessed data during benchmark run for later use in our experiments. This saved data also allow to run multiple experiments with different parameter settings, without the need to always recompute the preprocessing.

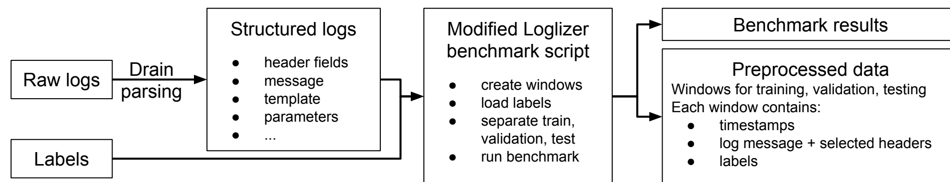


Figure 5.1: Process of data preprocessing and benchmark

Loglizers benchmark script expect structured log data on input. Open-source implementation of Drain parsing tool, provided in Logparser[5] project on GitHub, is used to parse raw logs to structured data. Drain has been chosen, among parsing tools provided by Logparser, because it is currently the best parsing tool, according to benchmarks and comparison in [2].

To obtain preprocessed data *dataloader.py* file in root of Loglizer project is modified. All methods in benchmark are using windows, so logs are firstly loaded and split to appropriate windows. For HDFS dataset session windows, based on block ID, are used and sliding time windows are used for BGL dataset. In this part original implementation stores only log keys from structured logs. Additional data structure is added to store also timestamps (for time delta custom feature) and strings for fastText embedding composed from log level, component and log message. After windows are prepared, labels are loaded for each window by original script. And modification is made so labels are also copied to the new data structure. Then windows are split to training and testing datasets. After that training dataset is further split to training and validation by our modification. All resulting datasets (training, validation and test) are saved to file, in a way that label, timestamps and strings for fastText embedding are included.

Two files are actually saved. This is caused by different requirements of supervised and unsupervised models. Copy for unsupervised model have filtered out anomalous samples form training and validation datasets. Data

[5]https://github.com/logpai/logparser

in both files are stored as dictionary with three entries, one for each dataset (training, validation,test). Python pickle module is used for data serialization.

Splitting data to windows is not required by models proposed in this thesis, since they are based on LSTM, which can operate in stream fashion. But sessions in HDFS dataset are actually parallel processes and their logs are intertwined. This thesis does not consider task separation of intertwined processes, but some articles like [24] focus on this problem. However HDFS data can be easily unwinded using block IDs, which is the same ID used when creating windows. It is good to keep already unwinded windows, since it can cause problem for LSTM based sequential models to process intertwined streams. Easier comparison using the same evaluation as benchmarks is another benefit of keeping data separated into windows.

5.2 Models

Implementation details of models will be described here before embedding. Because implementation of models defines additional requirements for type and shape of input data, which are not obvious from high level view of architecture described in Chapter 4. That is why some data transformations made during embedding and data formatting would be confusing without knowledge of the exact input and output definitions.

Architecture considers, that logs are streamed one statement after another and describes data flow on example of processing one log statement with some available history. Such approach is valid but not efficient in training phase when all logs are already available. PyTorch implementation of LSTM layer works by default as sequence-to-sequence. In this mode LSTM accepts sequence on input and returns another sequence of the same length, where i-th item of the resulting sequence corresponds to LSTM output after i items where processed. Using this mode, models output for each log statement within one window can be computed more efficient in one step.

Sequence-to-sequence mode is also used when trained model is used for anomaly detection. In real live data monitoring scenario, this change would require some sort of batch processing, resulting in lost of online detection ability. But it is acceptable in experiment setting, where all data are available before test. Simpler implementation, which reuse some code, is possible, when using the sequence-to-sequence mode in both training and evaluation phases. It also brings improved performance for training and evaluation cycle.

Both models supervised and unsupervised have many parameters set their exact size, learning rate, normalization etc. Parameters are passed as command line arguments, when creating new model. List of all available parameters is included in Appendix B.

Two normalization methods were implemented. Gradient clipping is method used to limit maximal weight change in one step. This can make learning

more stable and prevent so called exploding gradients, which is common phenomenon with recurrent neural networks, such as LSTM. Second normalization is option to include additional layer normalization[6], as defined in [25], in between layers.

Adam optimization is used for training. Initial learning rate is set to PyTorch default, but can be changed via parameter. During each epoch, gradients are computed from training data and back propagated to update weights, then loss over validation data is computed. Validation loss can be used to watch for over fitting. Number of epochs to compute is given as parameter and no smart termination condition depending on validation loss is implemented.

But model is saved after each epoch, as well as information about training and validation loss. Training can be resumed from last saved model, if training was interrupted or initial number of epochs was insufficient. Reference to the epoch with best validation loss is kept, but model from any epoch can be used for evaluation and anomaly detection.

Threshold is used in evaluation to determine if sample is normal or anomalous. Supervised model have sigmoid function on its output and resulting value represent probability of sample being anomalous. Default threshold is set to 0.5, which is reasonable assumption given the sigmoid function, but it can be fine tuned by parameter.

Situation is a little bit more complicated with unsupervised models. Decision about anomalous samples is made base on error between prediction and real value. Setting threshold on error by hand is a bad idea since error values ranges are different for each model and input data. As already mentioned in Section 4.2, there are more sophisticated methods like dynamic thresholding from [22]. But decision was made to use simpler anomaly detection and focus more on embedding part of the problem. Threshold based on standard deviation is computed during training in each epoch based on errors computed from training data using following formula.

$$t = E(errors) + 2std(errors)$$

That means about 5% of logs will be labeled as anomaly, with assumption that prediction errors follow normal distribution.

In addition to sequence-to-sequence mode PyTorch also works with batches. Batch is a common concept in neural network learning and it is used in most frameworks and libraries. Batches improve stability and often also efficiency during training phase.

Let e_{ij} be embedding vector of jth log statement in ith window and $\hat(e_{ij})$ donates prediction of such embedding. Let $\hat{l_{ij}}$ be estimated probability, for jth log statement in ith window, to be anomaly. Then Figure 5.2 illustrates

[6] https://pytorch.org/docs/stable/generated/torch.nn.LayerNorm.html#torch.nn.LayerNorm

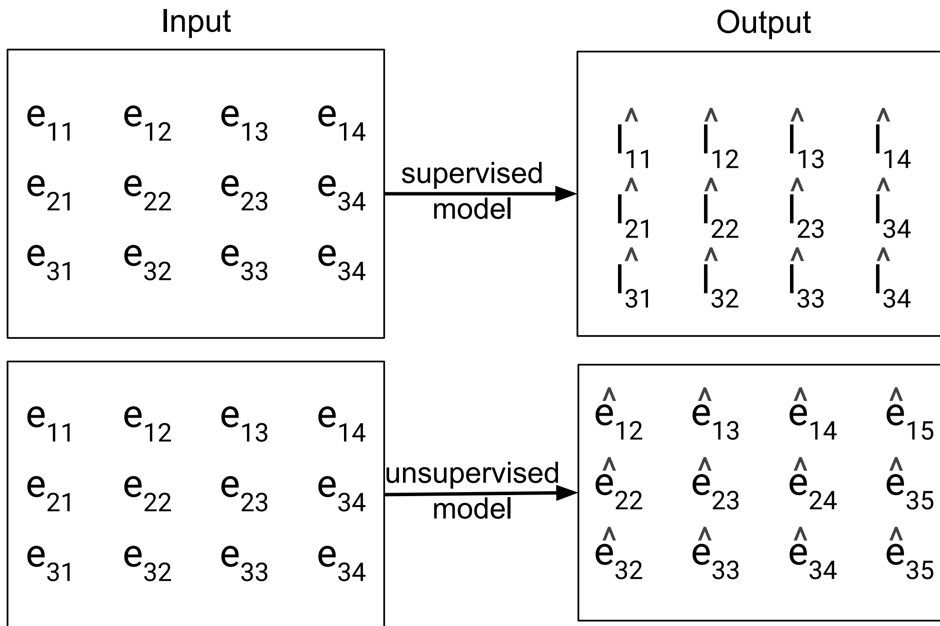


Figure 5.2: Input and output data for models

input and output batch format for supervised and unsupervised models. It shows example of batch containing 3 windows with 4 logs each.

All log statements in window are processed in one step thanks to sequence-to-sequence mode and multiple windows are put in one batch. Input is tensor with 3 dimensions (window, log, embedding feature) and have the same shape for supervised and unsupervised model.

Output of supervised model are estimated probabilities of log statements, to be anomaly. There is $\hat{l_{ij}}$ for each log in window, because sequence-to-sequence is used. So the information is 2 dimensional (window, log), but it is shaped as 3 dimensional tensor (window, log, 1), to have same output dimensions as unsupervised model. Making it compatible with same evaluation method.

Output for unsupervised model is prediction of next embedding in sequence. Thanks to batches and sequence-to-sequence output is a tensor with 3 dimensions (window, log, embedding feature). It is a tensor with the same shape as input data. But prediction causes logs in windows to be shifted by one step to the future, as shown in Figure 5.2.

Labels are provided, in addition to each input, output pair, as 2 dimensional array (window, log). This information is redundant for supervised model, but it is passed anyway to unify code for evaluation.

5.3 Embedding and data management

PyTorch library provides classes for common data loading and management tasks in *torch.utils.data* package. Two such classes are used, *Dataset* class is extended to load preprocessed data, in format described in Section 5.1, and compute embedding using provided transformation function. And *DataLoader* class is used for creating batches and shaping data to format required by models.

Purpose of Dataset class is to load and provide samples on demand for DataLoader. Preprocessed datasets are stored in file with same format for supervised and unsupervised experiments, but exact input and output format differ. To make implementation of Dataset reusable it requires transformation as parameter in constructor. Transformation is callable object which takes preprocessed sequence (window), computes embedding and uses it to create sample (inputs, expected outputs and labels), required by specific model, for one window.

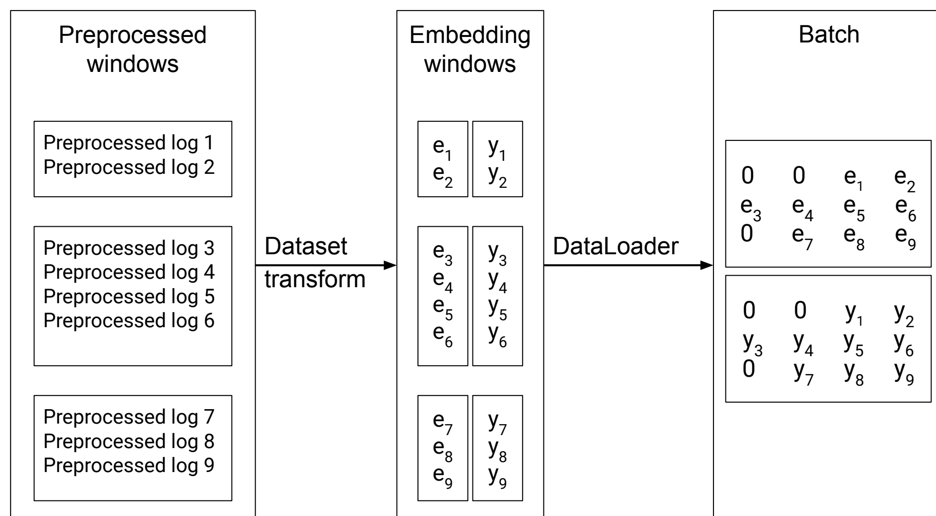


Figure 5.3: Process of data loading and padding in batches. e_t represents embedding of log t and y_t is expected output for given log.

Figure 5.3 illustrates how samples are loaded from by Dataset and then grouped by DataLoader to batches. Batch is triplet input, output and labels. Windows in one batch must be padded to have the same length, since both inputs and outputs are tensors. PyTorch provides some basic methods for sequence padding but it still needed some work to properly pad whole triplet.

Values y_i, in Figure 5.2, represents expected output corresponding to log i. But output differs, for supervised model it is label l_i, but for unsupervised model it is next embedding e_{i+1}.

Implementation of fastText embedding is straight forward. It uses python binding provided by fastText to access its binary library. Method *get_sentence_vector* is called on each preprocessed log line. Inner working of fastText are summarized in Section 2.3.3.

Then time delta custom feature embedding is computed and added to fastText embedding. Firstly time differences are computed form preprocessed timestamps. This raw difference is numeric value, but it has wide ranges causing numerical instability. Logarithm was used to reduce large values, when system is inactive for some time. After logarithmization values were then normalized. Embedding value for one raw time difference t can be computed by following formula.

$$timeDeltaEmbedding(t) = \frac{log(t) - \mu_{train}}{\sigma_{train}}$$

Were μ_{train} and σ_{train} are mean and standard deviation computed on training data.

Chapter 6

Experiments and evaluation

Multiple experiments were prepared and executed to verify hypotheses and evaluate proposed solution. Computation heavy tasks were executed on computation cluster provided by Research Center for Informatics[1].

This chapter firstly describes used datasets in Section 6.1. Then evaluates suitability of fastText models for embedding whole log statements including parameters is tested in Section 6.2. Both supervised and unsupervised variants of proposed architecture are evaluated and compared to other methods in benchmark in Section 6.3. And finally summary of findings is presented in Section 6.4.

6.1 Datasets

Two publicly available datasets, used in this thesis,were downloaded through LogHub[2], which is project on GitHub provided by authors of [2]. HDFS and BGL datasets were chosen from available ones, because both were already studied in multiple articles and benchmark script from Loglizer implements loading of HDFS data with labels and provide partial implementation for loading BGL. Basic summary of datasets is shown in Table 6.1.

[1]http://rci.cvut.cz
[2]https://github.com/logpai/loghub

	BGL	HDFS	HDFS_2
Data size	1.55 G	708 M	16.06 G
Labels	by log line	by block (session)	no
#Log lines	4,747,963	11,175,629	71,118,073
#Templates	619	30	-
#Windows	3132 (by time)	575061 (by block ID)	-
Anomalies	20.78% blocks (7,34% lines)	2.93% blocks	-

Table 6.1: Summary of datasets

6.1.1 DHFS

HDFS stands for *Hadoop Distributed File System*[3]. LogHub provides two parts, or in fact two separate datasets, for HDFS. Smaller part is labeled dataset which was originally presented in [26]. Description of this part from Loghub project:[4]

> This log set is generated in a private cloud environment using benchmark workloads, and manually labeled through handcrafted rules to identify the anomalies. The logs are sliced into traces according to block ids. Then each trace associated with a specific block id is assigned a groundtruth label: normal/anomaly (available in anomaly_label.csv).

Second larger part are unlabeled HDFS logs collected by LogHub authors in labs of The Chinese University of Hong Kong. This huge dataset (over 16GB) consist of logs from one name node and 32 data nodes.

Example of one HDFS log statement is shown below. HDFS logs have standard header composed from date, time, pid number, log level and component. Log message is then simple English sentence with some parameters in human readable form.

```
081109 203645 175 INFO dfs.DataNode$PacketResponder:
Received block blk_8482590428431422891 of size 67108864 from /10.250.19.16
```

Large unlabeled dataset is used for training fastText language model. And smaller labeled dataset is used for anomaly detection training and evaluation. Unfortunately labels are provided only on block level, as already mentioned in description above. This suggest creation of windows containing logs from one block. Such windows are essentially session windows, and differences are only in HDFS terminology. There is 575061 windows in total, when split by block ID. With 16838 labeled as anomaly, which makes about 2.93% windows in dataset. Figure 6.1 shows histogram of window lengths, which varies from 2 to maximum of 298 log statements, with mean 19.43 and median 19.

[3] http://hadoop.apache.org/hdfs
[4] https://github.com/logpai/loghub/tree/master/HDFS

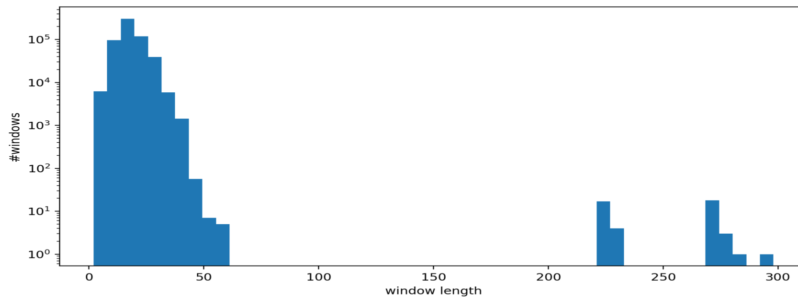


Figure 6.1: Window length distribution HDFS

6.1.2 BGL

BGL dataset was originally presented in [27]. Description from Loghub project: [5]

> BGL is an open dataset of logs collected from a BlueGene/L supercomputer system at Lawrence Livermore National Labs (LLNL) in Livermore, California, with 131,072 processors and 32,768GB memory. The log contains alert and non-alert messages identified by alert category tags. In the first column of the log, "-" indicates non-alert messages while others are alert messages. The label information is amenable to alert detection and prediction research. It has been used in several studies on log parsing, anomaly detection, and failure prediction.

BGL logs have richer header with some redundant fields. Header fields are UNIX timestamp, human readable date, job id, human readable time to μs, another job id, user, group and log level. Log messages themselves are a bit more cryptic when compared to HDFS. Some messages are still mostly English sentences as first log in example below. Other messages, as second line in example, are more dense and often use hexadecimal parameters values and other not human friendly formats.

```
- 1117840356 2005.06.03 R16-M1-N2-C:J17-U01 2005-06-03-16.12.36.079052
R16-M1-N2-C:J17-U01 RAS KERNEL INFO total of 31 ddr error(s) detected and corrected

- 1117840759 2005.06.03 R25-M1-N7-C:J17-U01 2005-06-03-16.19.19.369025
R25-M1-N7-C:J17-U01 RAS KERNEL INFO CE sym 7, at 0x10d85460, mask 0x80
```

BGL is labeled by log statement with $348k$ anomalous logs out of 11M logs in dataset. Dataset was split in time window, because other methods in benchmark require windows. Labels by line are preserved within windows but additional label for whole window is created. Window is considered anomalous if it contains one or more anomalous statement. Length of time window used is 60 hours, which is proposed default values in BLG loading

[5]https://github.com/logpai/loghub/tree/master/BGL

method in Loglizer benchmark script. It resulted to 3132 windows, with 2481 normal and 651 labeled as anomalous. Number of logs in window varied extensively from some empty windows which were discarded to maximum of 184265 log statements. But most windows contain reasonable number of logs since mean is 1504.73 and median is 78. Figure 6.2 show histogram of window lengths for BGL dataset.

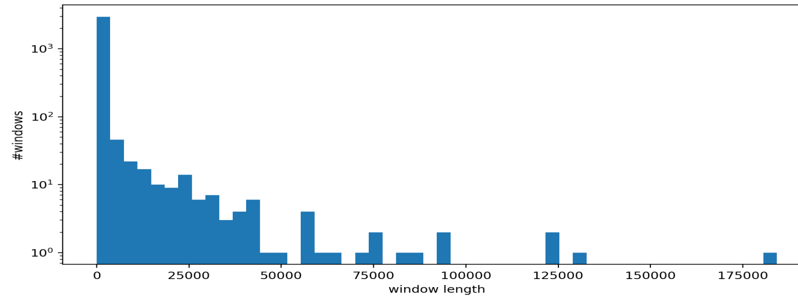


Figure 6.2: Window length distribution BGL

6.2 Embedding analysis

Custom fastText models were trained for log embedding on each dataset. Large unlabeled part of HDFS dataset was used for HDFS model and whole BGL dataset for BGL model. Additionally several parameter combinations were tried for each dataset. FastText has three parameters which directly affect resulting embedding. Embedding dimension with default value 100, and bounds for n-grams lengths with default range 3-6. Two embedding dimensions (50, 300) and n-gram range 1-1, were used in addition to default fastTest values. So total of 6 models with different parameter combination were trained for each dataset.

Embedding values were computed for selected sample of log statements containing 11 and 12 log templates for HDFS and BGL respectively. Hypothesis is that log statements belonging to the same template will create clusters. In theory, separated clusters allow LSTM based models extract information about template, while variance within cluster represents different parameter values.

All embedding models use high dimensions which cannot be examined directly, so methods of dimension reduction are used for visualization. PCA and t-SNE were tried as they are common and popular methods for dimension reduction. Presented figures use t-SNE reduction to 2 dimension, because it showed results with cleaner separation of clusters.

Figures 6.4 and 6.5 are visualization of fastText HDFS and BGL embedding models respectively. All 12 fastText models successfully clustered log statements by template. Also more spread clusters mostly belong to templates

with more parameters, or parameters with larger value space. In Figure 6.4, the most spread cluster in HDFS belong to template with 3 parameters:

<∗>Got exception while serving <∗> to <∗>

The same apply also to BGL models in Figure 6.5, where the most spread cluster also belong template with three parameters. It even created several separated clusters close to each other. Multiple clusters are probably caused by combination of relatively short and general template and multiple parameters.

CE sym <∗>, at <∗> mask <∗>

It also seems that some semantic information is passed in embedding, in addition to expected template separation by clusters. Semantic information can be derived from relative positions of clusters to each other. Especially in Figure 6.5 several clusters are close to each other, and relatively separated from other clusters. All these clusters belong to templates reporting some errors:

<∗> ddr errors(s) detected and corrected on rank <∗>, symbol <∗>, bit <∗>
<∗> L3 EDRAM error(s) (dcr 0x0157) detected and corrected
total of <∗> ddr error(s) detected and corrected
ddr: activating redundant bit steering: rank=<∗> symbol=<∗>
ddr: excessive soft failures, consider replacing the card

One anomaly detection experiments on BLG was executed with HDFS fastText model by accident. Surprisingly it did not show radically worse results. This brought up idea of one general language model for multiple systems, which would be pre-trained on large dataset compiled from multiple log types. It is just suggestion for future work, but embeddings computed by 'wrong' model are shown in Figure 6.3, to prove the concept. Log embeddings are still mostly clustered by template, but clusters are closer to each other and less separated.

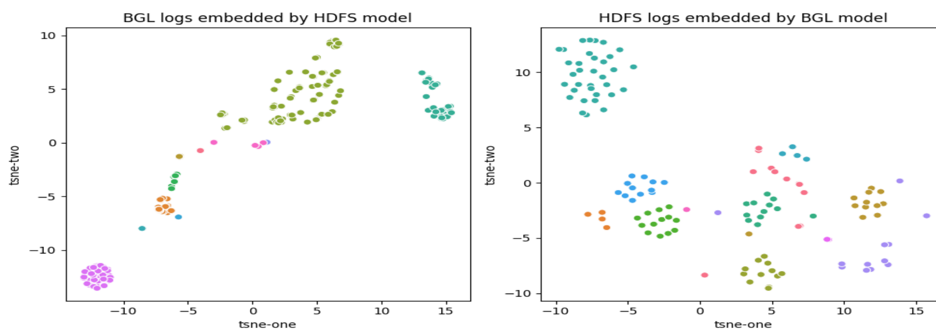


Figure 6.3: Embedding visualization across datasets (t-SNE reduction)

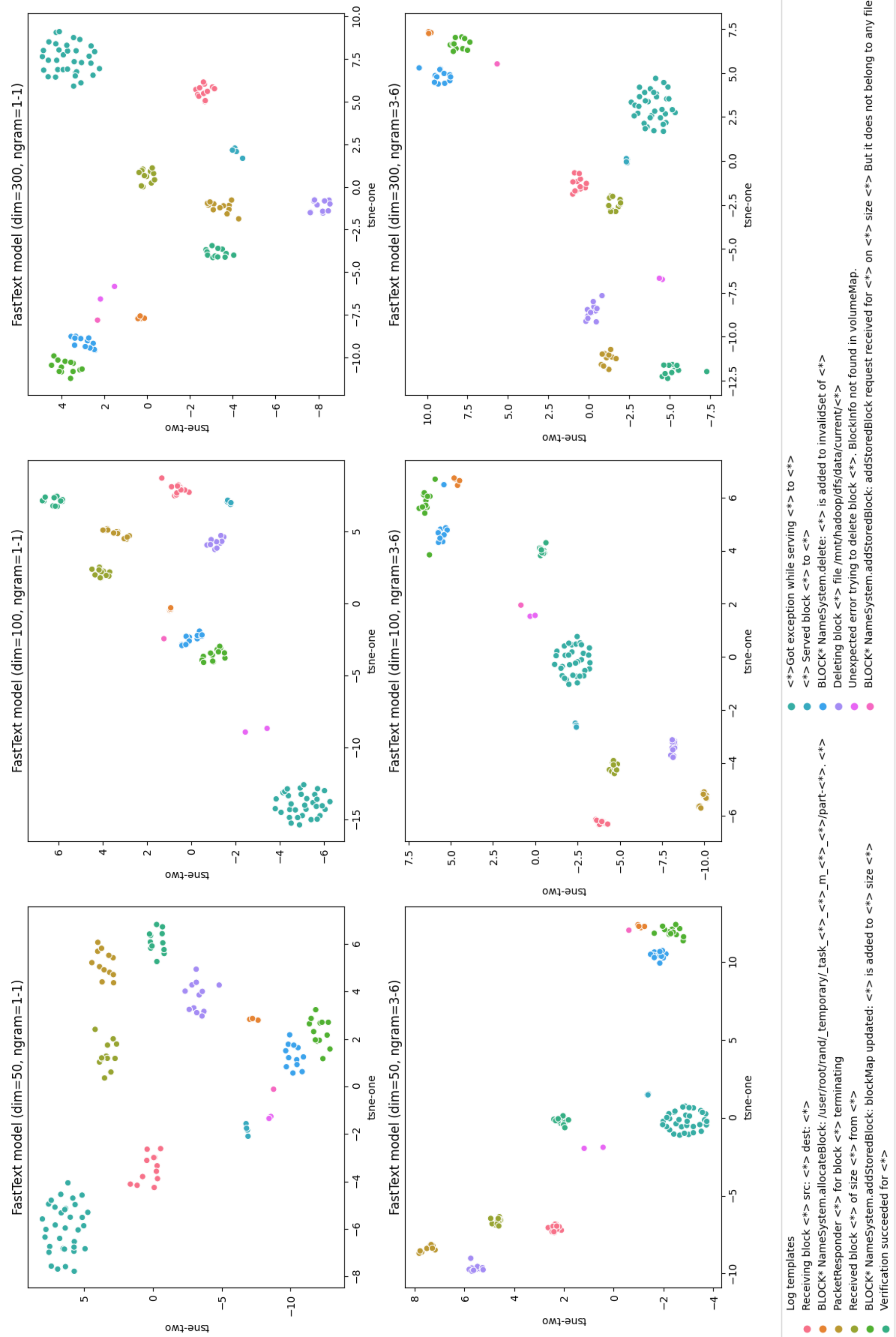


Figure 6.4: Comparison of embedding using fastText models with different parameters (t-SNE reduction, HDFS data)

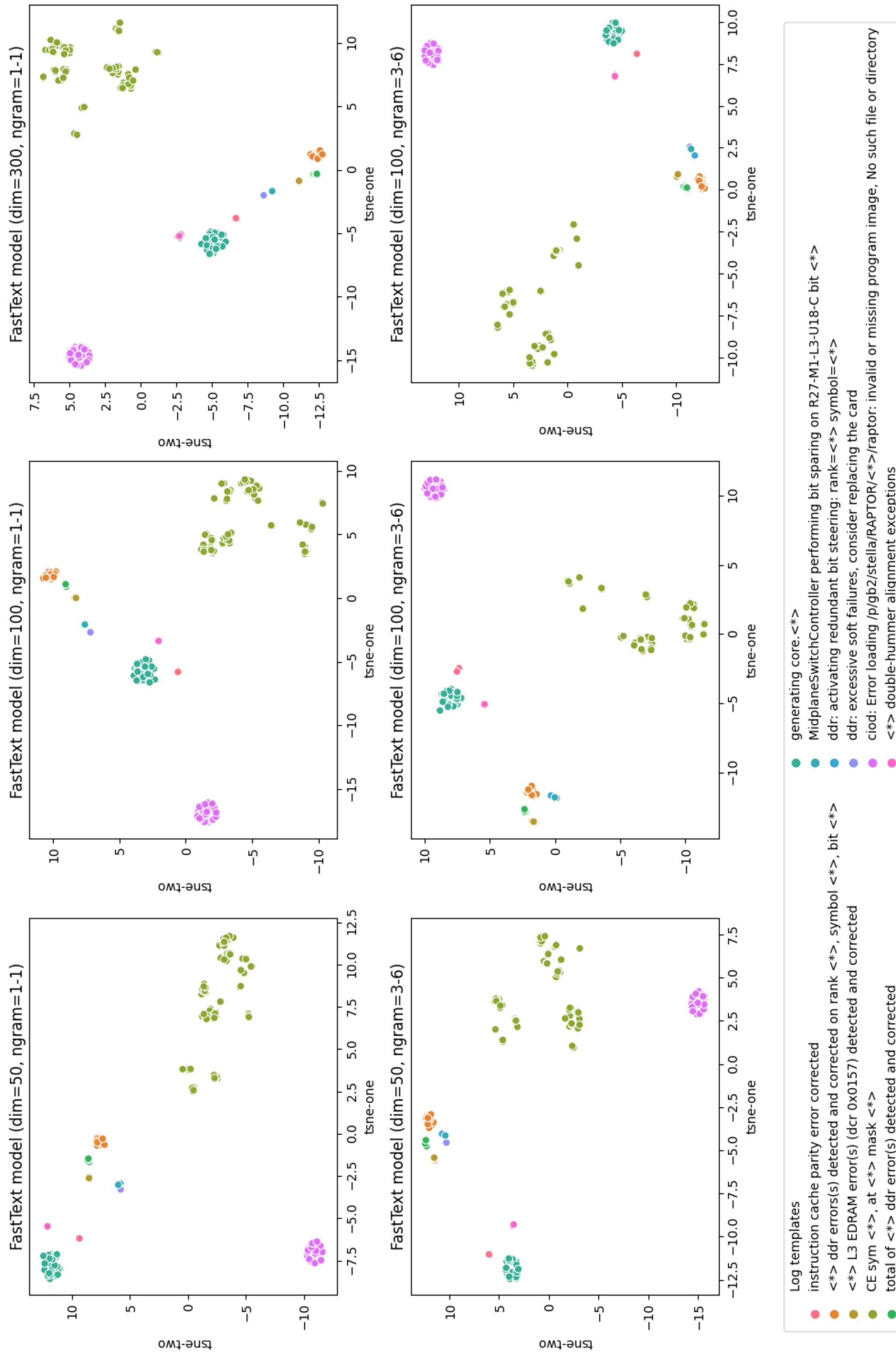


Figure 6.5: Comparison of embedding using fastText models with different parameters (t-SNE reduction, BGL data)

6.3 Anomaly detection

Benchmark script from Loglizer have provided evaluation of multiple anomaly detection methods for both datasets. And multiple experiments with different parameters for supervised and unsupervised model were performed. Only best obtained results are compared with other methods in Tables 6.2 and 6.3. Only general evaluation method and results of other method in benchmark are discussed here. And more detailed examination of experiments and their results is provided in Sections 6.3.1 and 6.3.2 for supervised and unsupervised models respectively.

Other methods in benchmark decide only on window level and HDFS dataset provide only labels on window (block) level. So same approach as in [20] is used to produce comparable results. That is decision about anomality is produced for each log statement in window and then window is labeled as anomaly if one or more statements were labeled as anomaly. For BGL dataset, where labels are available per log statement, evaluation is also provided on per log statement bases to show real results.

Same metrics are used for comparison, whether evaluation is computed on window or log statement level. *Precision*, *recal*l and *F1-measure* are used for comparison, as they are the most commonly used metrics, to evaluate the accuracy of anomaly detection methods. Firstly true-positives (TP), true-negatives (TN), false-positives (FP) and false-negatives (FN) need to be counted, and then metrics can be computed using following formulas:

$$precision = \frac{TP}{TP + FP}$$

$$recall = \frac{TP}{TP + FN}$$

$$F1 = \frac{2 * precission + recall}{precission + recall}$$

Tables 6.2 and 6.3 shows that most methods in benchmark works better on HDFS. Obtained benchmark results are similar to results presented Loglizer project page. *DeepLog* showed significantly worse results on HDFS, then the ones presented in original paper [20]. But it should be noted, that used open-sourced implementation of *DeepLog* is not complete and include only log key models. And no freely available implementation of its parameter models have been found.

Experiments with both supervised and unsupervised models did not beet the state of the art method from benchmarks, but are comparable to some of the methods. Exception is really good result on BGL with log level evaluation, but it uses different evaluation so it cannot be compared directly.

Model	Precision	Recall	F1
PCA	0.9938	0.2656	0.4192
DeepLog (log key model)	0.1312	0.2393	0.1695
InvariantsMiner	0.0756	1.0000	0.1406
LogClustering	1.0000	0.7778	0.8750
IsolationForest	0.0542	0.7001	0.1005
LR	0.8369	0.9994	0.9109
SVM	0.8368	0.9992	0.9108
DecisionTree	1.0000	0.9907	0.9953
Supervised	**0.0448**	**0.8756**	**0.0852**
Supervised*	**0.9467**	**0.9711**	**0.9588**
Unsupervised	**0.1535**	**0.4182**	**0.2246**

Table 6.2: Benchmark results on HDFS.
Suppervised* is modified version with uses sequence classification.

Model	Precision	Recall	F1
PCA	0.2766	0.0992	0.1461
DeepLog (log key model)	0.2402	0.9847	0.3862
InvariantsMiner	0.2311	0.9542	0.3720
LogClustering	0.2222	0.7481	0.3426
IsolationForest	1.0000	0.1298	0.2297
LR	1.0000	0.2977	0.4588
SVM	0.9872	0.5878	0.7368
DecisionTree	1.0000	0.6030	0.7524
Supervised	**0.4030**	**0.9818**	**0.5714**
Supervised (by log)	**0.9414**	**0.9974**	**0.9686**
Supervised*	**0.3359**	**0.9924**	**0.5019**
Unsupervised	**0.2157**	**0.9847**	**0.3539**
Unsupervised (by log)	**0.0014**	**0.0055**	**0.0022**

Table 6.3: Benchmark results on BGL.
Suppervised* is modified version with uses sequence classification.

6.3.1 Supervised anomaly detection

Firstly several supervised models with different number of layers and size of hidden layers were evaluated to figure out reasonable parameters of the model. Number of LSTM layers does not seem to affect accuracy, so it is set to 1. Number and width of following dense layers affected models accuracy. Low values did not provide sufficient number of parameters and model did not learn or underfit the data. On the other hand too large models took significantly longer to train, while not providing any additional improvements. Parameter values that worked well with all sizes of embedding are 3 dense layers with hidden width of 300. These values were used in all following experiments.

Supervised anomaly detection is binary classifier with labeled data on input. Classification, in general, does not work well on unbalanced datasets where some classes are represented less often. But anomalies are by definition rare, so in case of anomaly detection there is a large unbalance within dataset. There are several methods to deal with unbalanced data, since it is well known problem. General approach is to collect more data if possible, or resample data to get more balanced dataset. Then there are options like generation of synthetic samples or use model which can apply different weights to samples in training phase. Weights are used in this thesis. PyTorch implementation of BCEWithLogitsLoss allows to set per class weights which should be applied during training. Used weights for both datasets are 1:30, since there is 30 times more normal logs then anomalous in both datasets.

Then supervised models with different embedding models were trained and evaluated on both datasets. On BGL data models were evaluated twice. Once on log statement level and then on window level. Only results of the best models are shown in Table 6.4.

Model	**Precision**	**Recall**	**F1**
HDFS (dim=100, ngram=1-1)	0.0448	0.8756	0.0852
HDFS* (dim=100, ngram=1-1)	0.9467	0.9711	0.9588
BGL* (dim=100, ngram=3-6)	0.3359	0.9924	0.5019
BGL (dim=100, ngram=3-6)	0.4030	0.9818	0.5714
BGL by log (dim=100, ngram=3-6)	0.9414	0.9947	0.9686

Table 6.4: Results of supervised models.
* means modified model with sequence classification

Results in Table 6.4 shows outstanding accuracy on BGL data when evaluated by log statement. Unfortunately precision drops significantly when evaluated on window level. Drop is probably caused by FP log statements generated by model which fall to normal windows. Because even good precision in per log evaluation still generate about 2k of FP in testing part of BGL dataset, while this part is split to only 628 windows.

Models on HDFS dataset, on the other hand, perform poorly. It seems strange at first, since most methods performed better on HDFS then BGL. But there is one big difference to BGL dataset, labels are provided only on window level. It does not matter to the methods in benchmark which are designed to work with windows. But proposed supervised LSTM based model require per log labels for training. This result in labeling all logs in anomalous block window as anomalous, while most of them probably are not. More detailed examination of model behavior is in the following paragraph, to verify this assumption and eliminate other possible reasons as overfitting or other problem with training.

Firstly training and validation loss are check in Figure 6.6. On first look training loss is normal and validation is too flat. Both losses still slowly decreasing, which might suggest insufficient number of epoch. Even with

increased number of epochs losses did not started behave differently. Flat validation suggests hitting some block in training, which is confirmed by findings in following paragraphs.

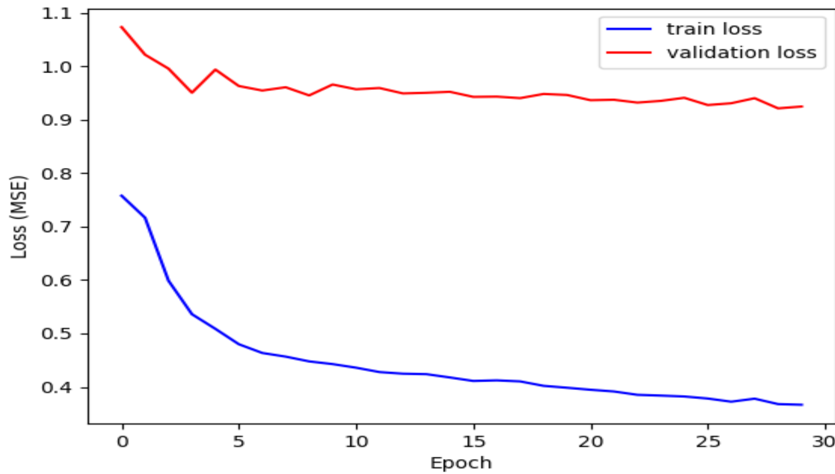


Figure 6.6: Supervised training and validation MSE loss (HDFS data)

Then distribution of model output is examined. Output of supervised model is estimated probability of log to be anomalous. Figure 6.7 shows distribution of estimated probability over logs in normal and anomalous windows. Spike close to zero probability is caused by batch padding. Problem is almost uniform distribution of probability on normal logs. This is caused by many normal logs in anomalous blog windows, which are during the training treated as anomalies. They force model to assign high probability even for normal logs.

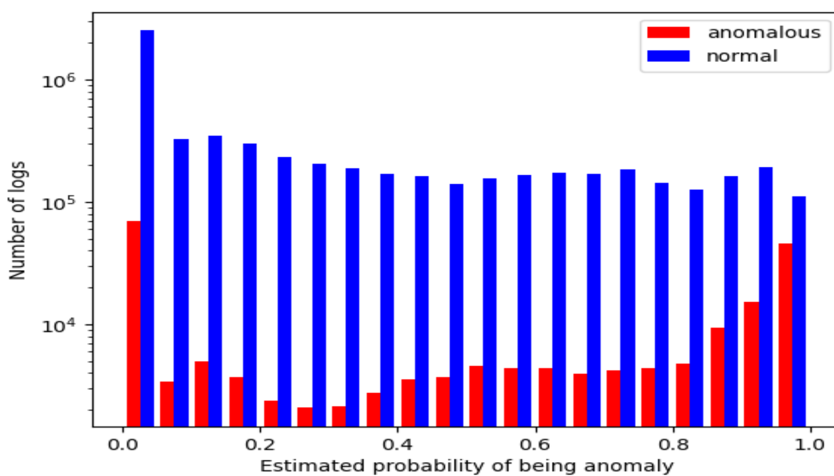


Figure 6.7: Supervised output distribution (HDFS data)

Next problem can be in manually set decision threshold. Default value of 0.5 worked well for BGL, but there might be some better values for HDFS data. Even though it is unlikely, with output distribution shown in Figure 6.7. Effect of different thresholds on metrics is shown in Figure 6.8. Metrics are mostly uniform and not affected by most threshold values. But over all performance can be improved by pushing threshold to extremely high probabilities. With such threshold only anomalies predicted with high confidence will be detected, which partly overcome uniform probability distribution on normal logs.

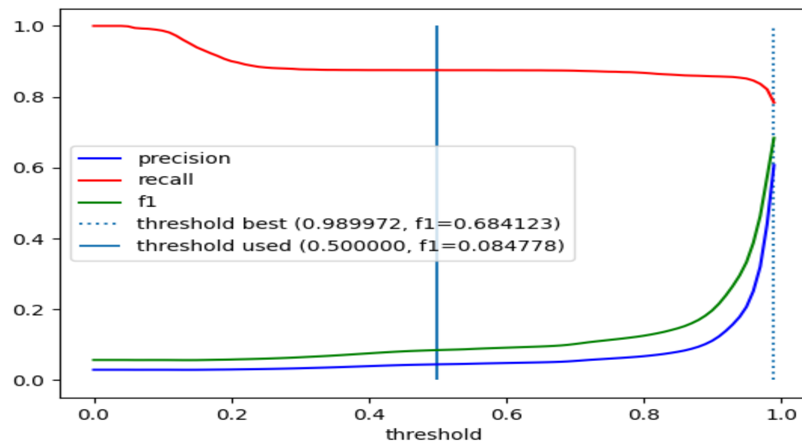


Figure 6.8: Effect of different threshold on metrics (HDFS data)

Modified supervised model was trained and evaluated, to further validate suspicions that bad accuracy on HDFS data is caused by labels. It implements sequence classification instead sequence-to-sequence approach, so it require only labels per window. Results of this models are included in Table 6.4 and marked by '*'. Sequence classification significantly improved accuracy, when compared to original model. It took second place among methods in benchmark on HDFS data, with F1-measure 95.88%. But it was slightly worse then original model, on BGL data.

To summarize results and findings in experiments with supervised methods. Goal of these experiments was to verify, that information required for distinguishing anomalous logs is included in embedding. Results on BGL dataset definitely proven that such assumption can be made. Experiments on HDFS data showed weaknesses of supervised approach. It relies on labels, which need to be provided in sufficient quality and quantity. And that is hard to fulfill in real word use-cases. But sequence classification model showed that good results can be achieved also on HDFS data. So also embedding from HDFS data include information required for anomaly detection.

6.3.2 Unsupervised anomaly detection

Firstly several unsupervised models with different number of layers and size of hidden layers were evaluated to figure out reasonable parameters of the model. Result ot this initial exploration was the same as for supervised model, that means. Number of LSTM layers does not seem to affect accuracy, so it is set to 1. Number and width of following dense layers affected models accuracy. Low values did not provide sufficient number of parameters and model did not learn or underfit the data. On the other side too large models took significantly longer to train, while not providing any additional improvements. Parameter values that worked well with all sizes of embedding are 3 dense layers with hidden width of 300. These values were used in all following experiments.

Then unsupervised model, for each of the embedding variant, was trained and evaluated on HDFS dataset to examine effect of different embedding sizes on accuracy. And two models in combination with default embedding (dim=100, ngrams=3-6) were trained and evaluate on BGL dataset. One evaluated on window level and other on log statement level. Default embedding was chosen as it was the most successful embedding on HDFS dataset. Result of these experiments are in Table 6.5.

Model	Precision	Recall	F1
HDFS (dim=50, ngram=1-1)	0.8044	0.0474	0.0895
HDFS (dim=50, ngram=3-6)	0.1458	0.4908	0.2249
HDFS (dim=100, ngram=1-1)	0.1182	0.4337	0.1858
HDFS (dim=100, ngram=3-6)	0.1535	0.4182	0.2246
HDFS (dim=300, ngram=1-1)	0.6313	0.0474	0.0882
HDFS (dim=300, ngram=3-6)	0.0906	0.3850	0.1466
BGL (dim=100, ngram=3-6)	0.2207	0.9924	0.3611
BGL by log (dim=100, ngram=3-6)	0.0014	0.0055	0.0022

Table 6.5: Results of unsupervised models with different embeddings

Results in Table 6.5 show that embedding size have significant effect on accuracy. In general it seems that larger n-grams are, better the char-grams, which fit hypothesis that n-grams can catch help exploit syntax of parameters like IP address or paths. Dimension of embedding seems to have smaller effect on accuracy then n-gram size. Better performance was expected with higher dimension, since more information can be stored and passed to the model. But there is a drop F1 measure for the largest dimension. This could point to insufficient training, but training and validation loss were stable for several final epochs. On the other hand drop for BGL line evaluation related to training. Its training and validation loss changes only first one or two epochs and then stagnate. There were several attempts to fix it by using different learning rate, normalization and regularization methods. But unfortunately all attempts failed. Which is unfortunate since supervised model extreme

strength in line by line evaluation.

But even the best results in the table are not satisfying and far behind state of the art methods from the benchmark. So additional effort was taken to check some assumptions about the processes in prediction model and dig deeper into its behavior.

Unsupervised method is build on idea, that model can learn to predict normal log sequences. Then anomalous logs in sequence will show as deviation from the normal sequence and will have significantly large prediction error. Prediction errors of some anomalous and normal windows, shown in Figure 6.9, support this hypothesis. On the other hand Figure 6.9 also shows that detection via simple threshold is not perfect. There are some FPs caused by high prediction error on normal logs as the one around time 800. There are also FNs when all logs anomalous window have smaller prediction error then threshold. First two windows in Figure 6.9 are examples of such FPs. But it looks like FNs might be limited only to very short sequences. All FN windows in example begin with long sequence of zero prediction error, which correspond to batch padding. It can help with reduction of FN, if most of them are in fact very short windows. But much bigger problem are the FPs which creates additional load on human operators. Precision values in Table 6.5 suggest large FP numbers.

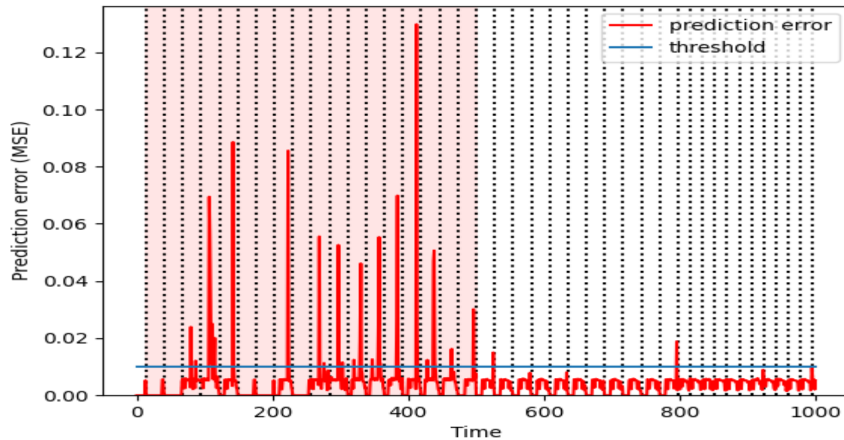


Figure 6.9: Prediction errors on HDFS data. Black vertical lines are window borders. Anomalous windows have red background.

Histogram describing prediction error distribution for normal and anomalous windows is shown in Figure 6.10, to check that random sample of windows from Figure 6.9 correctly represents prediction errors in whole dataset. It is important to note, that whole windows are labeled anomalous, which causes all logs in window to be considered as anomalous, even though most of them probably is not. As result distribution for normal and anomalous logs are expected to be similar, except the end with larger errors where there should be significantly more anomalous logs. Distributions in Figure 6.10 match

these expectations.

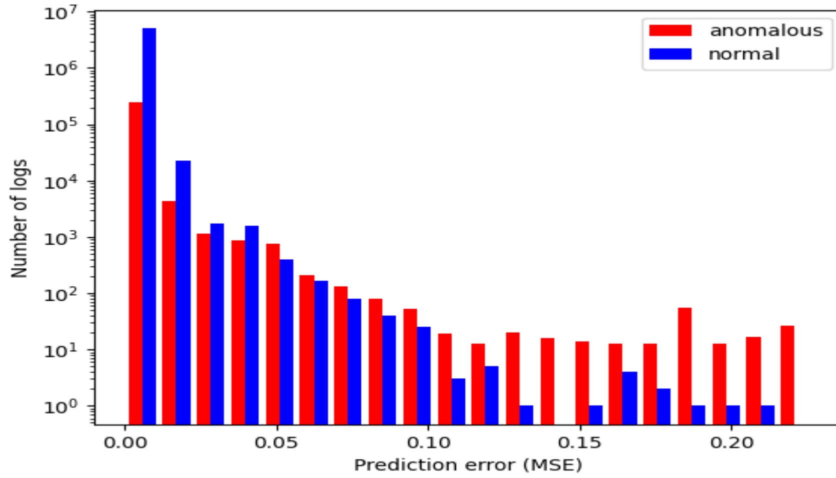


Figure 6.10: Prediction errors distribution (HDFS data)

Next potential weakness of supervised model is setting of correct threshold. This is true especially for simpler version anomaly detection with static threshold based on mean and standard deviation. Figure 6.11 show, how are metrics affected by changing threshold. It is clear, that used threshold, computed on training data, is not ideal and better threshold could double the F1-measure. But even the best threshold results are far behind the other methods from benchmark. This figure also nicely shows the trade off between precision and recall, and the possibility to change ratio between precision and recall by setting different thresholds.

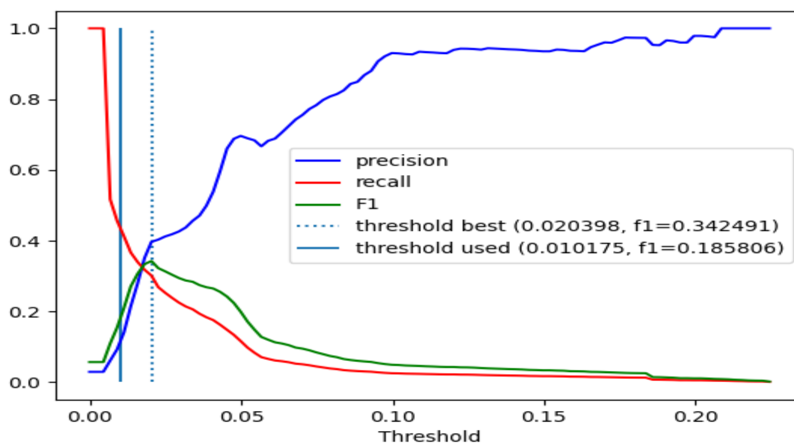


Figure 6.11: Effect of different threshold on metrics for unsupervised model (HDFS data)

The biggest problem is extremely fast fall of recall, meaning large number of FN. There have been some hints, that FN might be reduced by special handling of short sequences in windows. But would require more time and effort to properly investigate and propose some solutions.

To summarize experiments on unsupervised models. It seem that they internally work as expected and most of them learn during the training phase. But proposed prediction model in combination with simple thresholding, just do not provide required level of accuracy. It might be a good idea to try different unsupervised method for sequence processing as encoder-decoder or time convolution based models.

6.4 Experiments summary

Experiments with multiple fastText custom trained models verified that fastText embedding provide meaningful representation of log statement with parameters. Log templates can be detected as clusters in embedding space. Relative position of clusters to each other provide additional semantic information to template when compared with simple log key representation. And variance within cluster allow to pass some information about parameters.

Experiments with supervised models showed that embedding includes features required for distinguishing normal and anomalous logs. Outstanding results on BGL dataset proved the potential of proposed solution. However different model had to be used on HDFS dataset to properly utilize labels, which are provided only on window level for this dataset. This points to some weaknesses of supervised approach in real word use cases. Where large and correctly labeled datasets are rare.

And finally unsupervised models were small disappointment. Detail examination of inner working of models showed expected behavior. Models have learned normal sequence of logs and generated predictions. Prediction errors was much more unstable and included many spikes in anomalous windows. But over all accuracy of anomaly detection based on static threshold was low. Analysis of thresholds effect on accuracy showed that better thresholds for testing data exist. But even the best threshold would not be able to compete with state of the art methods from benchmark.

Conclusion

This thesis studied problem of automatic log analysis and log anomaly detection in particular. Research showed that most existing solutions depend on log parsing to log templates or log keys. Many approaches use just the log keys and throw away information hidden in the text, although there are some that tried various NLP methods to enrich log keys with semantic information from corresponding templates. But only few exceptions considered use of rich information contained in logs as message parameters or header fields.

Log representation, based on *fastText* sentence embedding combined with handpicked custom features, was proposed, to address issue of including semantic information from both log header and message including its parameters. Supervised and unsupervised LSTM based anomaly detection models using this new log representation were proposed, implemented and evaluated in this thesis.

Two publicly available datasets (HDFS, BGL) were used in experiments and benchmarks, with other anomaly detection methods. Firstly assumption that *fastText* embedding is suitable for processing log statements was verified. Then supervised models were trained and evaluated. Labels allow supervised model to learn directly the problem of anomaly detection. So it was used to prove, that information needed to distinguish normal and anomalous logs is included in proposed log representation. Supervised model showed outstanding results on BGL dataset (F1-measure 0.9686). And on the other hand it pointed out disadvantages of supervised method in real word, when different labeling caused pure results on HDFS dataset. But this problem was fixed by modified version of supervised mode, with sequence classification instead sequence-to-sequence, which shown results comparable to the best methods in benchmark.

Finally several unsupervised model were trained and evaluated. Unsupervised models unfortunately cannot compete with high bar set by other state of the art methods, despite very promising results of supervised models. This thesis focused more on log representation and embedding. Though proposed anomaly detection model is relatively simple. There are many ways how to

further improve unsupervised anomaly detection. One is to employ more sophisticated analysis of prediction errors to detect anomalies, then static threshold. There are approaches like dynamic thresholding from [22]. Or completely different unsupervised methods, as encoding-decoding models, can be tried. Since supervised methods proved that proposed log embedding is suitable for anomaly detection.

Appendix A

Bibliography

[1] 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)*, pp. 207–218, Oct 2016.

[2] J. Zhu, S. He, J. Liu, P. He, Q. Xie, Z. Zheng, and M. R. Lyu, "Tools and benchmarks for automated log parsing," *CoRR*, vol. abs/1811.03509, 2018.

[3] W. Xu, *System Problem Detection by Mining Console Logs.* PhD thesis, USA, 2010.

[4] R. Vaarandi and M. Pihelgas, "Logcluster - a data clustering and pattern mining algorithm for event logs," pp. 1–7, 11 2015.

[5] A. A. Makanju, A. N. Zincir-Heywood, and E. E. Milios, "Clustering event logs using iterative partitioning," in *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '09, (New York, NY, USA), p. 1255–1264, Association for Computing Machinery, 2009.

[6] Q. Fu, J.-G. Lou, Y. Wang, and J. Li, "Execution anomaly detection in distributed systems through unstructured log analysis," in *International conference on Data Mining (full paper)*, IEEE, December 2009.

[7] M. Du and F. Li, "Spell: Streaming parsing of system event logs," in *2016 IEEE 16th International Conference on Data Mining (ICDM)*, (Los Alamitos, CA, USA), pp. 859–864, IEEE Computer Society, dec 2016.

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

[9] S. Khatuya, N. Ganguly, J. Basak, M. Bharde, and B. Mitra, "Adele: Anomaly detection from event log empiricism," in *IEEE INFOCOM*

2018 - IEEE Conference on Computer Communications, pp. 2114–2122, 2018.

[10] S. Zhang, Y. Liu, W. Meng, Z. Luo, J. Bu, S. Yang, P. Liang, D. Pei, J. Xu, Y. Zhang, Y. Chen, H. Dong, X. Qu, and L. Song, "Prefix: Switch failure prediction in datacenter networks," in *Abstracts of the 2018 ACM International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '18, (New York, NY, USA), p. 64–66, Association for Computing Machinery, 2018.

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

[12] J. Wang, Y. Tang, S. He, C. Zhao, P. K. Sharma, O. Alfarraj, and A. Tolba, "Logevent2vec: Logevent-to-vector based anomaly detection for large-scale logs in internet of things," *Sensors*, vol. 20, p. 2451, Apr 2020.

[13] A. Tuor, R. Baerwolf, N. Knowles, B. Hutchinson, N. Nichols, and R. Jasper, "Recurrent neural network language models for open vocabulary event-level cyber anomaly detection," *CoRR*, vol. abs/1712.00557, 2017.

[14] T. Mikolov, G. Corrado, K. Chen, and J. Dean, "Efficient estimation of word representations in vector space," pp. 1–12, 01 2013.

[15] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," *CoRR*, vol. abs/1607.04606, 2016.

[16] A. Joulin, E. Grave, P. Bojanowski, and T. Mikolov, "Bag of tricks for efficient text classification," *CoRR*, vol. abs/1607.01759, 2016.

[17] W. Meng, Y. Liu, Y. Zhu, S. Zhang, D. Pei, Y. Liu, Y. Chen, R. Zhang, S. Tao, P. Sun, and R. Zhou, "Loganomaly: Unsupervised detection of sequential and quantitative anomalies in unstructured logs," in *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, pp. 4739–4745, International Joint Conferences on Artificial Intelligence Organization, 7 2019.

[18] X. Zhang, Y. Xu, Q. Lin, B. Qiao, H. Zhang, Y. Dang, C. Xie, X. Yang, Q. Cheng, Z. Li, and et al., "Robust log-based anomaly detection on unstable log data," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, (New York, NY, USA), p. 807–817, Association for Computing Machinery, 2019.

[19] M. Landauer, M. Wurzenberger, F. Skopik, G. Settanni, and P. Filzmoser, "Dynamic log file analysis: An unsupervised cluster evolution approach

for anomaly detection," *Computers & Security*, vol. 79, pp. 94 – 116, 2018.

[20] M. Du, F. Li, G. Zheng, and V. Srikumar, "Deeplog: Anomaly detection and diagnosis from system logs through deep learning," 2017.

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

[22] K. Hundman, V. Constantinou, C. Laporte, I. Colwell, and T. Soderstrom, "Detecting spacecraft anomalies using lstms and nonparametric dynamic thresholding," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '18, (New York, NY, USA), p. 387–395, Association for Computing Machinery, 2018.

[23] C. C. Aggarwal, A. Hinneburg, and D. A. Keim, "On the surprising behavior of distance metrics in high dimensional space," in *Database Theory — ICDT 2001* (J. Van den Bussche and V. Vianu, eds.), (Berlin, Heidelberg), pp. 420–434, Springer Berlin Heidelberg, 2001.

[24] S. Satpathi, S. Deb, R. Srikant, and H. Yan, "Learning latent events from network message logs," *IEEE/ACM Transactions on Networking*, vol. 27, pp. 1728–1741, Aug 2019.

[25] J. L. Ba, J. R. Kiros, and G. E. Hinton, "Layer normalization," 2016.

[26] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, (New York, NY, USA), p. 117–132, Association for Computing Machinery, 2009.

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

Appendix B

Command line interface

```
prediction_main.py arguments:
-h, --help                          show this help message and exit
-v, --verbose                       print logs to console
--data DATA                         path to preprocessed data
--title TITLE                       used when generating result directory
--path PATH                         directory where to save results
--load LOAD                         path to existing results to resume training
--only_evaluate                     do not train, only evaluate on test data
--evaluate_best                     evaluate on best epoch (default is last)
--label_by_block                    force evaluation per window,
                                    even if labels per log are available
--epochs EPOCHS                     number epochs to train
--batch_size BATCH_SIZE             batch size
--limit_train LIMIT_TRAIN           limit number of train windows
--limit_validation LIMIT_VALIDATION limit number of validation windows
--lr LR                             learning rate
--lr_gamma LR_GAMMA                 learning rate gama
--loss {cos,mse,L1}                 loss function used to measure
                                    embedding distance
--fasttext FASTTEXT                 path to fasText model
--lstm_layers LSTM_LAYERS           number of LSTM layers
--linear_width LINEAR_WIDTH         width of hidden dense layers
--linear_layers LINEAR_LAYERS       number of dense layers
--layer_norm LAYER_NORM             add layer normalization
--grad_clip GRAD_CLIP               value to which clip gradient
```

classification_main.py arguments:

Argument	Description
−h, −−help	show this help message and exit
−v, −−verbose	print logs to console
−−data DATA	path to preprocessed data
−−title TITLE	used when generating result directory
−−path PATH	directory where to save results
−−load LOAD	path to existing results to resume training
−−only_evaluate	do not train, only evaluate on test data
−−label_by_block	force evaluation per window, even if labels per log are available
−−epochs EPOCHS	number epochs to train
−−threshold THRESHOLD	threshold for anomaly detection
−−batch_size BATCH_SIZE	batch size
−−lr LR	learning rate
−−lr_gamma LR_GAMMA	learning rate gama
−−fasttext FASTTEXT	path to fasText model
−−lstm_layers LSTM_LAYERS	number of LSTM layers
−−linear_width LINEAR_WIDTH	width of hidden dense layers
−−linear_layers LINEAR_LAYERS	number of dense layers
−−weight WEIGHT	additional training weight for anomaly samples, to fight unbalanced dataset
−−layer_norm LAYER_NORM	add layer normalization
−−grad_clip GRAD_CLIP	value to which clip gradient

Appendix C

Content of enclosed CD

```
log-anomaly-detection
├── BGL_embedding_sample.txt
├── HDFS_embedding_samples.txt
├── loglizer (imported library modul)
├── logparser (imported library modul)
├── rci_batch_scripts
│   ├── benchmark_BGL.batch
│   ├── benchmark_HDFS.batch
│   └── fasttext.batch
├── requirements.txt
└── src
    ├── BGL_Drain_main.py
    ├── BGL_bechmark.py
    ├── HDFS_Drain_main.py
    ├── HDFS_bechmark.py
    ├── classification.py
    ├── classification_main.py
    ├── data_loaders.py
    ├── loglizer
    ├── logparser
    ├── model_environment.py
    ├── prediction.py
    ├── prediction_main.py
    ├── show_losses.py
    ├── utils.py
    ├── visualize_embedding.py
    └── visualize_env.py
```

```
log-anomaly-detection-thesis
├── assignment_cs.pdf
├── assignment_en.pdf
├── assignment_en_signed.pdf
├── ctu_logo_black.pdf
├── ctu_logo_blue.pdf
├── ctuth-core.tex
├── ctuth-names.tex
├── ctuth-pkg.tex
├── ctuth-templates.tex
├── ctuthesis.cls
├── ctuthesis.ist
├── figures
│   ├── anomaly_detection_framework.png
│   ├── architecture_overview.pdf
│   ├── bgl_window_len_hist.png
│   ├── cd_content.pdf
│   ├── classification_learning.png
│   ├── classification_model_arch.pdf
│   ├── classification_prob_hist.png
│   ├── classification_thresholds.png
│   ├── data_loading.pdf
│   ├── embedding_flow.pdf
│   ├── fasttext_embedding_bgl.png
│   ├── fasttext_embedding_cross.png
│   ├── fasttext_embedding_hdfs.png
│   ├── hdfs_window_len_hist.png
│   ├── logStructure.pdf
│   ├── models_in_out.pdf
│   ├── prediction_error_hist.png
│   ├── prediction_errors.png
│   ├── prediction_model_arch.pdf
│   ├── prediction_thresholds.png
│   └── preprocessing_benchmark.pdf
├── log_anomaly_detection.bib
└── log_anomaly_detection.tex
```