CHALMERS
UNIVERSITY OF TECHNOLOGY

Log Classification using NLP Techniques

Data-Driven Fault Categorization of Multimodal Logs using Natural Language Processing Techniques

Master's thesis in Mathematics

ADAM SUHREN GUSTAFSSON
ADAM WIREHED

DEPARTMENT OF MATHEMATICAL SCIENCES

CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2021
www.chalmers.se

Master's thesis 2021

Log Classification using NLP Techniques

Data-Driven Fault Categorization of Multimodal Logs using Natural Language Processing Techniques

ADAM SUHREN GUSTAFSSON
ADAM WIREHED

Department of Mathematical Sciences
Chalmers University of Technology
Gothenburg, Sweden 2021

Log Classification using NLP Techniques
Data-Driven Fault Categorization of Multimodal Logs using Natural Language Processing Techniques
ADAM SUHREN GUSTAFSSON
ADAM WIREHED

Supervisors: Johan Jonasson, Department of Mathematical Sciences
Rozita Akrami, Ericsson AB
Examiner: Marina Axelson-Fisk, Department of Mathematical Sciences

Master's Thesis 2021
Department of Mathematical Sciences
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2021

Log Classification using NLP Techniques
Data-Driven Fault Categorization of Multimodal Logs using Natural Language Processing Techniques
ADAM SUHREN GUSTAFSSON
ADAM WIREHED
Department of Mathematical Sciences
Chalmers University of Technology

Abstract

System logs record system states to facilitate debugging of issues and failures. At Ericsson, several logs are analyzed when faulty baseband hardware is returned from customer networks. Classifying a unit given several logs can be considered a multimodal classification problem where each log represents modes of the system. As systems increase in size and complexity, the resources needed for subject matter experts to analyze these logs increase to a point where it's no longer efficient. Therefore, Ericsson has used machine learning models using manual feature extraction patterns to analyze these logs according to the best understanding of which features should be used for classification. However, this manual feature engineering gives no guarantee of correlation between the best representation of the logs and the output of the classification model. In this thesis, we have shown that a data-driven NLP approach where concatenated bag-of-words representations for each log file fitted on an XGBoost classifier was able to match the production model used by Ericsson. Attempts to incorporate sequential representations of the log entries and parameter lists produced by the Spell and Drain log parsers did not yield improved results. In addition, while deep learning models like Transformers combined with neural Word2Vec embeddings were able to produce similar results, they are prohibitively complex in relation to the simpler solution. Our findings indicate that the baseband unit logs do not show the same high variability in sentence structure, nor seem to depend on structures of sequences for different hardware- or software faults. We also propose that care should be taken when treating logs as texts found in other classical NLP tasks, like sentiment analysis, or document classification where the text is in fact directly generated by humans, as opposed to automatic logging systems. All tested models were evaluated on a holdout test dataset used by the current production model. The existing Ericsson model achieved a macro F1-score of 0.866, the XGBoost model 0.885, and the Transformer model 0.861.

Keywords: NLP, log, classification, machine learning, word embedding, LSTM, transformer, XGBoost, Spell, Drain.

Acknowledgements

We would like to thank Ericsson for giving us the opportunity and resources to work on this thesis. Specifically, we also thank our supervisor Rozita Akrami at Ericsson who was personally responsible for initiating this thesis project with us. We also give our thanks to our supervisor Johan Jonasson at the Department of Mathematical Sciences for offering us guidance and valuable input on the theories and methods of the thesis. We would also like to thank our examiner Marina Axelson-Fisk at Department of Mathematical Sciences for her input on the thesis.

Adam Suhren Gustafsson & Adam Wirehed, Gothenburg, June 2021

Contents

List of Figures

List of Tables

List of Algorithms

Glossary

Baseband Unit
The actual radio hardware unit from which logs are extracted

BOW
Bag-of-Words

Drain
Log parser using a parser tree

HW, SW, NF
The three classes Hardware Fault, Software Fault, and No Fault Found

LSTM
Long-Short-Term Memory

ML
Machine Learning

NLP
Natural Language Processing

RegEx
Regular expression, a sequence of characters specifying a search pattern

RNN
Recurrent Neural Networks

SHAP
'SHapley Additive exPlanations', explaining ML models by connecting credit allocation with local explanations using Shapley values from game theory [2].

Shapley Value
A game theory solution concept predicting how to fairly distribute gains of cooperation among players [3]. In machine learning, this can be used to provide interpretability of ML models and the impact of input features.

SME
Subject Matter Expert

Spell
Log parser using the Longest Common Subsequence algorithm

TFIDF
Term Frequency Inverse Document Frequency

WAEMB
Weighted Average Embedding, line-level weighted average of Word2Vec word embeddings

XGBoost
'eXtreme Gradient Boosting' is a software library providing a gradient boosting framework [4]. Boosting is a sequential ensemble learning algorithm that converts weak models to stronger ones by adjusting the data weights of input samples.

1 Introduction

System logs record significant events and system states to facilitate debugging of performance issues and eventual failures. At Ericsson, such logs are used for analysis when potentially faulty hardware is returned from the customer network for troubleshooting. Historically, Ericsson engineers have found that a percentage of these returned units do not always have a hardware (`HW`) fault, but might instead actually have some software (`SW`) fault, or even no fault found (`NF`). As the return of hardware poses a large cost in both time and resources it is advantageous to be able to automate the troubleshooting process to reduce time and cost for both Ericsson and its customers.

Among these products are the Ericsson Baseband Units, each of which could have several different logs that could be used for troubleshooting. The logs might be interconnected with each other, or be produced by different sub-systems in the unit. This environment of having multiple sources of data available for categorization can be considered a *multimodal* setting, where each log represents different *modes* of the system. While these logs are most often able to be read by the developers and engineers who have created the systems, a large company like Ericsson cannot solely be dependent on these subject matter experts (SME) for troubleshooting the logs. As the systems increase in size and complexity, the time needed for SMEs to analyze these logs increase to a point where it's no longer efficient.

For this reason, Ericsson has taken a data-mining approach to the analysis of these logs, utilizing machine learning (ML) models. These current solutions utilize manually specified feature extraction scripts and data mining to produce statistical metrics and other rule-based features for the ML models. These features are engineered with the help of the SME and chosen according to their best understanding of which features should be used for categorization. However, it is not a guarantee that these features are in fact the best predictors of the system state for a specific ML model. This can lead to data with little to no correlation between the features of the system logs and its behavior and health. For this application, it is therefore favorable to investigate alternatives where humans neither have to monitor the logs themselves nor carry out the feature engineering required for feature extraction in the logs.

1.1 Problem Statement

The thesis aims to research and develop a multimodal log analysis system for classifying baseband units using NLP techniques in a data-driven manner. The classification of baseband units is to label them as Hardware Fault (`HW`), Software Fault (`SW`) or No Fault found (`NF`) based on multiple types of log data.

1.2 Ericsson Dataset

The data utilized in this thesis are hardware and software data logs generated by systems in Ericsson's Baseband Units. Each unit houses multiple hardware and software components that log their current state and occurring events. Different log types stores information about certain areas of the baseband units that could be informative of the type of fault. As the classification is of each unit, all the available log types in that unit should be utilized for the classification.

The dataset used for this thesis is a subset of datasets used previously at Ericsson, for which more information is presented in Section 1.3. An overview of this dataset can be seen in Table 1.1. In the left group of columns, we see information about the original dataset. Of special note, are the two flags specifying if the unit has *any* logs available at all, as well as if there is enough info in the logs to make predictions about the classes `SW` and `NF`.

- `[any_log_available]`: If there are no logs available, a special kind of prediction is made with a class specifying that no log is available. For this thesis, this scenario is not of any interest, so we filter out these samples.
- `[has_enough_info_sw_nff]`: There also exists a rule in the current model based on SME knowledge specifying whether or not a prediction about the labels `SW` and `NF` should be made to even start with. While it is completely possible to utilize the available information in the logs to make this prediction anyway, we opt to also filter out these units from the dataset.

After the removal of the units with these two flags, we are left with the dataset seen in the rightmost column group. We note that the availability of all different logs increases in relation to the original dataset in the leftmost column group. In addition, the ratio of the training/test sets changes from 0.8/0.2 to 0.84/0.16. We also note that the distribution of classes changes where the ratio of the `HW` class decreases, and the ratio of the `NF` class increases.

One of the log types found in the dataset is the `esilog`. This is a very large log file, which in fact is a collection of several smaller log files. For this reason, we extract the most useful of these smaller logs from this file, according to SME knowledge. Specifically, we extract the log files `bpmlog` and `hwlog`. After this extraction, we are left with the final dataset used in the project as seen in Table 1.2. We utilize the same train and test split of the data as found in the original production model. We

Table 1.1: Information about the original datasets provided by Ericsson for the previous production model.

	Original dataset			Selected dataset		
	All	Train	Test	All	Train	Test
Number of units	7740	6191	1549	3972	3325	647
Ratio of units	1.00	0.80	0.20	1.00	0.84	0.16
Flag `[any_log_available]`	0.90	0.88	0.96	1.00	1.00	1.00
Flag `[has_enough_info_sw_nff]`	0.51	0.54	0.42	1.00	1.00	1.00
Availability of `ailog`	0.68	0.66	0.77	0.96	0.96	0.99
Availability of `llog`	0.66	0.64	0.72	0.97	0.97	0.97
Availability of `uboot`	0.89	0.87	0.96	0.98	0.98	1.00
Availability of `esilog`	0.65	0.63	0.75	1.00	1.00	1.00
Ratio of `HW`	0.68	0.68	0.68	0.57	0.58	0.53
Ratio of `SW`	0.17	0.17	0.17	0.17	0.17	0.17
Ratio of `NF`	0.16	0.16	0.16	0.25	0.25	0.30

don't go into detail about exactly what the logs `ailog`, `bpmlog`, `hwlog`, `llog`, and `uboot` contain, or how they are generated by the system. Instead, we treat these all in the same way, with no further insights into what they might contain, or in what way they represent the unit state.

Table 1.2: Final datasets for use in the project, extracted from the original data provided by Ericsson.

	Samples	Ratio	`ailog`	`bpmlog`	`hwlog`	`llog`	`uboot`
Train	3325	0.837	0.955	0.999	1.0	0.972	0.982
Test	647	0.163	0.989	1.000	1.0	0.972	1.000
All	3972	1.000	0.961	0.999	1.0	0.972	0.985

Without getting into too much detail about the logs, they all contain information about different areas of the baseband units. Some of the log types contain information that was written during its time in the customer network, and some contain information from diagnostics that have been performed on the units at a screening center. One purpose of the thesis is to examine data-driven solutions where having deep knowledge of each log type and the differences between is not be required. You could argue that the less domain knowledge you have of the logs, the lower are the chances of using that domain knowledge to improve the solution. This makes the results and findings from this thesis of data-driven solutions more representative of real-world use cases. Excerpts from the contents of all log types are presented in Figure 1.1. Note that not only does the information in the logs vary between the log types, but also how that information is structured row-wise. Figures throughout the thesis may obfuscate information from the log files. Note that it does not affect the results or conclusions.

```
// Excerpt from "ailog"
.........1 YYYY-MM-DDTHH:MM:SSZ  "INFO: Setting system time to software build time: "YYYY-MM-DD HH:MM:SS""
.........2 YYYY-MM-DDTHH:MM:SSZ  "INFO: Networkloader type2 booted from partition /dev/sda1"
.........3 YYYY-MM-DDTHH:MM:SSZ  "INFO: Running version: "XXXXXXXXX-YYYYYY""
.........4 YYYY-MM-DDTHH:MM:SSZ  "INFO: Autointegration waiting for user input"
.........5 YYYY-MM-DDTHH:MM:SSZ  "INFO: Zero-touch application loaded - waiting for activation"
```

```
// Excerpt from "bpmlog"
=== NVM ===
*H* uptime=16/16 starts=0/1
[0] System Event: System Startup(e1) ID:0x0 P1:0x0 P2:0x0
[16] System Event: PLD Shutdown(e0) ID:0x0 P1:0x1 P2:0x0
*H* uptime=53/37 starts=1/1
```

```
// Excerpt from "hwlog"
1   700101000011  (004)  Power On (Cable plugged in). Timestamp is however from the power off moment    25   Z/XXXXXXXXXXXX/XX_YYY
2   700101000011  (004)  Power On (Cable plugged in). Timestamp is however from the power off moment    25   Z/XXXXXXXXXXXX/XX_YYY
3 * 180827024305  (003)  SW version : BASEBAND XXXXXXXXXXXX/Y ZZZZ started                              1    XXXXXXXXXXXX_X_YYYYYY
4 * 180827024504  (000)  Site : XXXXXXXX                                                                1    XXXXXXXXXXXX_X_YYYYYY
5   180827025046  (004)  Power On (Cable plugged in). Timestamp is however from the power off moment    25   Z/XXXXXXXXXXXX/XX_YYY
```

```
// Excerpt from "llog"
YYYY-MM-DDTHH:MM:SS du1 xxxx[XXXX]: rlog: rlog: $ Power on $ YYYY-MM-DD HH:MM:SS $ - $ - $ Cold $ - $ - $ -
YYYY-MM-DDTHH:MM:SS du1 xxxx[XXXX]: rlog: rlog: $ Ordered restart $ YYYY-MM-DD HH:MM:SS $ - $ - $ Cold $ - $ - $ 'Data restore'
YYYY-MM-DDTHH:MM:SS du1 xxxx[XXXX]: rlog: rlog: $ Power on $ YYYY-MM-DD HH:MM:SS $ - $ - $ Cold $ - $ - $ -
YYYY-MM-DDTHH:MM:SS du1 xxxx[XXXX]: rlog: rlog: $ Ordered restart $ YYYY-MM-DD HH:MM:SS $ - $ - $ Warm $ - $ - $ 'Manual restart'
YYYY-MM-DDTHH:MM:SS du1 xxxx[XXXX]: rlog: rlog: $ Power on $ YYYY-MM-DD HH:MM:SS $ - $ - $ Cold $ - $ - $ -
```

```
// Excerpt from "uboot"
[    0.000000] Booting Linux on physical CPU 0x0
[    0.000000] Initializing cgroup subsys cpuset
[    0.000000] Initializing cgroup subsys cpuacct
[    0.000000] Linux version XXX (xxxxxx@yyyyyyyy) (gcc version X.X.X (Linux X.X.X-Y.Y) ) ...
[    0.000000] CPU: XXX Processor [XXXXXXXX] revision X
```

Figure 1.1: Excerpts of lines from the different log types. These are slightly modified by removing white space and extra characters to better visualize their contents. In addition, their contents have been obfuscated in several places.

1.3 Ericsson Model

A classification model that has been used at Ericsson is an ensemble model in the form of a Random Forest. This model has been trained on the dataset as described in Section 1.2. For each sample (unit) in the dataset, features have been extracted from the raw text contents of the log files through manually created rules and patterns implemented using regular expressions and text processing scripts. These features can be the number of occurrences of specific messages in the log files or higher-level features such as the number of lines. Throughout this report, we will refer to this model and its feature extraction pipeline as the *Ericsson model*.

1.4 Scope and Delimitations

The Ericsson model previously mentioned, utilizes manually specified feature extraction scripts and data mining approaches to produce statistical metrics and other rule-based features for ML models. One of the purposes of the thesis is to research and evaluate alternative automated feature-extraction methods to the currently used manual feature extraction. Therefore, we will not consider manual feature extraction when developing and researching our own possible feature extraction techniques and models. However, these solutions will be compared against the Ericsson model which utilizes this manual data-mining approach.

2 Background

2.1 Log Analysis

Computer-generated records, or logs, are often used by software developers to debug or troubleshoot their system or application. Given a simple and small-scale system, it might be easy to read the contents of the log file and then understand if it is behaving as expected. As more and more industries start to rely on software systems, the corresponding logs get increasingly complex and larger. Without domain knowledge, these types of logs could be difficult or almost impossible to read or understand, both in terms of their complexity and the overall amount of logs.

Instead of needing multiple people with extensive domain knowledge to monitor large amounts of log data, automation would streamline the process and also reduce the need for monotonous work. Developing algorithms to automatically monitor and detect anomalous behavior, or classifying faults, in systems on a large scale is therefore highly sought after. Domain experts can assist during development by understanding what the algorithms should look for in the logs when evaluating the state of the system. With useful algorithms in place, SMEs can monitor the algorithms and their decisions instead of huge amounts of raw textual data. It is also possible to make the algorithms data-driven and let themselves decide what parts of the logs are informative and not. Regardless of the algorithms are data-driven or not, the textual data that is the logs need to be transformed into a numerical representation to make them interpretable for a computer.

2.2 Natural Language Processing

Natural Language Processing (NLP) techniques have been utilized for many different methods and solutions when working with textual data. For example, Topic Modeling and text categorization using Latent Dirichlet Allocation and other statistical methods. With the advent of deep learning, the NLP field was presented with new methods for solving both new and old problems. For example, by utilizing continuous vectorization of words and characters, with contextualized information stored as spatial relations between numerical vectors [5].

NLP techniques have been utilized in the log analysis field with promising results in the past [6], [7] The most common use case have been to utilize sequential neural networks such as Recurrent Neural Networks (RNN) or Long-Short-Term-Memory (LSTM) models for detecting anomalies in log data. The sequential properties of the models are able to identify patterns during normal behavior reflected in the log data and are also able to react when this pattern is broken. This makes it possible to perform binary classification of whether the logs are displaying normal or anomalous behavior, as well as detecting specifically when these potential anomalies occur. One related area that has been studied less is the multi-label classification of logs. Instead of classifying if a log is anomalous or not, the model should be able to classify the type of error or fault in the system.

With the introduction of Transformers and the attention mechanism in 2017, [8], the NLP field was again renewed with a new type of model architecture and theory of how to solve natural language related tasks. In [8] the Transformer model outperformed, with lower time complexity, many popular sequential models on NLP-related tasks such as machine translation. The publication had a large effect on the research within NLP and led to many new publications and high-performing language models which showcased new state-of-the-art performance [9], [10]. However, these new attention-based models have not yet made a large impact on the log analysis field.

2.3 Related work

When utilizing NLP techniques within log analysis it has mostly been for the purpose of *anomaly detection*. This is a separate problem to that of log classification, in that we want to detect *anomalies* locally in log files just as they occur, as opposed to classifying an entire log file. This might be the detection of an unexpected log entry, or an abnormal feature value, like e.g. detecting that the CPU temperature is too high. However, such anomalies might not necessarily be related to the classification of an entire log. Publications such as DeepLog [6] and LogAnomaly [7] have shown promising results and performance for detecting anomalous behavior in large systems. Both of these implementations utilize log parsers that are able to cluster log entries based on similarity measures. In DeepLog each log entry is parsed using parsers like Spell or Drain parser [11], [12] where the log key (message type) of the entry is identified, as well as the corresponding parameter values (see Sections 3.1.1 and 3.1.2). They take a holistic approach to the problem in which they not only take the actual log key of each entry into account but also the corresponding variable parameter values.

The objective of LogAnomaly is to, in an unsupervised way, automatically and accurately detect both sequential and quantitative log anomalies in real-time. Similar to Spell and Drain, LogAnomaly utilizes log-templates in its system. The parsing of the raw logs is handled by the log-parsing system FT-Tree [13] and converted into templates. The templates are then embedded with the proposed system Template2Vec. The system is inspired by Google's Word2Vec and embeds each word in the template using the distributional lexical-contrast embedding model dLCE [14]. For a given template, Template2Vec computes its template vector, which is the weighted average of the word vectors of the words in the template, to represent the distribution of the template.

Both of these papers are about anomaly detection, and not fault classification which is the focus of this thesis. One related topic that could prove useful is document classification which is another widely researched area in NLP. In the paper, [15] the authors investigate how deep versions of Neural bag-of-words models (NBOW) can perform on tasks that have been dominated by syntactically aware models, such as text classification. Adding depth to an NBOW creates what the authors call a Deep Averaging Network (DAN). The depth is added to counter one of the issues with averaging word embeddings, the similarity between averaged values. The authors examine if the depth could make these small distinctions more apparent and capture subtle variations in the input better than the standard NBOW model. The fast and simple computations are still kept as each layer only requires a single matrix multiplication, so the complexity scales linearly with the number of layers rather than the number of nodes in a parse tree. Their proposed model also integrates a *dropout* method where entire word embeddings are dropped from the vector average. Results show performance slightly worse or similar to more advanced models on sentence and document-level sentiment analysis and factoid question answering tasks. However, their model has a significantly lower training time, while also outperforming simple RecNN- and bigram naive Bayes models.

One major aspect of the thesis is the multimodal data. There are multiple log types within a single unit and either the prediction models or the feature extraction needs to be able to process it. There is existing work related to this which has been carried out using Ericsson logs previously. One of these projects is in the form of a bachelor's thesis from 2020 [16]. In this work, the two methods of early and late multimodal fusion are investigated for the purpose of anomaly detection. For the early fusion, two methods were examined:

- In **Early Fusion Match Preprocessing** the log entries are matched by finding the closest timestamps and concatenating the event codes of the merged entries into a tuple
- In **Early Fusion Merge Preprocessing** the individual events are selected in chronological order and appended to a new log

For the **late fusion**, there is only one method used in which the output of one LSTM model per log type is fed through a leaky integrator and a threshold function is applied for anomaly detection. This can be seen as a sort of temporal majority vote of the LSTM model outputs. This publication was a key inspiration for our thought process in how to process the multimodal data in this thesis.

3
Theory

3.1 Log Parsing

One of the first steps of log analysis is the conversion from *unstructured* raw log data into a *structured* representation. The unstructured representation is in general the raw text contents of some log file, with some separation of log messages, usually in the form of a newline character. What exactly defines the structured representation is however not the same for all log parsing. One structured representation could be to split each message into separate parts and store each such part in a columnar format, like a table. This way, the first column could for example contain the timestamp of the message, the second the log level, etc. Another useful feature of a structured representation is some sort of identifier specifying the *message type* of each message.

The simplest approach, and that which has conventionally been used, is to manually identify various structures of the messages in a log, and try to capture these using a search pattern system, usually through regular expressions. This has the advantage of being quite easy to implement, and the features that are extracted can be chosen using domain knowledge such that they should provide useful information for a machine learning model. The very obvious disadvantage is that it is very time-consuming, and while domain knowledge "might" lead to good features, this process is separate from the machine learning model meaning that it is not a guarantee or something which the optimization and training itself have control over.

A more auspicious approach is to automatically generate, or infer features from the logs in an automated manner. There exist several such systems, many of which have been compiled in the paper *Tools and Benchmarks for Automated Log Parsing* [17]. Among these, popular approaches are for example Spell [11] which utilizes the LCS algorithm, or Drain [12] which instead uses parser tree systems. These are systems that aim to identify which unique message types are present in the log, relate each line to one of these message types with a unique event id, and also provide a message template that can be used to extract the variable features (parameters) from each log line.

To summarize, the following methods are commonly used to parse log from an unstructured representation to a structured one:

- **Manual approaches**: E,g. through regular expression pattern matching
- **Automated log parsers**: Such as Spell and Drain

3.1.1 Spell

Spell (Structured Parser for Event Logs using Longest Common Subsequence), is a log parser capable of parsing "unstructured log messages into structured message types and parameters in an online streaming fashion" [11]. A key beneficial property of this log parser is the ability to parse logs in an online fashion, such that new message types are created by the system "on-the-fly" as new messages are processed by the log parser. This is opposed to the parser requiring offline batch-processing of logs before it can be used. In addition, the system implements various pre-filtering steps to reduce the time complexity of the parser such that it is close to linear with the number of log entries.

As the name implies, the basic building block of the parser is the Longest Common Subsequence (LCS) algorithm. As per the notation used in their work, let Σ denote an alphabet of characters. Given a sequence

$$\alpha = \{a_1, a_2, ..., a_m\},\ a_i \in \Sigma\ \forall i \in [1, m], \tag{3.1}$$

a subsequence of α is defined as

$$\{a_{x_1}, a_{x_2}, ..., a_{x_k}\},\ \forall x_i : x_i \in \mathbb{Z}^+,\ 1 \leq x_1 < x_2 < ... < x_k < m. \tag{3.2}$$

Let $\beta = \{b_1, b_2, ..., b_n\}, b_j \in \Sigma\ \forall j \in [1, n]$ be another sequence. A subsequence γ is then called a *Common Subsequence* of α and β iff it is a subsequence of each. The *Longest Common Subsequence* problem for the problem instance α and β is then the to find the longest such γ. An example given in the paper is to find the LCS of the sequences $\{1, 3, 5, 7, 9\}$ and $\{1, 5, 7, 10\}$, which is then $\{1, 5, 7\}$.

During parsing, each log entry (line) is tokenized using some system-defined set of delimiters consisting of suitable characters like spaces, colons, and equal signs. Each such log entry is assigned a unique incrementing id when they arrive. The algorithm utilizes a data structure called an `LCSOjbect` which contains ids and metadata of log entries which are candidates of a specific message type, see Figure 3.1. These are in turn stored in the `LCSMap`. When a new entry arrives, it is compared against each `LCSObject` in the `LCSMap` to decide whether to insert the entry into the existing `LCSOjbect` or to create a completely new `LCSObject` in the event that there's no match.

When a new log entry e arrives it is then first tokenized into a sequence of tokens s. Assume that the ith `LCSObject` has the existing LCS consisting of tokens q_i. The value l_i is then the length of LCS(q_i, s). If l_i is the longest such length among all objects in the `LCSMap`, it is also compared against the threshold $\tau = |s|/2$ to see if it is greater. If this is the case, it is considered to be of the same message type as the other entries in the ith `LCSObject`. The algorithm then uses backtracking

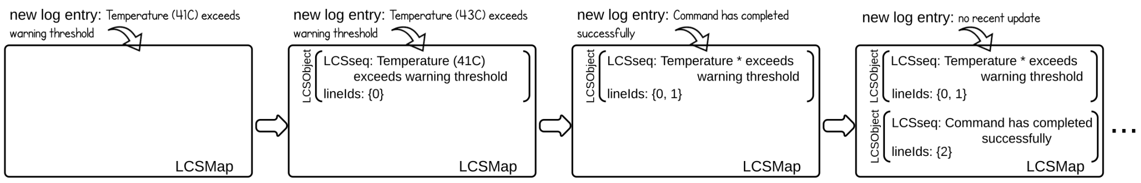

new log entry: Temperature (41C) exceeds warning threshold
LCSMap
new log entry: Temperature (43C) exceeds warning threshold
LCSObject
LCSseq: Temperature (41C) exceeds warning threshold
lineIds: {0}
LCSMap
new log entry: Command has completed successfully
LCSObject
LCSseq: Temperature * exceeds warning threshold
lineIds: {0, 1}
LCSMap
new log entry: no recent update
LCSObject
LCSseq: Temperature * exceeds warning threshold
lineIds: {0, 1}
LCSObject
LCSseq: Command has completed successfully
lineIds: {2}
LCSMap
...

Figure 3.1: Basic workflow of Spell [11]

to generate a new LCS sequence given q_i and s. Specifically, the `'<*>'` character is placed where the two sequences disagree which also marks placeholders for the *parameters*. This describes the basic procedure of Spell.

In addition, several efficiency improvements have been implemented in the form of pre-processing steps. These are performed when attempting to find the matching `LCSObject`, before attempting to carry out the full LCS algorithm. These can be seen in Figure 3.2.

Simple Loop: Maintain one pointer p_s at the first token of s and one pointer p_q at the first token of q_i. If the tokens match, advance both pointers, otherwise advance only p_q. When p_q is at the end of the string, check if p_s is also at the end of its string. A further pruning can also be performed here to skip q_i completely if $|q_i| < |s|/2$. In addition, a set similarity score using the Jaccard index of the strings can be utilized as a proxy to compare only sequences with at least half common tokens.

Prefix Trees: The sequences q_i can also be indexed using a Prefix Tree T to prune away candidates. For each consecutive token in s, only select the available branches matching the token, or mark the character as a parameter if there are no matching branches. This approach guarantees to return a $q_i = \text{LCS}(q_i, s)$ if it exists, but it does not guarantee that q_i is the longest such sequence among all LCS objects. In practice, this tends to work however since parameters in a log entry tend to appear near the end.

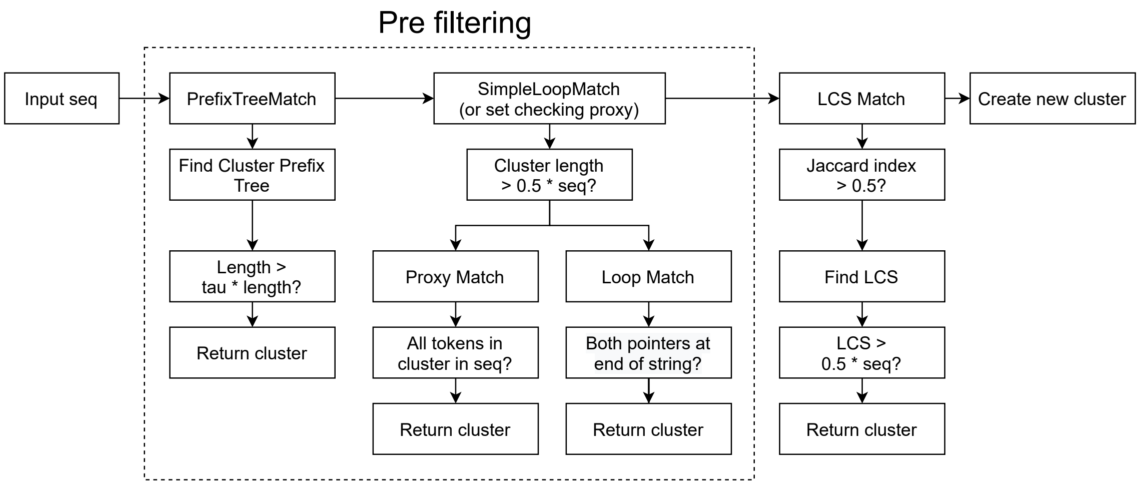


Figure 3.2: Illustration of the workflow of Spell when a new log entry arrives.

3.1.2 Drain

Another online log parser is described in the paper *Drain: An Online Log Parsing Approach with Fixed Depth Tree* [12]. Here, the authors aim to improve on earlier works like Spell [11] by achieving higher accuracies of the resulting event types and faster running times. Their system utilizes a *Parser tree*, as shown in Figure 3.3. When the log entry arrives, it is first matched to one of the children of the root node which matches the **number of tokens** of the message. At the subsequent children of that node, it is then matched on the first token in the message, then the second, and so on, until it reaches a maximum *fixed tree depth* at one of the leaves. The method handles numerical values in a token of the log entry by simply replacing the entire token with the `'*'` token as a catch-all entry for that token, as can be seen as an example in the rightmost branch in Figure 3.3. If no matching path is found in the tree, it is updated by adding more branches in the node in which the search terminated.

In each of the leaves several groups of log messages are stored, where each group has a *log event* and a list of log IDs. The log event is the template that is the common descriptor for the messages in the group consisting of the constant parts of the message, as well as parameters denoted by `'*'`. The log ID is simply a unique number associated with each line in the log. The best group is matched by a similarity measure of the entry and the log group sequence by the average number of matching aligned tokens in the sequences. The similarity between the new entry and the events in the groups is calculated as

$$simSeq = \frac{\sum_{i=1}^{n} \mathrm{equ}(seq_1(i), seq_2(i))}{n}, \tag{3.3}$$

where seq_1 and seq_2 represent the new log entry and the group event, respectively. $seq(i)$ denotes the token at location i in the sequence, and n is the length of the sequences. The function equ is defined as

$$\mathrm{equ}(t_1, t_2) = \begin{cases} 1 & \text{if} \quad t_1 = t_2, \\ 0 & \text{otherwise} \end{cases} \tag{3.4}$$

for two tokens t_1 and t_2. If the maximum *simSeq* for some group is larger than a similarity threshold *st*, we return this group, otherwise return nothing indicating that no group was found.

If a group was found, the log entry is added to the group. Further, the event is also updated by replacing aligned tokens that are not matching with a wildcard character `'*'`. If no group was found, a new group is created with only the log ID of the incoming entry, and where the event template is exactly the message of the entry.

One of the properties of the tree is in fact that it guarantees that the depth of all leaf nodes are the same and fixed. Further, there is also a *maxChild* parameter that limits the maximum number of children in a node. In addition, this method also assumes that all messages of the same event type are in fact of the same length, as this is the first property of the message which is used to traverse the tree.

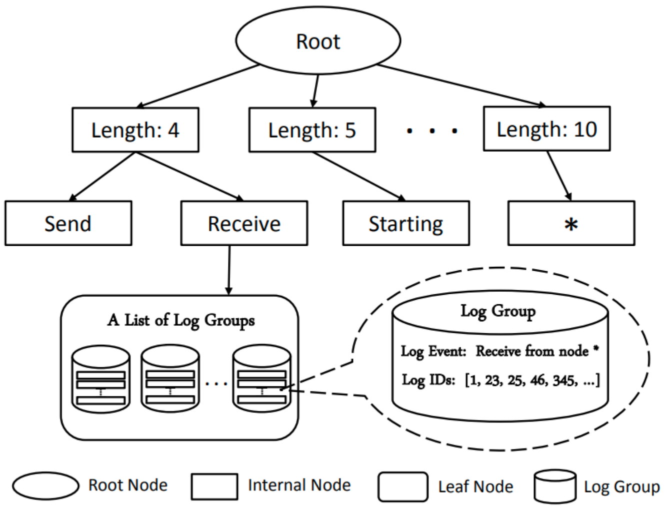


Figure 3.3: Illustration of a parser tree with depth 3 of Drain. [12].

3.2 NLP Techniques

When working with textual data in a machine learning context one needs to transform the data into numerical values. This is also referred to as vectorization or feature extraction. By utilizing NLP techniques instead of a conventional data-mining approach it is possible to transform the entirety of the textual data instead of pre-determined subsets. If extensive domain knowledge is available, data-mining methods might be more effective since you are aware of what parts of the data should be informative. The upside of data-driven NLP techniques is a more generalized approach to the problem where extensive domain knowledge is not necessary and the ML models themselves can determine what information in the textual data is useful or not. This could result in models finding subsets or properties of the log data that were not thought of as informative before, further extending the knowledge of the domain. Depending on the representation technique chosen, certain information or properties of the data might not be included in the numerical representation. This can vary from sequential properties to semantic relations between words and sentences. Where for example more simple feature representations such as Bag-of-Words do not contain any of these properties after the vectorization of the textual data.

3.2.1 Bag-of-Words

Bag-of-Words (BOW) is a method for extracting text and transform it into a numerical representation by describing the occurrences of words within a document. A document can therefore be represented by a numerical vector (vocabulary) of size $\mathcal{N}$ where each element represents the occurrences of a specific word from the vocabulary in the document. It is called a "bag" of words since there is no information about the sequence of the words or the sentences in the document. In Table 3.1 three different documents are shown with their corresponding BOW representation. Here the BOW model's vocabulary is of size $|\mathcal{N}| = 6$ and contains the words shown in the table.

Table 3.1: Example of Bag-of-Words method on three documents with vocabulary size of six

Document	the	cat	sat	in	hat	with
the cat sat	1	1	1	0	0	0
the cat sat in the hat	2	1	1	1	1	0
the cat with the hat	2	1	0	0	1	1

3.2.2 TFIDF

Term Frequency Inverse Document Frequency (TFIDF) is a variant of bag-of-words that intends to reflect how important a word is to a document or corpus. Instead of only representing the occurrences of each word a weighting factor to the words in the vocabulary is incorporated. TFIDF is the product of the two statistics, *term frequency* in Equation (3.5) and *inverse document frequency* in Equation (3.6). The term frequency, $\mathrm{tf}(t, d)$ is the frequency of term t,

$$\mathrm{tf}(t, d) = \frac{f_{t,d}}{\sum_{t' \in d} f_{t',d}}, \tag{3.5}$$

where $f_{t,d}$ is the raw count, i.e. the number of occurrences of term t in document d. Inverse document frequency measures how much information words provide based on how common or how rare they are across all documents in the corpus. It is the logarithmically scaled inverse fraction of the documents that contain the word,

$$\mathrm{idf}(t, D) = \log \frac{|D|}{|\{d \in D : t \in d\}|}, \tag{3.6}$$

where $|D|$ is the number of documents in the corpus. A TFIDF value then increases proportionally to the number of times the word appears in the document and is inversely scaled by the number of documents in the corpus that contains the word.

3.2.3 N-Grams

While methods like standard BOW and TFIDF take no consideration to the actual ordering (sequence) of the tokens in the text, bag-of-n-grams attempts to model this by calculating the frequency of tuples of n adjacent words in the corpora. It is also possible to construct n-grams of characters in words, but for this thesis, only n-grams of words are used. Commonly used frameworks such as Scikit-learn [18] include n-gram functionality in both their BOW and TFIDF implementations. This makes it easy to include some sequential information of the data with these simple NLP methods.

3.2.4 Word2Vec

To extract more information from text documents it could be desirable to produce feature representations that capture similarities and dissimilarities between words and sentences. Using BOW or TFIDF and representing words as one-hot encoded vectors will not capture these properties. The words *cat* and *dog* will be equally similar (or dissimilar) to each other as *cat* and *space* are to each other. They will simply be represented by an index in the feature vector. To produce feature representations of documents, sentences, and words with deeper properties one could use embedding techniques. Instead of representing words or sentences as one-hot encoded vectors, they are represented as a continuous vector instead. This could be done with an embedding layer, which is mathematically equivalent to a one-hot encoding followed by a linear layer that contains the feature vectors for the words in the vocabulary, see Figure 3.4.

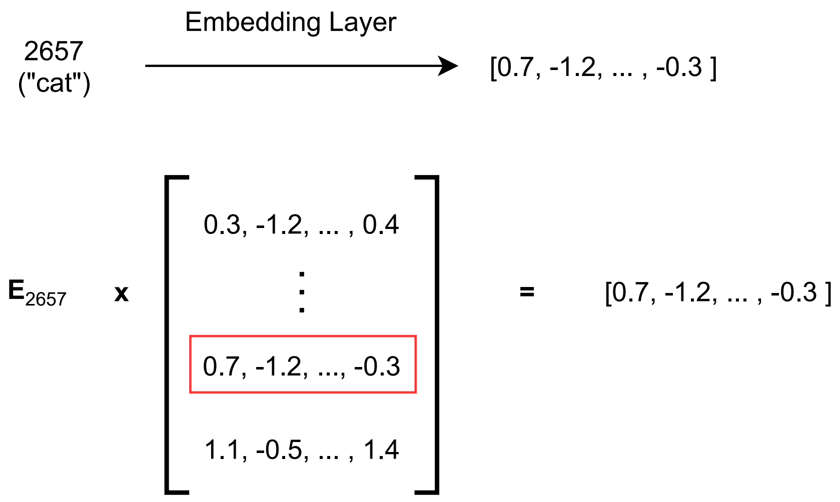


Figure 3.4: Example of embedding layer for the word "cat" with vocabulary index 2657

An embedding can be considered a *structure-preserving function*, or a feature representation method where embedding layers can learn and preserve the relationship between words. This relationship is often measured by the cosine distance between word vectors. So for the example given before, the cosine distance between *cat* and *dog* would be less than between *cat* and *space*. Given that the embedding layer is pre-trained on a large corpus where this is true, and not only randomly initialized before use.

One of the more popular word embedding models that can capture contextualized information is Word2Vec [5]. Two different architectures of the model were presented in the paper for computing continuous vector representations of words from large data sets. The two models architectures are Continuous Bag-of-Words (CBOW) and Continuous Skip-gram, see Figure 3.5.

The models learn word embeddings differently from each other. CBOW tries to predict the context (current) word based on the surrounding words. In training, this

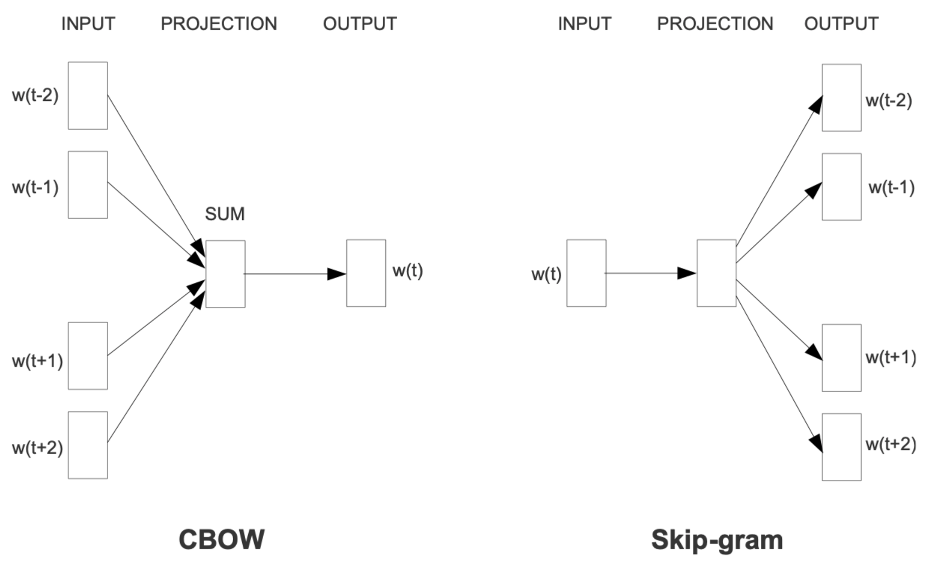


Figure 3.5: CBOW training method (left), Skip-gram training method (right) [5]

results in making the surrounding words closer to the context word in the vector space. The Continuous Skip-gram does the opposite. Instead of predicting the current word based on the context, it tries to maximize the classification of a word based on another word in the same sentence. During training, this results in making the context words closer to the surrounding words in the vector space.

The two models try to achieve the same goal of generating a vector space of word embeddings with spatial features. Meaning that similar words or words that appear together in the same context should be closer to each other in the vector space than words that do not appear together in the same context. In Figure 3.6 we see five different word embedding vectors. Due to the spatial relations between the vectors, by subtracting *man* and adding *woman* to the *king* embedding you get something that closely resembles the *queen* embedding. This spatial relation could be useful when the embeddings are used with a prediction model.

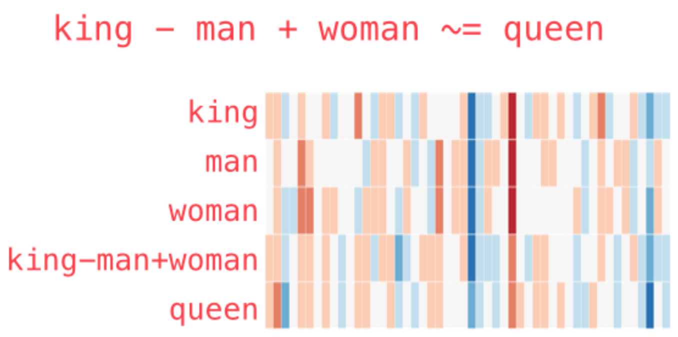


Figure 3.6: Spatial relation between word vectors [19]

3.3 Classification Models

To classify the vectorized textual data, machine learning models are implemented. For this thesis, there are multiple types of textual data (log types) for each sample (baseband unit). This requires that either the model is capable of ingesting multi-modal data and output one classification or to have one model for each log type and then merge the predictions based on a decision rule. Based on the format and properties of the log data used in the thesis, some model types are more suitable for one method than the other.

3.3.1 Random Forest

Bagging, or bootstrap aggregation, is a technique for reducing the variance of an estimated prediction function [1]. The idea of bagging is to average many noisy but approximately unbiased models to reduce the variance. Random Forest (RF) is an ensemble learning method for classification and regression that utilizes a modified bagging method. It builds a large collection of de-correlated isolation trees and then averages them. As trees can capture complex interaction structures in the data, with relatively low bias, they are good candidates for bagging. The algorithm for Random Forest classification is presented in Algorithm 3.1.

Algorithm 3.1 Random Forest for Classification [1]

```
1. For b = 1 to B:
    (a) Draw a bootstrap sample $Z^*$ of size N from the training data.
    (b) Grow a random-forest tree $T_b$ to the bootstrapped data, by re- cursively
        repeating the following steps for each terminal node of the tree, until the
        minimum node size $n_{min}$ is reached.
          i. Select $m$ variables at random from the $p$ variables.
         ii. Pick the best variable/split-point among the $m$.
        iii. Split the node into two daughter nodes.
2. Output the ensemble of trees $\{T_b\}_1^B$
```

To make a prediction at a new point x:

Classification: Let $\hat{C}_b(x)$ be the class prediction of the b:th random-forest tree. Then $\hat{C}_{rf}^B(x)$ = majority vote $\{\hat{C}_b(x)\}_1^B$

3.3.2 XGBoost

Ensemble learning is a technique in which multiple weak machine learning models are combined to improve performance. For example, the Random Forest model (presented in Section 3.3.1) is an example of a *bagging* (parallel) ensemble model. Multiple decision tree models are grown in parallel, trained on random subsets of the data, and individual model decisions are combined through a majority vote. Another method is a *boosting* (sequential) ensemble where the separate models are instead trained sequentially. Initially, the first model of the sequence is trained on a random subset of samples from the data. It then assigns the samples for which it made the correct prediction a *low* weight, and the samples with the wrong prediction a *high* weight. These re-weighted samples are then put back into the data before it is fed to the next model in the sequence where the process is repeated. Through this technique, the next predictor in the sequence can learn from the mistakes of the previous. *Gradient boosting* is a further improvement on the boosting technique, in which the weights assigned by the weak learners are updated using gradient descent using the error of the predicted output and the ground truth.

XGBoost, Extreme Gradient Boosting, is an optimized distributed gradient boosting library [20]. It introduces an extreme gradient boosting algorithm and is specially designed to improve speed and performance compared to other ensemble model implementations. What separates XGBoost from standard gradient boosting models is by optimizing this gradient boosting algorithm both on software and hardware level, such as cache awareness, out-of-core computing and parallelized tree building. It is, therefore, given the name *extreme* gradient boosting algorithm. This results in a highly effective and capable decision-tree-based ensemble machine learning algorithm. To be clear, XGBoost is in itself a gradient boosting *algorithm*, while the underlying model can be considered to be a sequential tree ensemble model. For this work, however, we refer to this entire algorithm and the underlying model as simply the "XGBoost model".

3.3.3 LSTM

A Recurrent Neural Network (RNN) is a type of artificial neural network that uses sequential data, e.g. time-series data. Unlike traditional feed-forward networks, RNNs can utilize information from previous inputs in the sequence to influence future outputs. They do this by having recurrent connections in the network to allow information to persist, see Figure 3.7.

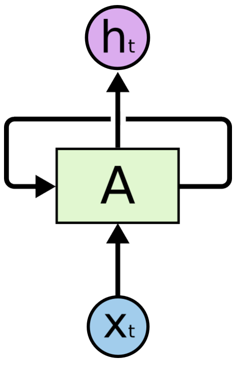


Figure 3.7: Recurrent Neural Network structure example [21]

In Figure 3.7 the neural network A takes the input x_t and produces the output h_t. The recurrent connection allows the information that was present during sequence t to also be present during sequence $t+1, t+2$ and so on. You can think of an RNN as multiple copies of the same network in a sequence, see Figure 3.8. By this chain-like structure, RNNs have a natural architecture to work with sequential data as information from previous sequences can influence later ones.

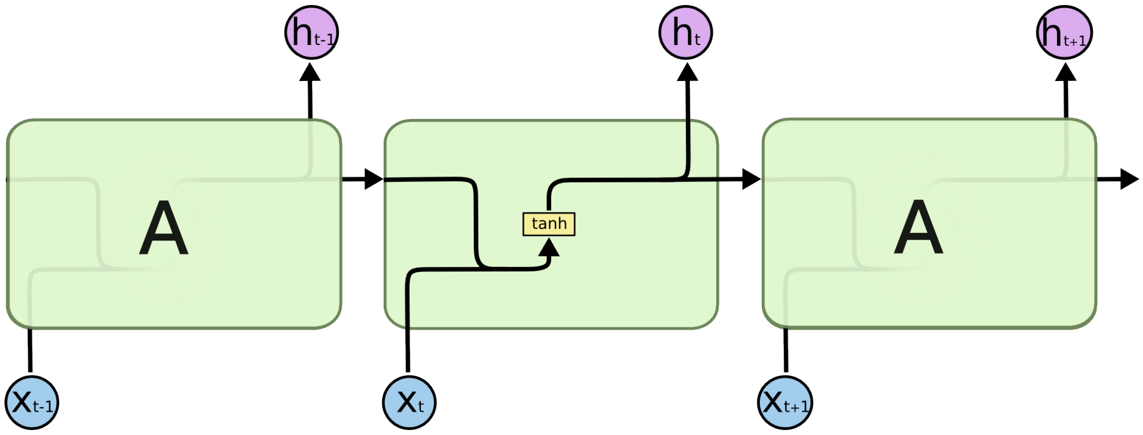


Figure 3.8: Unrolled Recurrent Neural Network chain structure [21]

When working with long-sequence data, RNNs have the issue of not being able to handle long-term dependencies. In theory, RNNs should be able to handle these dependencies, but in practice, they fail to do so. The cause of this is the vanishing gradient problem. The information from the recurrent connections are put through a non-linear activation function for each sequence, see Figure 3.8 where the precious output h_{t-1} is through the activation function tanh. Each time the stored information is put through an activation function, its values are in practice scaled down. For long sequences, this results in less and less information being passed on. To solve this problem the LSTM (Long Short Term Memory) model was presented by Hochreiter & Schmidhuber in 1997 [22]. The LSTM model was explicitly designed to avoid the long-term dependency problem.

As with RNNs, LSTMs also have a chain structure, but instead of only having a single neural network layer it has four interacting in different ways, see Figure 3.9.

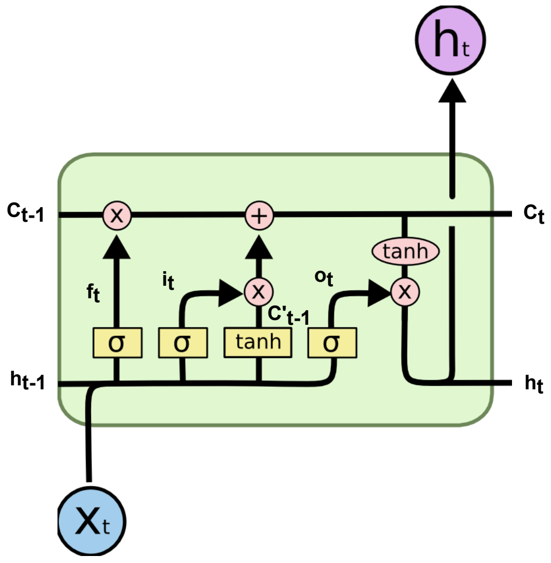


Figure 3.9: LSTM chain structure [21]

The key mechanism in LSTMs is the cell state, which is the horizontal line running across the top of the diagram in Figure 3.9. It runs through the entire chain with some minor linear interactions, meaning that it is easy for information to just flow through it unchanged. This is exactly what is wanted to be able to handle long-term dependencies. The LSTM has the ability to add or remove information from the cell state by point-wise operations. The left-most operation in the diagram is a multiplication of the cell state from the previous iteration, C_{t-1} and a sigmoid layer output f_t. This operation decides how much of the previous cell state we want to keep in our current iteration and is called the “forget gate layer”. It looks at the previous hidden state h_{t-1} and the current input x_t, and outputs a number between 0 and 1 for each number in the cell state, see Equation (3.7).

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{3.7}$$

If the model decides to keep nothing from the previous state it is multiplied by 0, and 1 if it wants to keep all of the previous cell state. The second operation is adding new information to the cell state from the current input. This is done through two steps, first by a sigmoid layer called the “input gate layer” that decides which values should be updated (i_t). Secondly, candidate values (C_t') that should be added to the cell state are computed, see Equation (3.8). These two vectors are then combined through multiplication and added to the cell state, as seen in Equation (3.9).

$$\begin{gathered} i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \\ C_t' = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \end{gathered} \tag{3.8}$$

$$C_t = f_t * C_{t-1} + i_t * C_t' \tag{3.9}$$

Lastly, the new hidden state h_t, also called the output, will be computed. The output is based on the updated cell state C_t, but a filtered version of it. A sigmoid layer produces o_t which decides what parts of the cell state will be in the output. The cell state is put through the activation function tanh to transform the values to be between -1 and 1. The output is then the product of the filtering index term o_t and the transformed cell state, see Equation (3.10).

$$\begin{gathered} o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \\ h_t = o_t * \tanh(C_t) \end{gathered} \tag{3.10}$$

3.3.4 Transformer

As presented in Section 2.2, when the Transformer model was published it paved a new way of solving NLP-related tasks by using a sequence transduction model [8]. Previously recurrent nets, e.g. LSTM, were the most popular models for these tasks. However, the authors had an idea for a new kind of solution. Instead of iteratively going over the sequence and recurrently combining the new input with the previous output, they just “put attention over everything”.

The transformer in Figure 3.10 consists of an encoder (left gray box) and a decoder (right gray box). The encoder is given the source sentence and the decoder is given the current target sentence. It could be a sentence where the two first words are already produced but is missing the last three words. The decoder is followed by a linear and softmax layer that outputs the token at a specified position for the target sentence. What makes this efficient is that each single output token is one sample as the backpropagation is only dependent on the given source and target sentence. For recurrent models, the entire sentence to sentence is one sample as it backpropagates through all the recurrent steps.

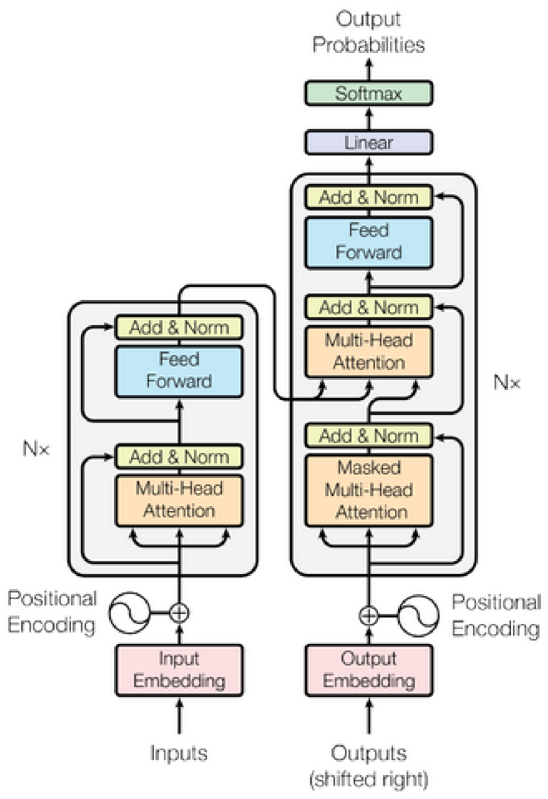


Figure 3.10: Transformer architecture [8]

Both the encoder and decoder begin with an embedding layer followed by positional encoding. As the recurrent property is not present in the encoder or decoder the positional information is not taken into consideration. Therefore, the positions of the words are encoded into the embeddings using trigonometrical functions as seen in Equation (3.11).

$$\begin{aligned}\mathrm{PE}_{(\mathrm{pos},2i)} &= \sin\left(\frac{\mathrm{pos}}{10000^{2i/d_{\mathrm{model}}}}\right)\\ \mathrm{PE}_{(\mathrm{pos},2i+1)} &= \cos\left(\frac{\mathrm{pos}}{10000^{2i/d_{\mathrm{model}}}}\right)\end{aligned} \tag{3.11}$$

The three orange blocks in Figure 3.10 are utilizing the central attention mechanism. The implemented attention method is called "Scaled dot-product Attention" and "Multi-head Attention" as seen in Figure 3.11. The three inputs to each of these attention modules are Q (queries), K (keys), and V (values). For the top right orange module, the V and K are given from the encoder (part from the source sentence). The Q comes from the decoder (target sentence). The "Masked multi-head self-attention" means that some positions in the decoder input are masked and thus ignored by the self-attention layer. In practice, the attention function is computed on a set of queries simultaneously, packed together into a matrix Q. The keys and values are also packed together into matrices K and V. The matrix of the outputs is computed as:

$$\text{Attention}(Q, K, V) = \text{Softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{3.12}$$

The equation is described in a flow chart in Figure 3.11. Matrix multiplication between Q and K^T yields the pair-wise dot-product between their row vectors. In high dimensions, this product will be close to 90°(product value around 0). However, if the row vectors in Q and K are close to aligning it will be non-zero. The scaling by d_k is done to prevent pushing the softmax function into regions where it has extremely small gradients (vanishing gradient problem). If K_i and Q_i align with each other, the softmax will yield a high value for index i, and multiplying this with V yields a high value for V_i. In a way selecting V_i.

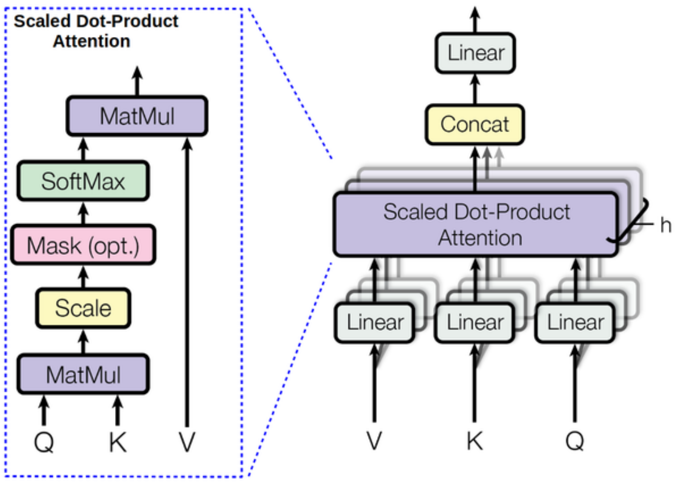


Figure 3.11: Attention concept used in the Transformer [8]. Scaled Dot-Product Attention (left) and Multi-head Attention (right)

The encoder builds Key-Value pairs for the source sentence while the decoder builds Queries for the target sentence. Keys can be viewed as a way to address the Values, and the Queries can be viewed as information the decoder wants to know about the target sentence.

3.4 Model Evaluation

For this thesis, the performance of the researched methods and models will be measured by comparing the predicted labels of the baseband units to the ground truth labels assigned by the SMEs.

3.4.1 F1-score

The main metric used for evaluating the classification performance for this thesis is the *F1 score*. This performance metric is also the one primarily used by the team at Ericsson which makes comparing models and methods easier. It is calculated from the *precision* and *recall* of the results.

Precision is the number of true positives TP divided by the number of all positive results. Since all positive results is the sum of all true positives TP and false positives FP, we can calculate the precision as

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{3.13}$$

Recall is the number of predicted true positives divided by the number of all positives. Since all positives is the sum of all true positives TP and false negatives FN, we can calculate the recall as

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{3.14}$$

The F1-score F_1 is then computed as the harmonic mean of the precision and recall, such that

$$F_1 = \frac{2}{\text{Precision}^{-1} + \text{Recall}^{-1}} \tag{3.15}$$

In a multiclass setting, the *F1-score* is calculated for each class in a binary classification fashion, where one specific class is considered the positive, and all other classes are considered negative. These scores can then be presented per class or averaged in different ways, such as **macro** or **weighted**. The macro version takes the average of the scores of each separate class without considering the class proportions in the data, while the weighted version takes the weighted average with regard to the proportion for each class.

3.4.2 Shapley and SHAP values

While some evaluation metrics focus on quantifying the predictive power or strength of a learner or classifier, some metrics also offer insight into their *interpretability*. A common scenario is, given a classifier, some input features, and a prediction, to explain in what way the input features drove the classifier to make the prediction.

I.e., can we get insight into how the various input features affected, or caused the model output, by perhaps assigning it some importance value?

One such metric is the **Shapley value**, which originally was devised as a *solution concept* in cooperative game theory [3]. A solution concept is itself a rule predicting how a game will be played.

Consider the set of all players N, and some coalition of players $S \subseteq N$. Then let the real-valued function $v(S)$ denote the worth or the total expected surplus generated by the coalition S.

The Shapley value is then a way of fairly distributing this worth among players [23]. It gives the worth φ_i given to player i as

$$\varphi_i(v) = \sum_{S \subseteq N \setminus \{i\}} \frac{|S|!\ (|N| - |S| - 1)!}{N!}\ (v(S \cup \{i\}) - v(S)). \tag{3.16}$$

We can explain this formula as follows: We consider all possible coalitions of players that can be formed without i, We then calculate their contribution by taking the difference in worth between the coalition with, and without the player i. This is averaged over the number of possible ways of forming the coalition. To highlight this fact, we can also rewrite it as

$$\begin{aligned} \varphi_i(v) &= \frac{1}{|N|} \sum_{S \subseteq N \setminus \{i\}} \binom{|N| - 1}{|S|}^{-1} (v(S \cup \{i\}) - v(S)) = \\ &= \frac{1}{\text{no. players}} \sum_{\substack{\text{coalitions} \\ \text{without } i}} \frac{\text{marginal contribution of } i \text{ to coalition}}{\text{no. coalitions without } i \text{ of this size}}. \end{aligned} \tag{3.17}$$

For a classification model, we can instead consider N to be a set of input features, and $v(S)$ to be the model output for some combination of input features S. E.g. for binary classification where $v(S) \in [0, 1]$ with some decision boundary of say 0.5, we could have that $\varphi_i(v) < 0$, indicating that i contributes to a prediction of the 0 class, or $\varphi_i(v) > 0$, indicating that i contributes to a prediction of the 1 class.

One practical issue with this approach is the number of possible coalitions there are to consider. As seen in Equation (3.16), there are $2^{|N|}$ possible coalitions, which means that we would need to fit $2^{|N|}$ classifier models. This is in practice completely unfeasible for even the simplest models as the number of features grows.

Building of the classical Shapley values, there exists a framework specifically made for interpreting machine learning model predictions called **SHAP** (<u>SH</u>apley <u>A</u>dditive ex<u>P</u>lanations) [2]. These are based on the Shapley Values, which are optimal from a game-theoretical point of view but implement several efficient estimation methods to make the calculations practically feasible. Among these estimations exists estimations methods for specific models, like a framework specifically for Ensemble models like Random Forest and XGBoost (Sections 3.3.1 and 3.3.2) [24]. This takes advantage of the inherent architecture of the models to reduce the number of calculations needed, while still offering a usable estimate of the Shapely values.

3.4.3 Cross-validation

When testing the capabilities of a predictive model, it is important to separate the data into training- and test sets. We first fit the model parameters on the training set and then evaluate the predictive capabilities on the hold-out test set. The reason for this is to make sure the model *generalizes* on unseen data, which in practice is what it will be used on in the future.

For this reason, we would like to select model parameters using the training set in such a way that we maximize the capabilities of the model to generalize. The selection of model parameters can be driven by selecting different combinations of parameters and then evaluating some metrics like the F1-score of the model to rank which combination yielded the best result. However, if we use the test set to evaluate these metrics for several combinations of parameters, and then select the best one, we have in fact *over-fitted* the model on the test data. I.e., we have selected model parameters that specifically work well for the test set, but not necessarily new unseen data.

It is here we introduce the so-called *validation* set. This is created by splitting the training set into a new training set and a validation set. We can now fit the model on the new training set, and evaluate model parameters on the validation set to select the best combination. Finally, we can then test the best model on the unseen data in the hold-out test set to see how well the model actually generalized.

However, it could still be possible that the selected validation set might not be completely representative of the entire dataset. I.e., by evaluating our model parameters using it we might end up with a sub-par model. To further mitigate this we can perform *k-fold cross-validation.* Instead of splitting the original training set into only one training and validation set, we do it consecutively for several *splits* k by dividing it into k separate folds. For each of these splits, we let one fold act as the validation set, and the remaining $k-1$ act as the training set, as seen in Figure 3.12. We can then test one combination of model parameters by training and evaluating on each of these k splits and take the mean test metric (e.g. F1-score) to see how well it generalizes over the entire training dataset. This further reduces the chances of overfitting but does of course introduce more complexity in the fitting stage, as we fit the model k times instead of only 1.

3.4.4 Bootstrapping

Bootstrapping is a resampling technique in which random sampling with replacement can be carried out on a single sample dataset to create additional simulated datasets. These simulated datasets can then be used to estimate statistics about the entire population of which the original sample dataset is part. If the sample dataset is large enough, and samples are drawn properly (as described), the resulting statistics calculated over these sample datasets can be used to calculate e.g. the mean or standard deviation of the statistic over the population. The technique is summarized as pseudo-code in Algorithm 3.2. For example, if the chosen statistic is the accuracy

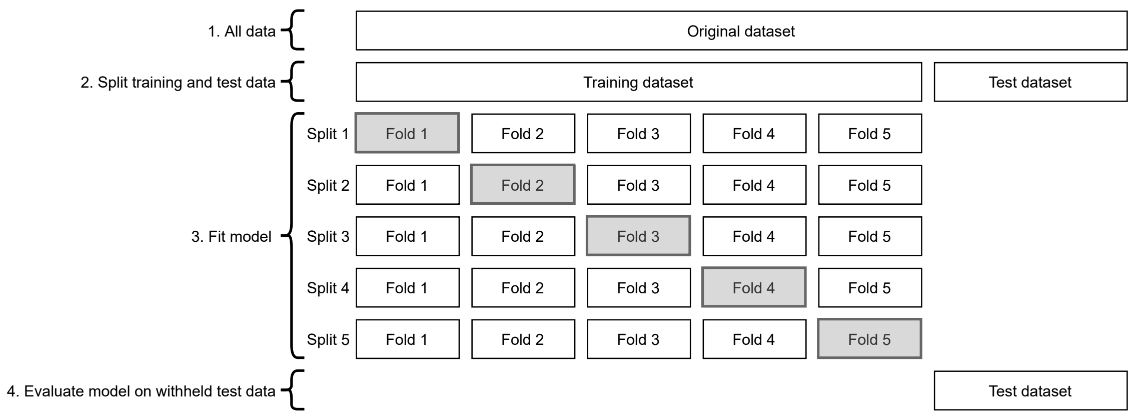


Figure 3.12: Visualization of 5-fold cross-validation data splitting. After the initial split intro training and test, the training set is further divided into k splits containing k folds of the data each. For each split, one of the folds is used as the validation set, while the remaining $k-1$ folds are used for training.

of some classifier, it can be used to estimate its mean and standard deviation over the dataset.

Algorithm 3.2 Bootstrap

```
Input: Dataset $X = \{x_1, x_2, ..., x_n\}$, Sample size $m$, Bootstrap samples $k$
Output: List of statistics $Y = \{y_1, y_2, ..., y_k\}$
1: Initialize empty list $Y \leftarrow \emptyset$
2: for Bootstrap iteration up to $k$ do
3:     Draw $m$ samples $X_{\text{Sample}}$ with replacement and uniform probability from $X$
4:     Calculate statistic $y$ from $X_{\text{Sample}}$ and add this to $Y$
5: return $Y$
```

4

Methods

In this chapter, we present our methodology for data ingestion, feature extraction, and model selection for the project. In Section 4.1 we introduce the data structure for parsing, processing, and representation of the data for the models. In Section 4.2 we present the different feature representations extracted from the logs. In Section 4.3 the workflow and model of the main baseline solution are presented. In Section 4.4 other more experimental solutions and their methods that were interesting to investigate are presented.

4.1 Data ingestion

The data ingestion is handled by a data pipeline created for this thesis. The data pipeline houses functionality for parsing and processing the data as well as the representation of the data for the models. The initial dataset contains a collection of units with corresponding labels, where each unit has several log files represented as text files with the lines of the log. Several preprocessing steps are common among the investigated solutions and are therefore presented in this section.

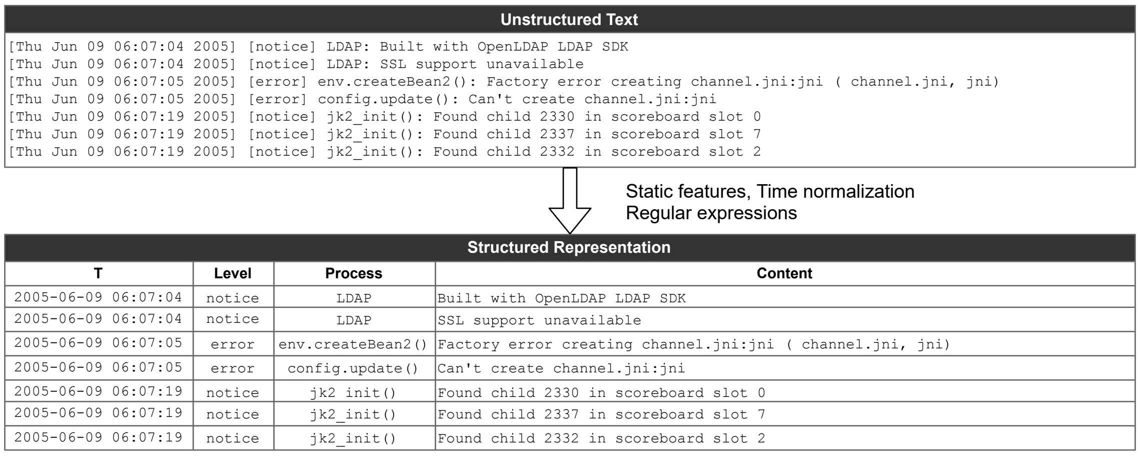
4.1.1 Data Structure

During the initial data ingestion and feature extraction, the log data and the features are stored in two custom data structures we denote `Unit` and `Log`. We let $\mathcal{X} = \{X_1, X_2, ..., X_n\}$ denote the dataset as a list of `Unit` objects, and $\mathcal{Y} = \{y_1, y_2, ..., y_n\}$ denote the corresponding classification labels where $y_i \in \{\texttt{NF}, \texttt{SW}, \texttt{HW}\}$.

Each `Unit` $X \in \mathcal{X}$ is itself capable of containing multiple `Log` objects of different log types. E.g., a unit might contain both the `hwlog` and `llog`, but perhaps not the `ailog`. Each such `Log` object contains both the original text lines, as well as various features extracted from these lines such as vectorized word-count embeddings or parameter-list aggregates (see Section 4.2). Consider a `Log` object L, with N lines of log entries, and some feature embedding dimension M. We make the distinction between two main feature categories: A *log-level* feature $F_{\text{Log}}(L)$ represents the entire log file as one numerical vector, such that $F_{\text{Log}}(L) \in \mathbb{R}^M$. A *line-level* feature embedding $F_{\text{Line}}(L)$ creates feature vectors for each line in the log file as $F_{\text{Line}}(L) \in \mathbb{R}^{N \times M}$.

4.1.2 Basic Regex Parsing

The first step in the data pipeline is regex parsing of the logs. This is to transform the logs from an unstructured representation into a structured representation as seen in Figure 4.1. As each of the log types have their data structured differently, individual regex patterns had to be written for each of the log types.


Unstructured Text
Static features, Time normalization
Regular expressions
Structured Representation

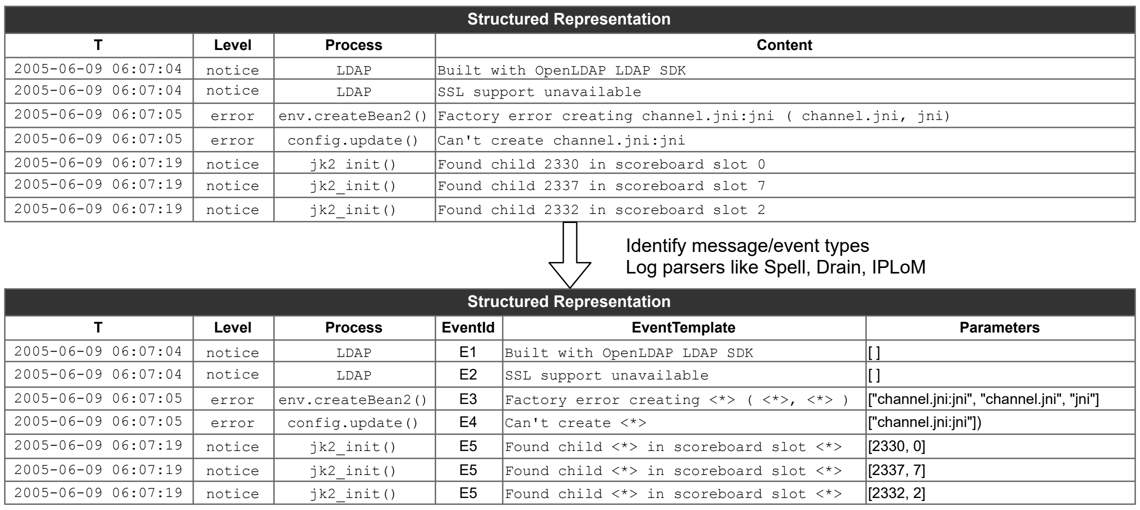
```
[Thu Jun 09 06:07:04 2005] [notice] LDAP: Built with OpenLDAP LDAP SDK
[Thu Jun 09 06:07:04 2005] [notice] LDAP: SSL support unavailable
[Thu Jun 09 06:07:05 2005] [error] env.createBean2(): Factory error creating channel.jni:jni ( channel.jni, jni)
[Thu Jun 09 06:07:05 2005] [error] config.update(): Can't create channel.jni:jni
[Thu Jun 09 06:07:19 2005] [notice] jk2_init(): Found child 2330 in scoreboard slot 0
[Thu Jun 09 06:07:19 2005] [notice] jk2_init(): Found child 2337 in scoreboard slot 7
[Thu Jun 09 06:07:19 2005] [notice] jk2_init(): Found child 2332 in scoreboard slot 2
```

T	Level	Process	Content
2005-06-09 06:07:04	notice	LDAP	Built with OpenLDAP LDAP SDK
2005-06-09 06:07:04	notice	LDAP	SSL support unavailable
2005-06-09 06:07:05	error	env.createBean2()	Factory error creating channel.jni:jni (channel.jni, jni)
2005-06-09 06:07:05	error	config.update()	Can't create channel.jni:jni
2005-06-09 06:07:19	notice	jk2_init()	Found child 2330 in scoreboard slot 0
2005-06-09 06:07:19	notice	jk2_init()	Found child 2337 in scoreboard slot 7
2005-06-09 06:07:19	notice	jk2_init()	Found child 2332 in scoreboard slot 2

Figure 4.1: Example of transforming a unstructured log into a structured representation using regex patters.

4.1.3 Advanced Structured Parsing

After the logs have been processed into a structured format, a more advanced parsing method such as Spell or Drain is applied to the structured representation of the logs (see Sections 3.1.1 and 3.1.2). This produces the EventIds, EventTemplates, and parameter lists as can be seen in Figure 4.2.


Structured Representation
Identify message/event types
Log parsers like Spell, Drain, IPLoM
Structured Representation

T	Level	Process	Content
2005-06-09 06:07:04	notice	LDAP	Built with OpenLDAP LDAP SDK
2005-06-09 06:07:04	notice	LDAP	SSL support unavailable
2005-06-09 06:07:05	error	env.createBean2()	Factory error creating channel.jni:jni (channel.jni, jni)
2005-06-09 06:07:05	error	config.update()	Can't create channel.jni:jni
2005-06-09 06:07:19	notice	jk2_init()	Found child 2330 in scoreboard slot 0
2005-06-09 06:07:19	notice	jk2_init()	Found child 2337 in scoreboard slot 7
2005-06-09 06:07:19	notice	jk2_init()	Found child 2332 in scoreboard slot 2

T	Level	Process	EventId	EventTemplate	Parameters
2005-06-09 06:07:04	notice	LDAP	E1	Built with OpenLDAP LDAP SDK	[ ]
2005-06-09 06:07:04	notice	LDAP	E2	SSL support unavailable	[ ]
2005-06-09 06:07:05	error	env.createBean2()	E3	Factory error creating <*> (<*>, <*>)	["channel.jni:jni", "channel.jni", "jni"]
2005-06-09 06:07:05	error	config.update()	E4	Can't create <*>	["channel.jni:jni"])
2005-06-09 06:07:19	notice	jk2_init()	E5	Found child <*> in scoreboard slot <*>	[2330, 0]
2005-06-09 06:07:19	notice	jk2_init()	E5	Found child <*> in scoreboard slot <*>	[2337, 7]
2005-06-09 06:07:19	notice	jk2_init()	E5	Found child <*> in scoreboard slot <*>	[2332, 2]

Figure 4.2: Example of information extracted from a structured log by log parsers.

4.1.4 Tokenizing

Several embeddings methods, like BOW, TFIDF, and Word2Vec, all work by embedding text data in the form of tokens. This requires that the lines of the logs are divided into individual tokens through *tokenization*. In normal NLP circumstances, this could be achieved by simply splitting a line by white spaces and other special characters. However, for logs, there are usually many more characters that can represent boundaries between potential tokens, such as brackets, parentheses, equal signs, etc. Further, several potential tokens might pose issues for tokenizers, such as numbers, timestamps, or IP addresses. A token consisting of a single number could for example be any number between zero and a few million, which would lead to vocabulary sizes increasing uncontrollably, i.e. they have a high *cardinality*. A solution to this is by first filtering out these values from the string using regular expressions, and replacing them with special tokens. Consider the example in Table 4.1. Here we identify that the timestamp of the log entry has a high cardinality, and potentially the two numbers `2330` and `0`. For this reason, we write a simple regular expression to find these types of entries and replace them with the special tokens `<T>` for timestamps and `<N>` for number. We call this step the *Filter* step. After this step, we identify the typical delimiters for this log type. Here we chose white spaces, brackets, parentheses, and colons. By splitting on these delimiters we get a list of tokens in the *Token* step. Finally, we can also apply lower casing in the *Lowercase* step to further reduce the size of the vocabulary.

For each log type, we specify these filter patterns to replace potentially problematic tokens like numbers, as well as lists of delimiters that are typical for each log type. Since this configuration is only specified once per unique log type in the dataset, it is feasible to do it manually, and without domain knowledge about the log contents.

Table 4.1: Example of how tokenization is performed on a log line.

Step	Data
Input	`[Thu Jun 09 06:07:19 2005] [notice] jk2_init(): Found child 2330 in scoreboard slot 0`
Filter	`<T> [notice] jk2_init(): Found child <N> in scoreboard slot <N>`
Token	`['<T>', 'notice', 'jk2_init', 'Found', 'child', '<N>', 'in', 'scoreboard', 'slot', '<N>']`
Lowercase	`['<t>', 'notice', 'jk2_init', 'found', 'child', '<n>', 'in', 'scoreboard', 'slot', '<n>']`

4.2 Feature Extraction

Various feature extraction methods of the logs have been implemented for this thesis. While some of these are commonly used in NLP settings, some are also related to more classical data mining approaches used in log analysis. They are all presented in Table 4.2.

4.2.1 Word token counts

The entire log file is represented by a BOW (Section 3.2.1) or TFIDF (Section 3.2.2) representation of the word tokens present in the log. This is computed using a `CountVectorizer` from the Python library Scikit-learn [18]. All the log files are

Table 4.2: Feature representations of the logs

Code	Feature Name	Description
`word`	Word token counts	BOW or TFIDF representation of the tokens in the logs
`event`	Event token counts	BOW or TFIDF representation of the EventIds in the logs from log parsers
`param`	Parameter aggregates	Aggregates of parameter extracted from logs by log parsers
`waemb`	Weighted Average Embedding	Continuous Word2Vec line embeddings

viewed as a corpus where a log file is represented as a list of lines, where each line is a list of tokens. It is also capable of producing N-grams of the tokens within the same sentence (line). Each log type has its own BOW vocabulary and does not share it with the other log types. Pseudocode for this feature extraction process can be seen in Algorithm 4.1.

Algorithm 4.1 Word token count embedding

```
Input: List of logs $\mathbf{L}$, embedding dimension $d$
Output: List of logs $\mathbf{L}$ with word count feature added
 1: Initialize empty list $X_{\text{Logs}}$
 2: for Every log $L \in \mathbf{L}$ do
 3:     Initialize empty list $X_{\text{Log}}$
 4:     for Every line $l \in L$ do
 5:         for Every token $w \in l$ do
 6:             Append $w$ to $X_{\text{Log}}$
 7:     Append $X_{\text{Log}}$ to $X_{\text{Logs}}$
 8: Generate BOW or TFIDF embedings $X \in \mathbb{R}^{|L| \times d}$ from $X_{\text{Logs}}$
 9: for Index $i \leftarrow 0$ to $|\mathbf{L}|$ do
10:     Store row vector $X_i$ in log-level features of $L_i$
11: return $\mathbf{L}$
```

4.2.2 Event token counts

The log parsers Spell and Drain produces EventIds (see Section 4.1.3) which are representations of the unique message types of each line in the log files. To represent the sequences of the lines, i.e. in what order these message types appear in the log, we can compute N-grams of these EventIds. This yields information of both the occurrences of individual lines with 1-grams, but also the occurrences of line sequences with e.g. 2-grams or 3-grams. Consider the example where a log file contains the sequence of EventIds `[E1, E2, E1, E3]`. A 1-gram representation of this sequence would tell how many of each message type was found in the log file.

However, it might be the case that event `E3` very rarely should follow event `E1`, where a 2-gram representation would be able to capture this feature with the 2-gram `(E1, E3)`. We wanted to include this type of feature to examine if the sequence of log entries might be indicative of the fault type of the unit. Each of the log types has its own vocabulary of the N-grams represented using BOW or TFIDF. Pseudocode for this feature extraction process can be seen in Algorithm 4.2.

Algorithm 4.2 Event token count embedding

```
Input: List of logs $\mathbf{L}$, embedding dimension $d$
Output: List of logs $\mathbf{L}$ with event count feature added
 1: Initialize empty list $X_{\text{Logs}}$
 2: for Every log $L \in \mathbf{L}$ do
 3:     Initialize empty list $X_{\text{Log}}$
 4:     for Every EventId $e \in L$ do
 5:         Append $e$ to $X_{\text{Log}}$
 6:     Append $X_{\text{Log}}$ to $X_{\text{Logs}}$
 7: Generate BOW or TFIDF embedings $X \in \mathbb{R}^{|L| \times d}$ from $X_{\text{Logs}}$
 8: for Index $i \leftarrow 0$ to $|\mathbf{L}|$ do
 9:     Store row vector $X_i$ in log-level features of $L_i$
10: return $\mathbf{L}$
```

4.2.3 Parameter aggregates

The log parsers also extract parameter lists from the lines in the log files. Only the numerical parameters are kept, and for each such series of numerical values, several *aggregates* are calculated. These are the mean, standard deviation, minimum, and maximum of the values. These are then concatenated over all EventIds for the log files into a feature vector representing the numerical values of the parameter lists in each log. Pseudocode for this feature extraction process can be seen in Algorithm 4.3.

4.2.4 Weighted Average Embedding

To create an embedding for an entire log line we created a method we call WAEMB (Weighted Average Embedder). It utilizes a Word2Vec or FastText model which has been trained on the tokens in the logs. For each line, each of the tokens in that line is embedded into a d-dimensional vector. The embedding for the entire line is then the weighted average of all the embedded tokens which is used to represent a line in a log file, see Figure 4.3. Note that there is one such embedding model for each of the log types. Pseudocode for this feature extraction process can be seen in Algorithm 4.4.

4.3 Baseline Model

The baseline solution is the main solution developed during the thesis. It was originally thought of as an initial solution using very simple NLP techniques to get a better understanding of how complex the classification task might be. Over time

Algorithm 4.3 Parameter aggregate embeddings

Input: List of logs $\mathbf{L}$

Output: List of logs $\mathbf{L}$ with parameter aggregate feature added

```
 1: Find all unique events $\mathbf{e}$ present in the logs $\mathbf{L}$
 2: Find corresponding size $\mathbf{s}$ of the parameter list for each unique event
 3: Create mapping $f(e, j) \mapsto g \in F$ from an events $e \in \mathbf{e}$ and indices $j = 1, ..., s_e$ in
    parameter list to a new name $g$.
 4: Let $d \leftarrow$ number of new feature names $|F|$
 5: for Every log $L \in \mathbf{L}$ do
 6:     Let $n \leftarrow$ number of lines in log
 7:     Initialize parameter matrix $X \in \mathbb{R}^{n \times d}$ filled with zeros
 8:     for Every line index $i \leftarrow 0$ to $n$, event $e \in L$ and parameters $\mathbf{p} \in L$ do
 9:         for Index $j \leftarrow 0$ to $|\mathbf{p}|$ do
10:             Let $g \leftarrow f(e, j)$ be the new feature name
11:             Let $X_{ig} \leftarrow p_j$
12:     Initialize empty list of aggregate features $X_{\text{Agg}}$
13:     for Feature name $g \leftarrow 0$ to $d$ do
14:         Extract parameter series column $x \leftarrow X_{:,g}$
15:         if $x$ contains only numeric values then
16:             Append mean of $x$ to $X_{\text{Agg}}$
17:             Append standard deviation of $x$ to $X_{\text{Agg}}$
18:             Append minimum of $x$ to $X_{\text{Agg}}$
19:             Append maximum of $x$ to $X_{\text{Agg}}$
20:     Store matrix $X$ in line-level features of $L$
21:     Store matrix $X_{\text{Agg}}$ in log-level features of $L$
22: return $\mathbf{L}$
```

its performance increased to a point where it had to be considered to be the main solution for this thesis.

The prediction model chosen for the baseline solution is the gradient boosting model XGBoost [4]. As described in Section 4.1, each unit could contain multiple types of log data. Since the prediction is at unit-level, and not necessarily log-level, the prediction model needs to be able to handle multimodal data. Therefore two different types of architectures using XGBoost have been implemented in this thesis that can work with multiple data types. The main one is utilizing early fusion, and the other is an alternative variant that utilizes late fusion.

4.3.1 Early fusion XGBoost model

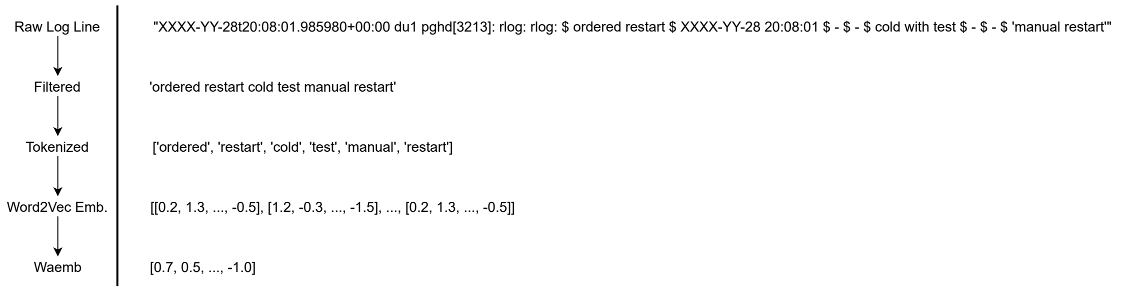
The main baseline model architecture uses early fusion where it concatenates feature vectors of each log type in a unit into a single feature vector for the entire unit. For example, taking the `word` feature vectors for the log types in the unit and concatenate them into one vector that represents the entire unit. This feature vector is then given to an XGBoost model for classification, see Figure 4.4.

Algorithm 4.4 Weighted Average Embedding

Input: List of logs **L**, embedding dimension d

Output: List of logs **L** with weighted average embedding feature added

```
 1: Initialize empty list $X_{\text{Lines}}$
 2: for Every log $L \in \mathbf{L}$ do
 3:     for Every line (tokenized list) $l \in L$ do
 4:         Append $l$ to $X_{\text{Lines}}$
 5: Fit Word2Vec or FastText model on $X_{\text{Lines}}$
 6: for Every log $L \in \mathbf{L}$ do
 7:     $n \leftarrow$ number of lines in log
 8:     Initialize embedding matrix $X \in \mathbb{R}^{n \times d}$
 9:     for Every line $l \in L$ and index $i \leftarrow 0$ to $n$ do
10:         $m \leftarrow$ number of tokens $|l|$ in line
11:         Generate embeddings $X_{\text{Line}} \in \mathbb{R}^{m \times d}$ from $l$
12:         Create column-wise average $\hat{X}_{\text{Line}} \in \mathbb{R}^{d}$ from $X_{\text{Line}}$
13:         Let $X_i \leftarrow \hat{X}_{\text{Line}}$
14:     Store matrix $X$ in line-level features of $L$
15: return L
```



```
"XXXX-YY-28t20:08:01.985980+00:00 du1 pghd[3213]: rlog: rlog: $ ordered restart $ XXXX-YY-28 20:08:01 $ - $ - $ cold with test $ - $ - $ 'manual restart'"
'ordered restart cold test manual restart'
['ordered', 'restart', 'cold', 'test', 'manual', 'restart']
[[0.2, 1.3, ..., -0.5], [1.2, -0.3, ..., -1.5], ..., [0.2, 1.3, ..., -0.5]]
[0.7, 0.5, ..., -1.0]
```

Figure 4.3: Workflow of the Weighted Average Embedding (WAEMB) method

4.3.2 Late fusion XGBoost model

The late fusion variant does not concatenate the feature representation of the log types but instead has one XGBoost model per log type. Each log type's feature representation is given to an XGBoost model for classification. This yields a prediction for each log type. The classification for the entire unit is made by a softmax layer of the summed prediction probabilities of each model, see Figure 4.5.

4.3.3 Model Selection

For both the early- and late fusion variants of the baseline model, there are several combinations of features and hyperparameters to consider:

1. **What logs should be used?** It might e.g. be the case that only a subset of the logs actually contains useful information. Theoretically, we have $2^5 - 1 = 31$ different combinations of logs to consider.

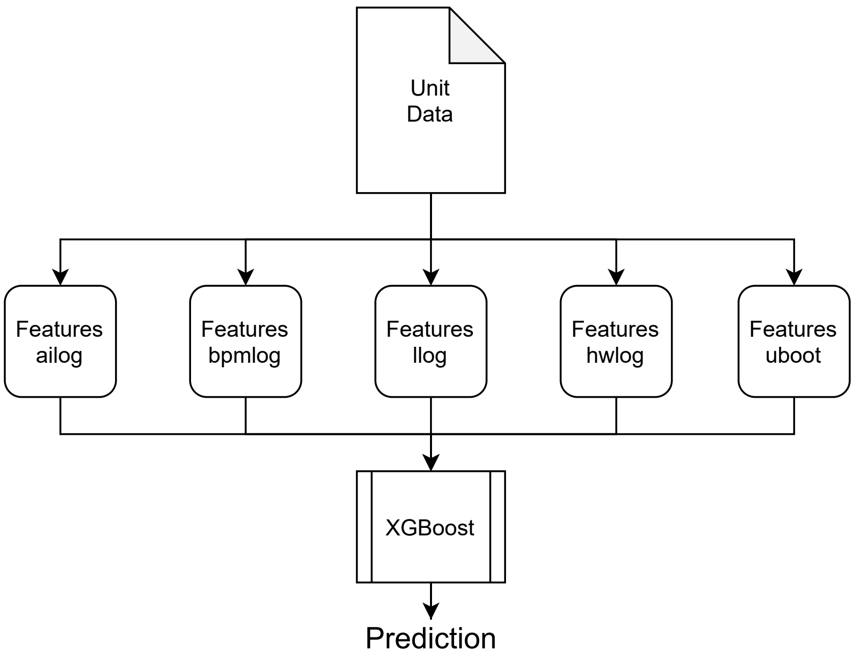


Figure 4.4: Architecture of the implemented early fusion XGBoost model

2. **What features should be used?** If we consider the features of **word token counts**, **event token counts**, and **parameter aggregates**, there are $3! = 6$ combinations for each log file.

3. **What feature parameters should be used?** In the case of performing BOW or TFIDF on e.g. words or events in the log files we e.g. need to consider the maximum vocabulary size, and the n-gram ranges. If we would like to try both BOW and TFIDF, 3 different vocabulary sizes, and 2 n-gram ranges, this would mean $2 \times 3 \times 2 = 12$ different combinations of parameters (for only the word count feature).

4. **What model parameters should be used?** Finally, we must also consider the actual hyperparameters of the classification model. For an XGBoost model, we can consider e.g. the max depth, learning rate (eta), and minimum split loss (gamma) parameters, among others, which could practically yield around 30 combinations to consider. Optimally, these parameters are determined through k-fold cross-validation on the training data. For 5-fold cross-validation, this would mean fitting the model 5 separate times.

In total considering these options, we would have somewhere around $31 \times 6 \times 12 \times 30 \times 5 = 334\ 800$ total model training iterations on the data to carry out. This also assumes that we use the same feature parameters for each feature type in all log types (i.e. not BOW for the words in some log types, and TFIDF in others). This is of course not feasible, so we need to prune the search space using a few assumptions:

1. We only fit the model on **all log types** at the same time. We assume that if a log file doesn't contain useful features, it will not lead to useful partitions in

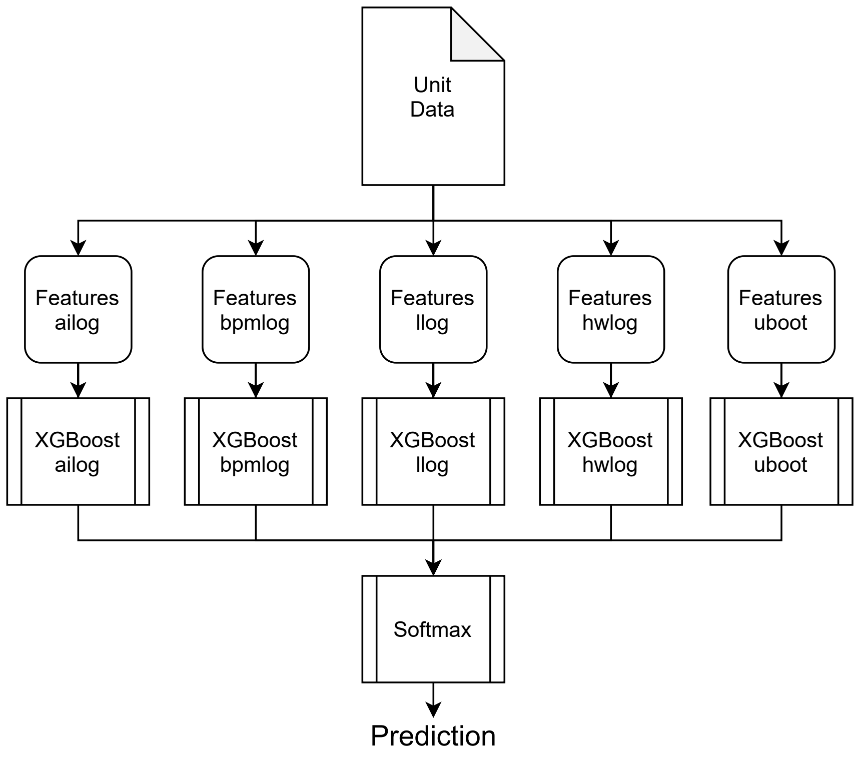


Figure 4.5: Architecture of the implemented late fusion XGBoost model

the leaves of the XGBoost model. We can then inspect the feature importances of the model and investigate which log files contributed most to the predictions. In addition, we assume that the model is not hindered by the addition of additional, potentially non-beneficial log files.

2. We separately investigate the feature types such that the model only uses one feature representation at a time. This is to best understand what feature parameters should be used in the next step. We test different parameters for feature extraction for each of the 3 separate feature types to best understand how well each feature works compared to each other:
   (a) For the **word token counts** we consider 12 combinations
   (b) For the **event token counts** we consider 16 combinations
   (c) For the **parameter aggregates** we consider 2 combinations
3. Finally we consider the $2^3 - 1 = 7$ combinations of the three feature types **word token counts**, **event token counts**, and **parameter aggregates**, extracted using the best feature parameters found in the previous step to best understand which ones are the most useful in relation to each other.

Including 5-fold cross validation, this yields a total of $1 \times 5 \times (12+16+2) + 5 \times 7 = 185$ fits of the model in total.

4.4 Experimental Implementations

Besides variants of the baseline implementation, other possible solutions and methods to the thesis problem were researched and tested. Due to the recent success of ML models utilizing the attention mechanism, presented in the paper *Attention is all you need*[8], a Transformer-like model was implemented and tested. Just as natural language, logs are also generated in a sequence which is an area where LSTMs have been successfully utilized. Therefore an LSTM based solution has also been implemented and tested.

4.4.1 Transformer Model

The Transformers utilized in this thesis consist of two variants, one utilizing positional encoding, and one without. A simplified visualization of the used model architecture is shown in Figure 4.6.

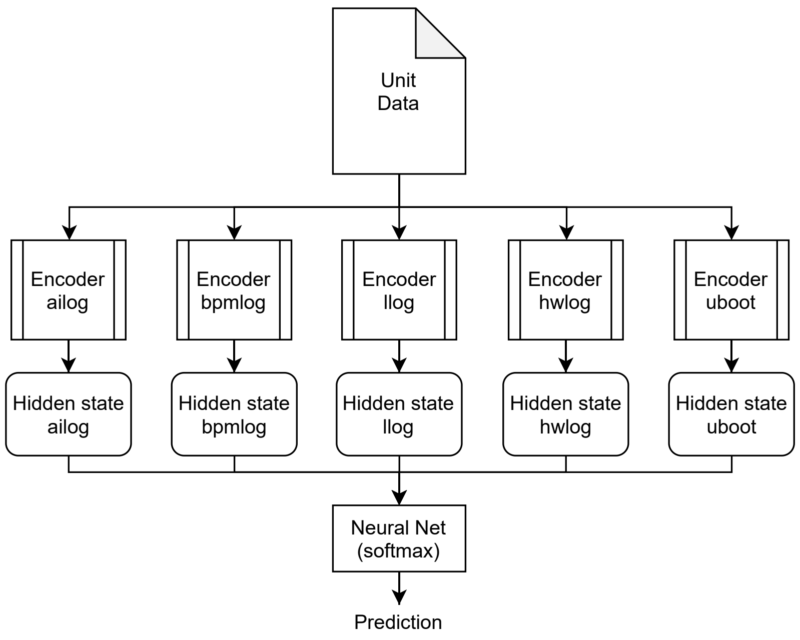


Figure 4.6: Simplified architecture of the implemented Transformer

The model architecture can be divided into two main parts, the individual log type part, and the combined log types part. Since the data contains multiple log types per unit we decided to use one Transformer encoder to process the features for each of the log types, see Figure 4.6. The data from each of the log types are given as input to these separated input layers. Here each type of the data is put through their individual multi-head attention block, global average pooling, dropout, and finally a dense layer. Each of the encoders transforms each original log representation of [512, 50] into a vector of size 30. These vectors are then concatenated into a single feature vector representing the information contained in all the logs for the unit. That vector is then put through a dense layer, dropout, and finally a softmax layer for classification.

The positional encoding is done within the encoder. Both of the model variants only utilize one multi-head attention block per log type, and not multiple blocks as are often seen in other attention-based models such as the original Transformer. This decision was made to reduce training and inference times since by adding one more multi-attention head to the model you actually add five of them, one for each log type.

Due to the length of the logs, the input to the attention layer is not the embeddings of each word in the logs, but instead, the feature representation WAEMB which is an embedding of an entire log line.

Unlike recurrent models such as the LSTM, Transformers have a specified maximum sequence length. Therefore one of the issues when working with Transformers on log data is the high variance in length. For some log types, the length of the log could be tens of lines or in the thousands. That is also why the chosen embedding representation for the Transformer has been line embeddings and not word embeddings, to further decrease the sequence length. To further simplify, if the number of lines in a log exceeds 512 only the last 512 lines are given to the model. This reduces the computation time for the Transformer and the need for large amounts of padding. The input size of 512 is commonly used in language models such as BERT and was therefore also chosen here. The reason behind choosing the 512 last lines in the log files comes from the assumption that if a unit is experiencing faulty behavior it is likely that the cause behind it occurred recently. For those units that do not contain all the log types, the input will be dummy encoded by zeros. The model does not mask the dummy encoded input, meaning that they will affect the gradient. The transformers were not trained with cross-validation, but instead on a single training split of the data with early stopping patience of seven epochs on the validation loss.

4.4.2 LSTM Model

The LSTM solution implemented in this thesis has a lot in common with the Transformer model. Again, each of the log types used is given as input to LSTM models, specific for that log type. The models run through the WAEMB representations of the logs for the unit. If some logs are not available in the unit they are dummy encoded as zeros. The last hidden state in the LSTM models from all the log types, see Section 3.3.3, are then concatenated into one single feature vector. That vector is then put through a softmax layer for classification. See Figure 4.7.

Unlike the Transformer model, the LSTM model masks the gradients of the missing log types for the units. Meaning that for units that do not have all of the log files present, the gradient will not be affected by a dummy encoded input. The LSTM model was not trained using cross-validation, but on a single train-test split with early stopping patience of seven epochs on the validation loss.

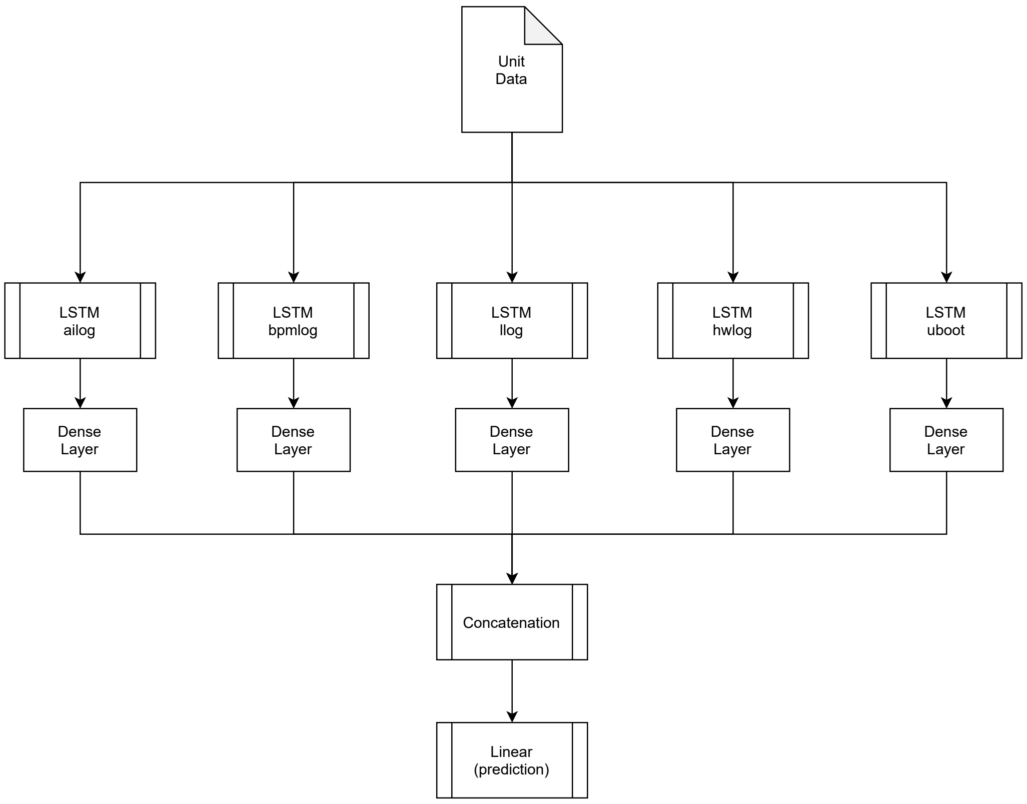


Figure 4.7: Architecture of LSTM model

4.5 Evaluation Method

The methodology for evaluation of the feature combinations and models are as follows:

1. Baseline XGBoost model:

   (a) Extract the following features from the log files in the dataset: `word`, `event`, and `param`.

   (b) Fit the early- and late fusion XGBoost models on each feature type separately using different feature parameters to see how best to extract each feature type.

   (c) Fit the early- and late fusion XGBoost models on combinations of the best features in the previous step to see which combination yields the best result

   (d) Save the best model for the best feature combination

2. LSTM and Transformer models:

   (a) Extract the `waemb` feature from the log files.

   (b) Fit the LSTM and Transformer models on the embeddings using different sizes of the embedding dimension.

   (c) Save the best models for the best embedding dimension

3. Compare the best models and corresponding feature combinations using the holdout test dataset and compare macro F1 scores, as well as F1 scores for each class.

All of the model types are trained on an oversampled training set. The oversampling is random with replacement within the three classes. This results in a balanced training dataset in regards to the classes.

4.6 Used Hardware and Software

The different models were evaluated mainly using two different platforms:

- Part of the evaluation was done on a laptop provided by Ericsson. Among its specifications is an Intel i7-8650U @ 1.90Hz with four physical cores and eight logical cores, as well as 32GB DDR4 memory.

- The LSTM model was trained on an Ericsson compute cluster using an Nvidia Tesla A100 GPU.

The software was written in Python 3.7 and major libraries used include Scikit-learn, TensorFlow and Keras, PyTorch, Nltk, Gensim, XGBoost, and HuggingFace for the Transformers.

5

Results

5.1 Baseline Models

In this section, we present the results for the baseline XGBoost models which were the first models tested. The results here represent what was achievable using the most straightforward NLP techniques for feature extraction, as well as the comparatively lightweight classification model XGBoost.

5.1.1 Early fusion XGBoost model

In Table 5.1 we see the validation set performance of the early fusion XGBoost model for different word embeddings. The tested embeddings methods are BOW and TFIDF for different numbers of features (maximum vocabulary size), as well as n-gram ranges. Here, an n-gram range of `[1,3]` means that 1-grams, 2-grams, and 3-grams are considered when building the vocabulary. Evidently, it seems both BOW and TFIDF work equally well, with n-gram ranges of either `[1,1]` or `[1,3]` as long as the maximum vocabulary size is at least 500. We note that the standard deviation of the scores is large compared to the difference between different feature representations. For this reason, we can't really say that any of the top representations is evidently better than the other.

We decide to use 1000-dimensional BOW embeddings with n-gram ranges of `[1,3]` going forward for a few reasons. It easier to interpret than the TFIDF embeddings, the added complexity of 1000 dimensions over 500 is not an issue as the model is itself quite simple and easy to run as-is. Also, by having access to 2-grams and 3-grams it might be easier to interpret the resulting features, as this gives more specific feature such as `'download starting'` and `'download done'`, compared to the disjoint feature `'download'`, `'starting'`, `'done'` which might be hard to relate with each other.

In Table 5.2 we see the validation set performance of the early fusion XGBoost model for different event embeddings. We test EventIds produced by different log parsers, and the tested embeddings methods are BOW and TFIDF for different numbers of features (maximum vocabulary size), as well as n-gram ranges. Here it seems that the most important parameter is the n-gram ranges. If we use a range of `[1, 1]`, i.e. only counting the number of different EventIds, we get the worst performance. When

Table 5.1: Resulting F1-scores of the early fusion model on 5-fold cross-validation for different word embedding methods. The table is sorted by the F1 Macro score in descending order and presented as the mean score plus-minus the standard deviation.

Mode	Features	Ngrams	F1 Macro	HW	NF	SW
tfidf	500	[1, 1]	.905±.014	.951±.006	.855±.029	.910±.020
bow	1000	[1, 1]	.905±.016	.953±.005	.849±.039	.913±.016
bow	500	[1, 1]	.905±.013	.954±.004	.847±.032	.914±.016
bow	1000	[1, 3]	.903±.013	.952±.005	.842±.031	.915±.017
tfidf	1000	[1, 3]	.900±.015	.951±.004	.846±.033	.905±.021
tfidf	1000	[1, 1]	.896±.015	.950±.004	.834±.033	.905±.019
bow	500	[1, 3]	.894±.020	.948±.007	.831±.046	.903±.013
tfidf	500	[1, 3]	.892±.019	.946±.007	.830±.044	.899±.014
tfidf	100	[1, 1]	.884±.023	.937±.006	.821±.051	.894±.020
bow	100	[1, 1]	.880±.020	.936±.005	.812±.039	.892±.022
tfidf	100	[1, 3]	.815±.013	.887±.005	.722±.040	.835±.010
bow	100	[1, 3]	.807±.011	.879±.009	.713±.029	.829±.015

we increase this range to include 2-grams and 3-grams we get better performance, indicating that the actual order of the events may play a role. Evidently, it seems both BOW and TFIDF work equally well, with n-gram ranges of either `[1,1]` or `[1,3]` as long as the maximum vocabulary size is at least 500. We note that the standard deviation of the scores is large compared to the difference between different feature representations. For this reason, we can't really say that any of the top representations is evidently better than the other.

We decide to use 1000-dimensional BOW Drain EventIds with n-gram ranges of *[1,3]* going forward. It is, once again, easier to interpret than TFIDF embeddings and the added complexity of 1000 dimensions over 500 is not an issue.

In Table 5.3 we see the validation set performance of the early fusion XGBoost model for different parameter aggregates produced by different parsers. There is really no discernible difference between the two parsers, and any of them seems to work. Of note, is that these scores seem to be on par with the scores using the EventIds in Table 5.2. Since we chose the Drain parser in that table, we choose it here as well going forward.

In Table 5.4 we see the validation set performance of the early fusion XGBoost model for different combinations of the best word, event, and parameter representations presented and chosen in Tables 5.1 to 5.3. Here it is clear that the driving feature behind the model performance is the word embeddings. There is no noticeable difference between including either event- or parameter representations of the log as long as the word representations are present. This seems to indicate that neither the order of the lines as represented using n-grams of the EventIds nor the numerical values found in the parameter lists produced by Drain offer anything which is not

Table 5.2: Resulting F1-scores of the early fusion model on 5-fold cross-validation for different event embedding methods. The table is sorted by the F1 Macro score in descending order and presented as the mean score plus-minus the standard deviation.

Parser	Mode	Features	Ngrams	F1 Macro	HW	NF	SW
drain	bow	500	[1, 3]	.824±.016	.901±.007	.736±.027	.836±.017
drain	bow	1000	[1, 3]	.824±.015	.903±.007	.727±.039	.840±.014
drain	tfidf	500	[1, 3]	.816±.008	.898±.007	.719±.014	.832±.017
drain	tfidf	1000	[1, 3]	.816±.009	.898±.010	.719±.019	.832±.010
spell	tfidf	500	[1, 3]	.815±.017	.895±.011	.718±.024	.833±.023
spell	bow	1000	[1, 3]	.815±.023	.890±.010	.723±.042	.831±.025
spell	tfidf	1000	[1, 3]	.815±.017	.890±.009	.723±.026	.830±.021
spell	bow	500	[1, 3]	.808±.033	.886±.015	.710±.064	.827±.029
drain	bow	1000	[1, 1]	.807±.009	.883±.015	.716±.025	.823±.027
drain	bow	500	[1, 1]	.800±.012	.880±.014	.700±.048	.819±.016
drain	tfidf	500	[1, 1]	.792±.010	.876±.009	.691±.030	.808±.008
drain	tfidf	1000	[1, 1]	.791±.004	.877±.009	.696±.013	.800±.011
spell	bow	1000	[1, 1]	.789±.020	.871±.006	.683±.045	.813±.019
spell	bow	500	[1, 1]	.788±.019	.868±.007	.684±.041	.813±.022
spell	tfidf	1000	[1, 1]	.787±.011	.869±.009	.691±.020	.801±.011
spell	tfidf	500	[1, 1]	.779±.020	.863±.014	.678±.043	.797±.018

Table 5.3: Resulting F1-scores of the early fusion model on 5-fold cross-validation for parameter aggregates of different log parsers. The table is sorted by the F1 Macro score in descending order and presented as the mean score plus-minus the standard deviation.

Parser	F3 Macro	HW	NF	SW
drain	.737±.029	.829±.018	.641±.061	.742±.027
spell	.730±.023	.816±.010	.638±.043	.734±.031

already found in the word embeddings (in the case of this classification task). For this reason, we see no reason to include neither the representations of the events nor the parameters for the final model. In other words, we only use the word embeddings as chosen in Table 5.1.

Finally, we also visualize the top important features using their SHAP values, as seen in Figure 5.1. Here we can e.g. see that the model has found a positive correlation between the words `'error'`, `'disk'`, and `'hw'` in the `hwlog` and the hardware fault class. In addition, we also take the mean of the absolute SHAP values for the different log types to investigate which log type contributed the most to the prediction. These results can be seen in Figure 5.2 where we see that `llog` seem to be most important in determining the `NF` and `SW` class, while `hwlog` is most important for the `HW` class.

Table 5.4: Resulting F1-scores of the early fusion model on 5-fold cross-validation for combinations of the best feature representations found previously. The table is sorted by the F1 Macro score in descending order and presented as the mean score plus-minus the standard deviation.

Words	Events	Parameters	F1 Macro	HW	NF	SW
X	-	-	.910±.016	.955±.005	.855±.030	.919±.021
X	-	X	.904±.019	.952±.006	.844±.041	.915±.020
X	X	X	.903±.013	.952±.006	.846±.035	.911±.009
X	X	-	.902±.013	.953±.005	.842±.035	.911±.008
-	X	-	.826±.013	.901±.004	.735±.021	.842±.018
-	X	X	.818±.013	.901±.006	.716±.023	.839±.014
-	-	X	.729±.023	.825±.013	.635±.054	.728±.027

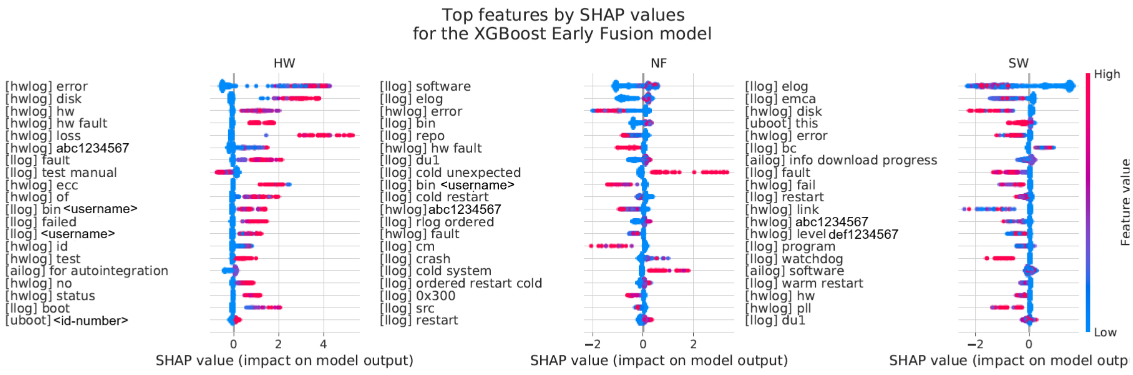


Figure 5.1: Top features by their corresponding SHAP value for the early fusion model. The horizontal position indicates whether the feature contributed to a positive prediction, and the color indicates the magnitude of the feature value.

5.1.2 Late fusion XGBoost model

Using the best feature representations for words, events, and parameters found in Section 5.1.1 we also investigate which combination of these work best for the late fusion model, as seen in Table 5.5. Once again, we see that the most useful feature is the word representations, with both the events and parameters offering little to help the classification performance. Again, we can use only the word representations for the final late fusion model.

5.2 Experimental Models

In this section, we present the results achieved using the more experimental LSTM and Transformer models. These results should be viewed in relation to the baseline results in Section 5.1 as these models are comparatively more complex and utilize features in the form of WAEMB which represent the logs on a per-line level.

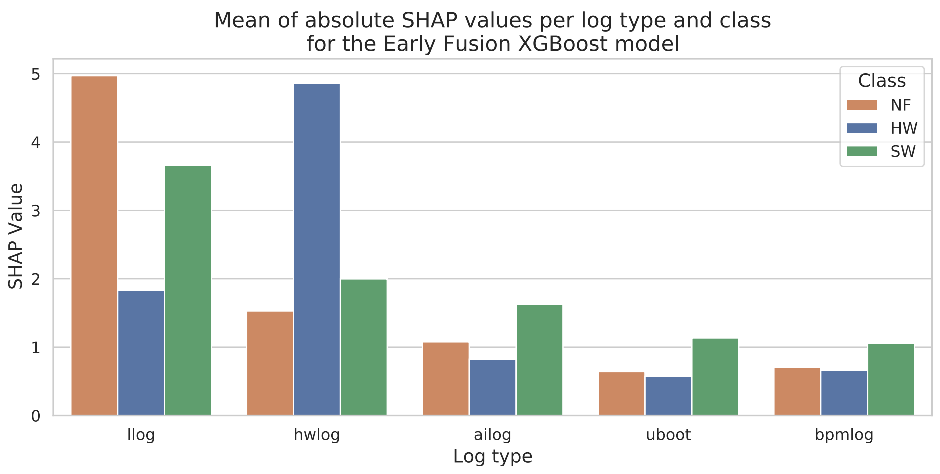


Figure 5.2: Visualization of the mean SHAP value per sample per log type of the early fusion XGBoost model. The values are also grouped by the true label for the corresponding samples.

Table 5.5: Resulting F1-scores of the late fusion model on the validation set for different word embedding methods. The table is sorted by the F1 Macro in descending order.

Words	Events	Parameters	F1 Macro	HW	NF	SW
X	-	-	.781±.020	.905±.014	.682±.035	.756±.035
X	X	-	.780±.022	.909±.011	.674±.046	.757±.042
X	X	X	.767±.036	.901±.012	.647±.079	.752±.057
X	-	X	.760±.028	.897±.016	.652±.037	.730±.051
-	X	-	.703±.022	.801±.015	.610±.039	.700±.033
-	X	X	.668±.017	.821±.008	.542±.030	.641±.028
-	-	X	.655±.024	.802±.017	.526±.054	.637±.042

5.2.1 Transformer

In Table 5.7 we evaluate the Transformer model validation performance for different sizes of the WAEMB Word2Vec embeddings and whether or not the positional encoding of the lines help the model, For each combination, the model was trained using stochastic gradient descent using the ADAM optimizer with 0.005 learning rate, 0.9 for `beta_1` and 0.999 for `beta_2`. The number of training epochs was limited by using early stopping with validation patience of 7 epochs, see Table 5.6 for model hyperparameters and training parameters.

Taking the standard deviation of the scores into account, it is evident that any of the dimensions between 25 and 100 seem to work for the model. In addition, we also see that the positional encoding might even make the model perform slightly worse. This implies that the model doesn't need to take the order of the lines into account when performing classification on the units. As the top scores are within the margin of each other for different embedding sizes, it appears to be sufficient to choose the

Table 5.6: Hyper parameters used in Transformer model

Hyperparameter	Value
Input size	[25, 50, 100]
Attention heads	3
Feed Forward dimension	50
Max sequence	512
Positional Encoding	[False, True]
Dropout rate	0.1
Learning rate	0.005
Early stopping	7

25-dimensional embedding without positional encoding for the final model

Table 5.7: Resulting F1-scores of the Transformer model over 100 bootstrap iterations with replacement on the validation data for different WAEMB embedding dimensions and positional encoding. The table is sorted by the F1 Macro score in descending order and presented as the mean score plus-minus the standard deviation.

Dimension	Positional	F1 Macro	HW	NF	SW
100	False	.889±.015	.934±.009	.820±.030	.912±.018
25	False	.887±.015	.930±.010	.819±.029	.913±.019
50	False	.881±.013	.928±.009	.812±.025	.903±.016
50	True	.879±.013	.924±.009	.812±.025	.901±.019
25	True	.874±.014	.917±.010	.783±.030	.921±.016
100	True	.860±.015	.911±.011	.778±.028	.892±.019

In Figures 5.3 and 5.4 we can see the training history for the chosen Transformer model. We can see that after around 10 epochs there is no further improvement in the model's validation classification performance. For information on the training histories of the other tested models, see Appendix A.1.

5.2.2 LSTM

In Table 5.9 we evaluate the LSTM model validation performance for different sizes of the WAEMB Word2Vec embeddings, For each combination, the model was trained using stochastic gradient descent using the ADAM optimizer with 0.005 learning rate, 0.9 for `beta_1` and 0.999 for `beta_2`. The number of training epochs was limited by using early stopping with validation patience of 7 epochs, see Table 5.8 for model hyperparameters and training parameters.

Taking the standard deviation of the scores into account, it is evident that any of the dimensions between 25 and 100 seem to work for the model. As the top scores are within the margin of each other for different embedding sizes, it appears to be sufficient to choose the 25-dimensional embedding for the final model

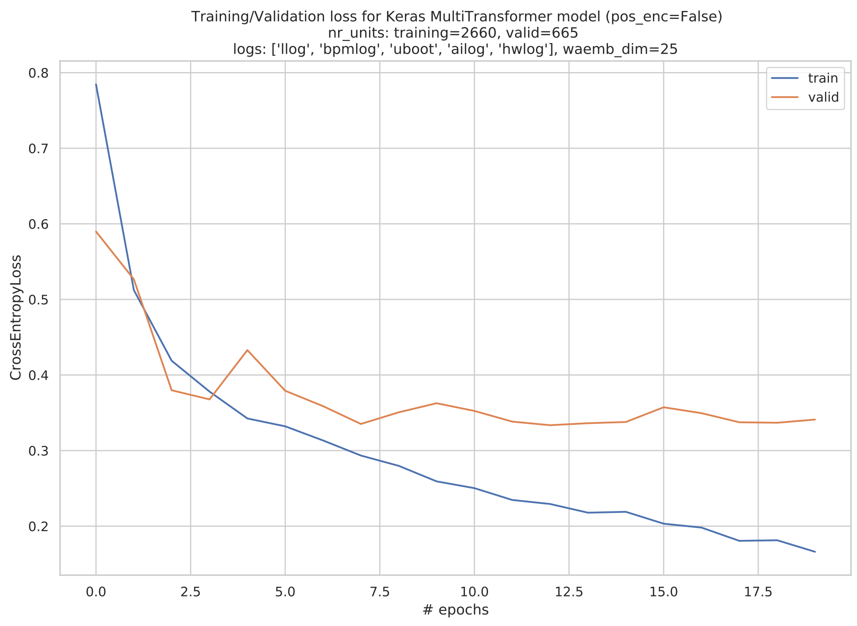


Figure 5.3: Training and validation loss for Transformer without positional encoding and WAEMB dimension 25

Table 5.8: Hyperparameters used in LSTM model

Hyperparameter	Value
Input size	[25, 50, 100]
Hidden size	50
Max sequence	3000
Recurrent layers	2
Bidirectional	True
Learning rate	0.005
Early stopping	7

In Figures 5.5 and 5.6 we can see the training history for the chosen LSTM model. We can see that after around 10 epochs there is no further improvement in the model's validation classification performance. For information on the training histories of the other tested models, see Appendix A.2.

5.3 Comparison

In Sections 5.1 and 5.2 we investigated and selected the best candidate models through evaluation on validation data. We can now compare these models against the existing Ericsson model on the test data. These results are presented in Table 5.10. In addition to the previously presented models in this chapter, we now also have the Ericsson model which is the same one currently in use in production. We also

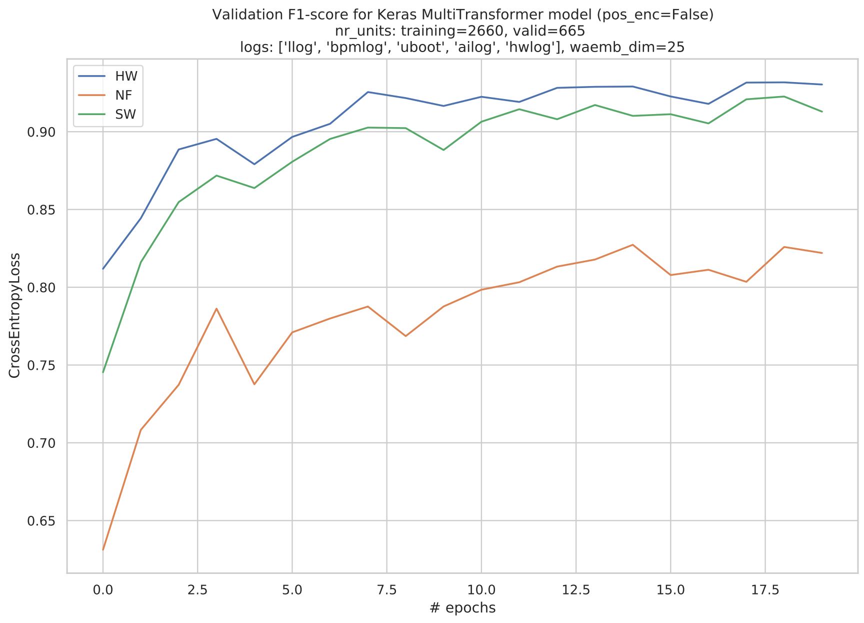


Figure 5.4: F1-score on validation data for Transformer without positional encoding and WAEMB dimension 25

Table 5.9: Resulting F1-scores of the LSTM model over 100 bootstrap iterations with replacement on the validation data for different WAEMB embedding dimensions. The table is sorted by the F1 Macro score in descending order and presented as the mean score plus-minus the standard deviation.

Dimension	F1 Macro	HW	NF	SW
25	.709±.019	.785±.018	.612±.033	.729±.025
100	.704±.019	.775±.017	.599±.038	.740±.024
50	.673±.020	.728±.021	.586±.038	.705±.025

include a stratified baseline model which represents the base scores if we were to guess randomly on labels with regard to their total proportion.

From these results, we can see that both the early fusion XGBoost model and the transformer model can achieve parity with the current production model by Ericsson. However, both the LSTM and late fusion XGBoost model struggle. We, once again, note that the standard deviations of the top-performing models are so large that we can't really say which one (if any) of them is the best with regards to these metrics.

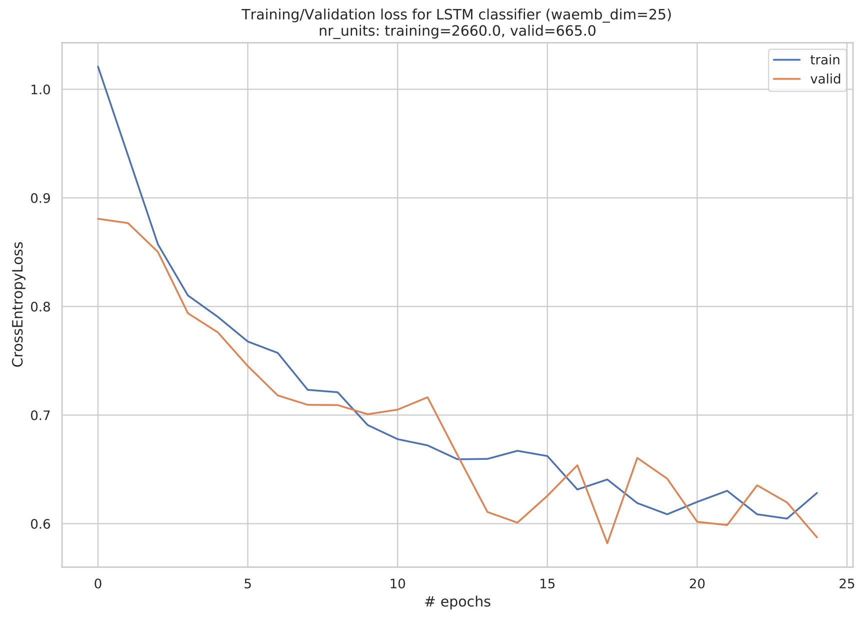


Figure 5.5: Training and validation loss for LSTM model with WAEMB dimension 25

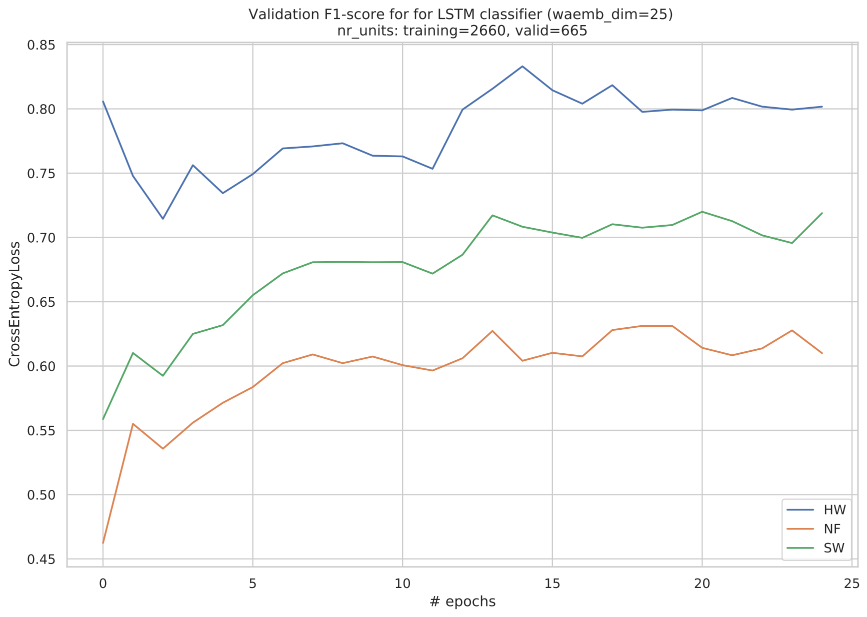


Figure 5.6: F1-score on validation data for LSTM model with WAEMB dimension 25

Table 5.10: Resulting F1-scores of the best models over 100 bootstrap iterations with replacement on the test data. The table is sorted by the F1 Macro score in descending order and presented as the mean score plus-minus the standard deviation.

Best Model	F1 Macro	HW	NF	SW
XGBoost Early Fusion	.885±.014	.946±.009	.812±.029	.897±.016
Ericsson Model	.866±.015	.937±.009	.781±.033	.878±.017
Transformer	.861±.012	.934±.009	.785±.025	.865±.017
LSTM	.686±.018	.787±.018	.563±.033	.708±.025
XGBoost Late Fusion	.654±.023	.829±.013	.486±.049	.646±.032
Stratified baseline	.325±.019	.560±.023	.186±.033	.227±.032

6
Discussion

From the final model comparison results presented in Table 5.10, the XGBoost model with early fusion had the best performance according to the test evaluation. Both in regards to F1 macro and the individual classes. The performance of our early fusion solution compared to Ericsson is within the bootstrapped variance on the test dataset, with ours being on the upper end for all metrics. The Transformer model is also performing within variance compared to the Ericsson model.

The performance starts to clearly degrade with the LSTM and XGBoost late fusion models. Their F1 macro and F1 for each class are lower and outside of the variance than the top three models. Between the two, the performance is comparable with the LSTM model having higher scores on all metrics besides F1-score for `HW`.

With the performance of our XGBoost early fusion model being better or as good as the Ericsson model we have to start to compare their workflows to distinguish them. As presented in Section 1.3 the Ericsson model utilizes a manual rule-based data-mining feature extraction for the logs. Subject Matter Experts (SME) informs data scientists of subsets and features in the logs that should be informative of the state of the unit. This method requires people with domain knowledge to understand what should be extracted from the logs. With large expanding companies, having this domain knowledge timely available within multiple teams is difficult to achieve. This is where NLP techniques are very useful. Instead of transforming manually determined subsets of the logs into numerical representations, NLP techniques have the ability to automatically transform almost the entirety of the logs for any given log type. This results in being able to present the model with more information from the logs, while also giving the model itself the responsibility to find informative parts of the data. This generalized and scalable approach makes it easier to work with new types of data where domain knowledge is lacking. The only requirement is of course the access to labeled datasets, which will require some SME understanding of the issue in the labeling stage.

Besides not relying as much on domain knowledge, using NLP techniques might also result in a better understanding of the data. Depending on the type of prediction model used, its interpretability could be used to gather new insights into the data. For example, BOW and TFIDF representations of the data are easy to interpret with an ensemble model such as the XGBoost. The effect the individual words have on the ensemble model's output can be retraced back to the actual locations in the log file with the use of SHAP values (Section 3.4.2), see Figure 5.1. Further, words or log lines that previously have been seen as non-informative or unnecessary might be viewed as key features by the prediction model. This could start a discussion between data scientists and SMEs to better understand why the model interprets the data this way, perhaps leading to better insights about the data.

While the Transformer model also achieves performance similar to the XGBoost model it has some drawbacks in comparison. The XGBoost utilizes BOW representations which make for both fast feature extraction and training of the model. The Transformer model is a neural model and utilizes the feature representation WAEMB. Additionally, to compute the Wameb representation, another neural model like Word2Vec or FastText needs to be trained on the data. This results in a much longer training time compared to the XGBoost model. Another drawback the Transformer model has compared to the XGBoost model is the level of interpretability. While the XGBoost model can detect the effect of individual words in the logs, the feature representation WAEMB used in the Transformer is an embedding of an entire line. This way we can only consider the effect entire lines have on the model output. Even the key mechanism of attention utilized by the Transformer does not seem to offer any advantage over the XGBboost model. In theory, attention can be utilized to find relationships between lines in the logs. We have not conducted a deeper quantitative analysis of the behavior of the attention in the model. However, from the results, we have not seen any indication of that making a difference on this data compared to the XGBoost model and its BOW representation. With these three things in mind, the Transformer model achieves similar performance to the XGBoost model but with these additional drawbacks.

The LSTM model performed significantly worse compared to the Transformer model, even though they both used the same feature representation of the data (WAEMB) and shared similar model architectures. Both of the models use their Transformer or LSTM models to produce a hidden state of each log type to be concatenated and given to a prediction layer. The Transformer model has some additional layers such as dropout layers to help with regularization. One key difference is also that the Transformer model has a much lower max sequence length of 512 compared to the LSTM's 3000. If the results were reversed you could argue that giving the LSTM more lines from logs increases the amount of information presented, therefore increasing performance. However, this is not the case here as the Transformer outperforms the LSTM with less data. This point towards that there is enough information about the state of the unit in the last 512 lines, and that the hidden states produced by the Transformer are more representative of the data than the LSTM's. The Transformer is also able to retrieve information from any of 512 lines without any losses, making the model more robust compared to LSTMs which suffer from the vanishing gradient problem.

From our results, we have not seen convincing proof that the sequence of the lines gives any substantial information about the state of the unit. In Table 5.7 both Transformers with and without positional encoding have been trained and tested on the same data. From those results having positional encoding does not increase the performance of the model, rather the opposite. We do however do see some advantage of incorporating sequential information in the features in Table 5.2. The XGBoost model using EventIds produced by Spell or Drain is performing better if the EventIds can be expressed in N-grams of range 1-3 instead of only 1. Keep in mind that these results are restricted by how good Spell and Drain are on the data. Using the same XGBoost model we achieve higher results anyway by only using a BOW representation on the tokens in the logs, see Table 5.1. In the end, the LSTM does not perform as well as other models that do not have the sequential property.

Two different types of fusion of data for the XGBoost model have been investigated: early and late fusion. In Table 5.10 we clearly see that the early fusion model performs significantly better than the late fusion version. One apparent drawback of the late fusion model is that it can not utilize the relationship of features between the log types in its prediction. Instead, it makes a prediction for each of the log types and then makes the final prediction based on the sum of the prediction probabilities. If certain log types do not contain useful information for some labels their ”vote” will simply confuse the model instead of assisting it. Implementing a more sophisticated prediction rule is time-consuming, and taking into account the great performance achieved with a simple early fusion method, it’s not worth the effort.

In Figure 5.1 we also see an example of how easy it is to interpret the BOW features and their importance of the XGBoost model using SHAP values. At a glance, it is even quite understandable even without major domain knowledge of the logs that the features are quite reasonable. E.g., we see words like `'hw'` and `'error'` in the `HW` class features, and `'software'` and `'program'` in the `SW` class features. In Figure 5.2 we also see that it is mainly the `llog` and `hwlog` log types that are driving the predictions of the model. While not presented in this report, initial testing of only utilizing these log types did indeed yield similar performance as using all log files. Through this data-driven approach. it is then also possible to find that not all log files suggested by the SMEs were in fact important for the classification.

Both the Transformer model and the LSTM utilizes the WAEMB feature. In Tables 5.7 and 5.9 there are results from models that use different sizes of the WAEMB feature. The different feature dimensions are the cause of different embedding sizes from the Word2Vec embedding models. The higher feature dimension results in being able to store more information about the embedded word, or in this case the embedded log entry. For both the Transformer and the LSTM it seems that neither increasing or decreasing the feature dimension affects the models’ performances. For the Transformer, all the results without positional encoding are within the variance of each other. This also applies to the LSTM model. From our results, it did not matter if we chose 25, 50, or 100 as the embedding dimension, the models performed similarly with most configurations. One explanation of this could be the low amount of unique tokens present in the lines of the log files, which can easily be embedded in the relatively low-dimensional vector space.

6.1 Anomaly Detection and Fault Classification

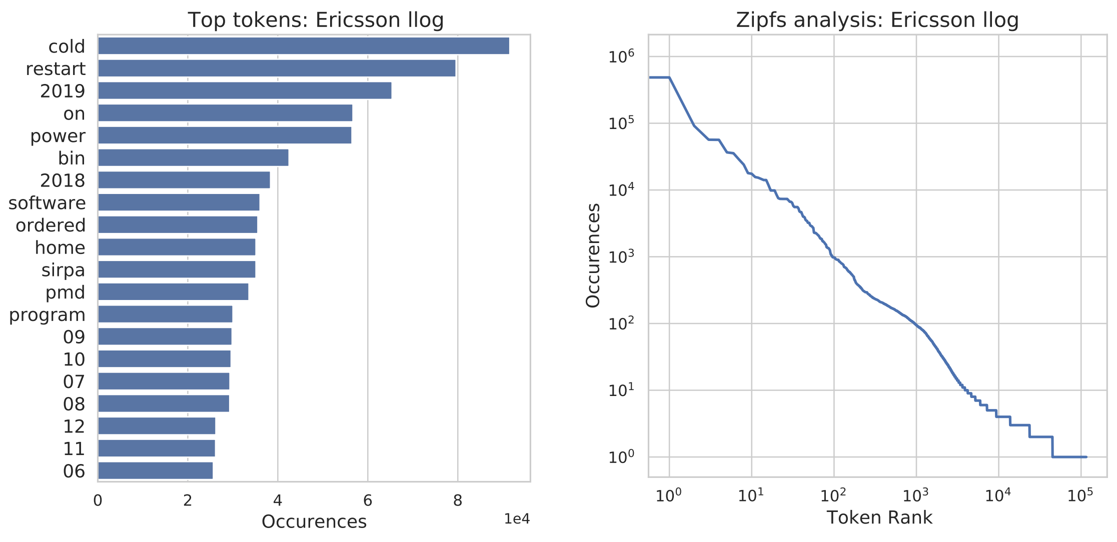
One remark we made in Section 2.3 is that the area of log anomaly detection seems to be more researched compared to that of log classification. Therefore, we instead looked into the NLP area of document classification. However with all the information available on log anomaly detection, one could ask if those findings could be utilized for the classification of logs. In most of the studied papers, the main focus of anomaly detection was to identify **when** an anomaly is occurring. Often in the context of this regression analysis, sequential models such as LSTM have been used [6], [16]. As an anomaly can be represented in a single log entry or short sequences, regression or sequence models that can process the data sequentially can capture these local anomalies. From our results, we have not seen any indication that the sequence of the log lines affects the fault label for the baseband unit. This causes many of the current methods for anomaly detection to not be suitable for our thesis. From what we have observed the state (fault label) of the log, or the entire unit, might not be determined by a sequence of lines or where a line occurs in the log. Instead, simply the number of occurrences of certain lines or words might be what is needed to correctly classify the log. This type of information can be captured in log-level feature representations such as BOW. Despite not being able to capture anomalous sequences, models like Random Forests seem to offer sufficient predictive capabilities for this type of data.

6.2 NLP in Log Analysis

Most of the recently published NLP techniques, such as attention, that have shown promising results are usually on textual data written by a person. The tasks are usually translation tasks or classifying if a sentence or document is positive or negative. The structure of the textual data could vary heavily depending on what type of language the text is written in. One could then ask how different the textual data in software and hardware logs are compared to written human language such as English or French. To better understand if commonly used NLP techniques can be effectively utilized on data logs we conduct a simple analysis to find the similarities and dissimilarities between written language and computer-generated logs. This analysis was conducted on approximately 4000 logs of the log types `llog` and `hwlog` from Ericsson. To compare the log data against human written text we chose the IMDb (Internet Movie Database) movie review dataset [25], containing 50000 written reviews with binary labels (positive or negative).

One simple qualitative indication of a corpus is to see how well it follows Zipf's Law. By Zipf's law, if you plot the occurrences of a word against its rank in a large corpus, you should get something that looks like the power-law distribution. For a plot with logarithmic scales on the axis that would result in a line.

Looking at `llog` in Figures 6.1a and 6.1b we can see that the bar plot follows a power-law distribution and results in a linear line in the log-log plot. This is also found in Figures 6.2a and 6.2b for the IMDb dataset which was expected.



(a) Bar plot of most occurring tokens in `llog`

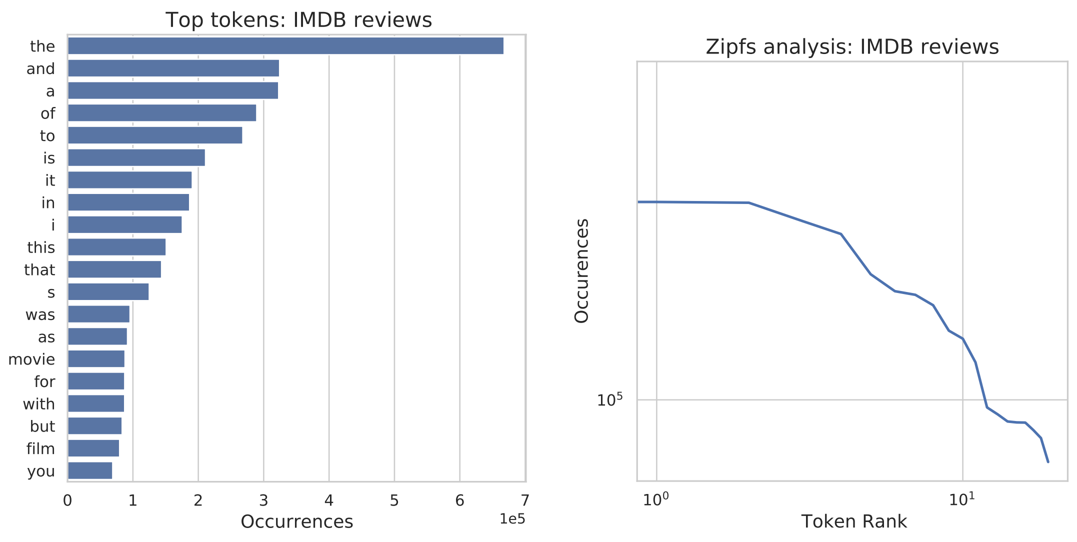
(b) Log-Log plot of occurrences of tokens in `llog` relative to their token rank

Figure 6.1: Analysis of Ericsson token occurrences in the Ericsson `llog`

Looking at Figures 6.3a and 6.3b however, we do not see the same behavior in `hwlog`. Many high-ranked tokens share a similar number of occurrences which skews a part of the distribution. By looking at the bigrams and trigrams of `hwlog` in Figures 6.4a and 6.4b we understand why this occurs. Multiple tokens seem to almost only occur together with each other, such as `power` and `on`. Since these words only occur together you could view them as a single "word" instead of two, a pattern not found in the IMDb dataset. This indicates that some highly occurring words are never used in different contexts.

Besides looking at the occurrences of tokens we can also investigate how the sequences of words are structured in sentences and log lines. The log parsers Spell and Drain have the functionality to group log lines by a similarity measure to extract features from standardized templates, see Figure 4.2. By applying Spell to these three datasets we can estimate how diverse the construction of sentences is by computing the number of unique EventIds for an increasing number of documents. If certain words only appear before or after another word you could have low diversity in terms of sentences and how words are used. This could result in words that always have the same context. If the words in the corpus are not utilized in different contexts and kinds of sentences you could argue that there is no need for advanced models that can learn to understand this contextualized information.

In Figure 6.5 we see how the number of unique EventIds for the datasets increases with the number of documents. For both `llog` and `hwlog`, we see that the number of unique EventIds barely increases after the initial 100 documents. Indicating that Spell within these 100 documents has already seen all possible sentence templates. For IMDb however, we see a steady increase of new EventIds relative to the number of documents.



(a) Bar plot of most occurring tokens in IMDB dataset

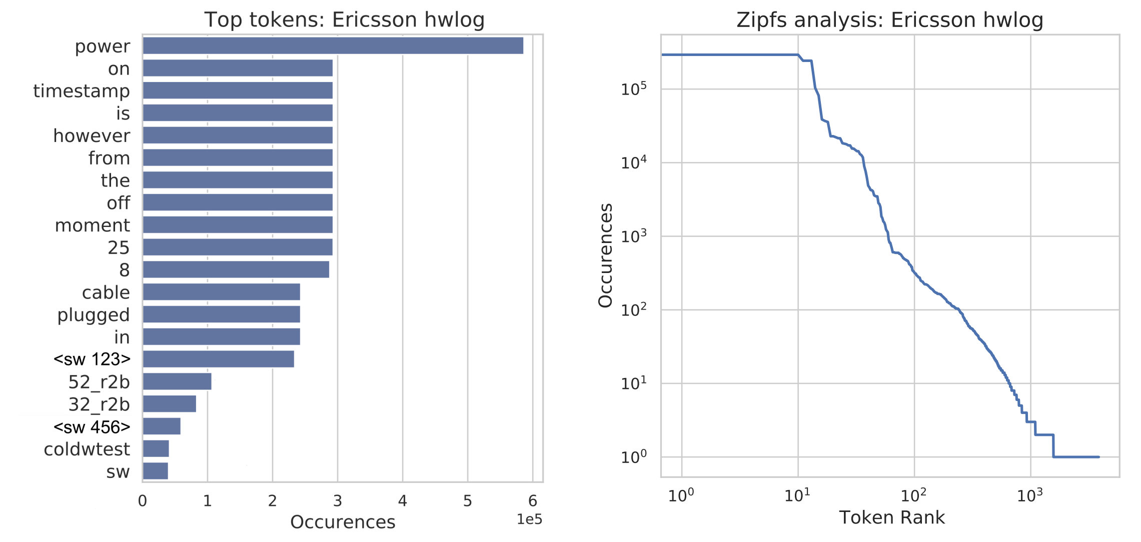
(b) Log-Log plot of occurrences of tokens in IMDB dataset relative to their token rank

Figure 6.2: Analysis of IMDb movie reviews

With this analysis, we can't prove that there is a major distinctive difference between written textual data (IMDb movie reviews) and the Ericsson logs. What we can see is some indications of how the variety of sentences and combinations of words are different in the Ericsson logs compared to IMDb's movie reviews.

Most of the progress made in the NLP area in the last years has been because of large language models inspired by the Transformer such as BERT (Bidirectional Encoder Representations from Transformers) [9] and GPT-3 [10]. There is no doubt that these models achieve state-of-the-art performance on translation tasks as well as text classification. However, all these tasks share one common trait, the type of data. The datasets used for these tasks are often text or documents written by people, and not computer-generated like logs. If there is a large enough difference between "normal" language and computer logs in terms of how words and sentences are constructed, these large language models might be unnecessarily complex for log classification. From the analysis, we can show that for `hwlog` and `llog` the number of unique sentences (log entries) does not increase as much as the IMDb dataset relative to the number of documents, see Figure 6.5. We also observe a unique behavior for `hwlog` where multiple high-occurring words only appear in the same context throughout the entire corpus. This results in words that do not change semantic meaning depending on context. This takes away one major property of the language models: understanding and learning words and sentences in different contexts. If that property is not useful there are even fewer reasons to choose a complex language model over a simple ensemble model.

From the results presented in Table 5.10 the attention-based model performs almost as well as the XGBoost model. However, that is with the feature representation



(a) Bar plot of most occurring tokens in `hwlog`. Some tokens are obfuscated.

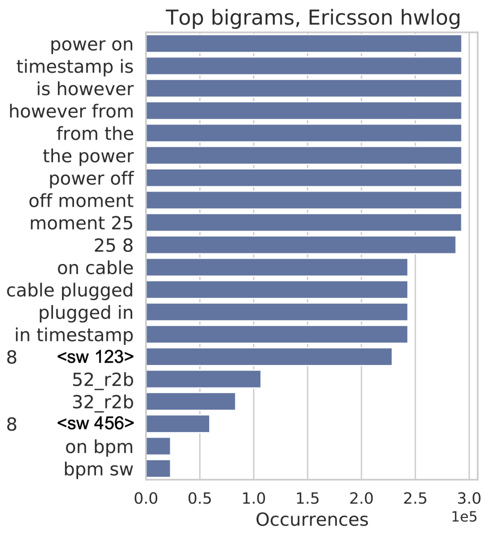
(b) Log-Log plot of occurrences of tokens in `hwlog` relative to their token rank

Figure 6.3: Analysis of the token occurrences in the Ericsson `hwlog`

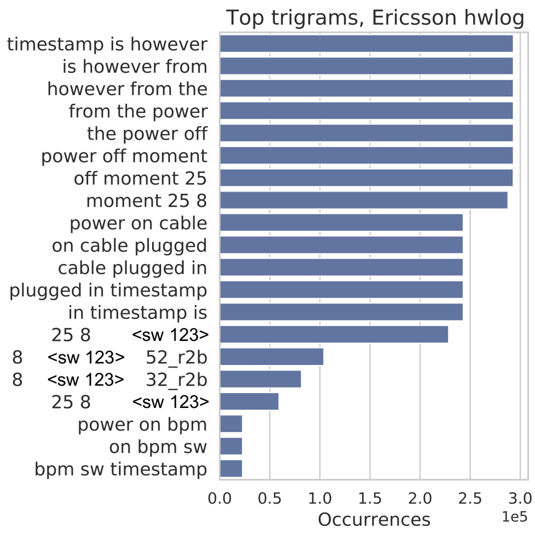
WAEMB which reduces the dimensionality of the log files by using sequences of line-embeddings instead of sequences of tokens. One issue a BERT model could have is the tokenization. A pre-trained BERT model also has a pre-trained tokenizer. Therefore all the domain-specific words in the logs will be split into sub-words or even individual characters since these are never seen before words for the model. This could result in logs with tokens in the tens of thousands. As most BERT models only take a maximum sequence length of 512 to 2048, you'll have to implement a solution for this problem such as a sliding window. Due to time limitations and problems with software, a proper evaluation of a BERT model was not conducted. A simple DistilBERT [26] model was implemented on the log type `llog` with a sliding window method. Since the model is only able to train and predict on units that have `llog`, the results are not comparable with the other models in this thesis. Therefore it was chosen not to be a part of this thesis results. Figures of the training/validation loss as well F1-scores on the validation data for the DistilBERT model can be found in Appendix A.3.

6.3 Future Work

One type of model we did not have enough time to examine closely was a larger language model such as BERT. In Section 6.2 we bring up our concerns of using a language model on log data. However, we did not present any results to further prove our speculation besides a simple pre-trained DistilBERT implementation on the log type `llog`, see Appendix A.3. As this model was neither trained nor tested on the same dataset as the other models we could not draw any conclusions based on its results. With the immense success of language models in NLP-related tasks, one apparent way to extend this study is to include a thorough analysis of language



(a) Bar plot of most occurring bigrams in `hwlog`. Some tokens are obfuscated.



(b) Bar plot of most occurring trigrams in `hwlog`. Some tokens are obfuscated.

Figure 6.4: Bigrams and trigrams of Ericsson `hwlog`

models on multimodal log classification, either through a pre-trained model or entirely from scratch.

For the XGBoost model, SHAP values for the features were computed to analyze what features in the log files the model found informative. As shown in Figures 5.1 and 5.2 by the SHAP values we can identify individual words, n-grams, or entire log files to be more informative for certain labels than other features. A similar analysis for the Transformer model was not conducted, but since its performance was similar to the XGBoost model it would be interesting to further understand how the model uses attention for its classifications. Since the Transformer uses the WAEMB representation, the attention can only be used to get information on how lines in the log affect the model output compared to the XGBoost which has token-level representation. However, we could get some new insights about the data on line-level better understand the model's reasoning.

Another area of analysis that could be investigated more is the log parsers Spell and Drain. The information EventIds and parameter lists can provide are only as good as the parsers on the data. With poor configurations, the produced EventIds and parameters will not be informative or of use for many models. Due to the large dataset and long log files, the training of these parsers takes extensive time which limited the number of tested configurations. Instead, more time and resources were invested in other areas of the thesis. It would however be interesting to further examine the possibilities and performance of these parsers by using different configurations as well as incorporating them in new ways for log classification.

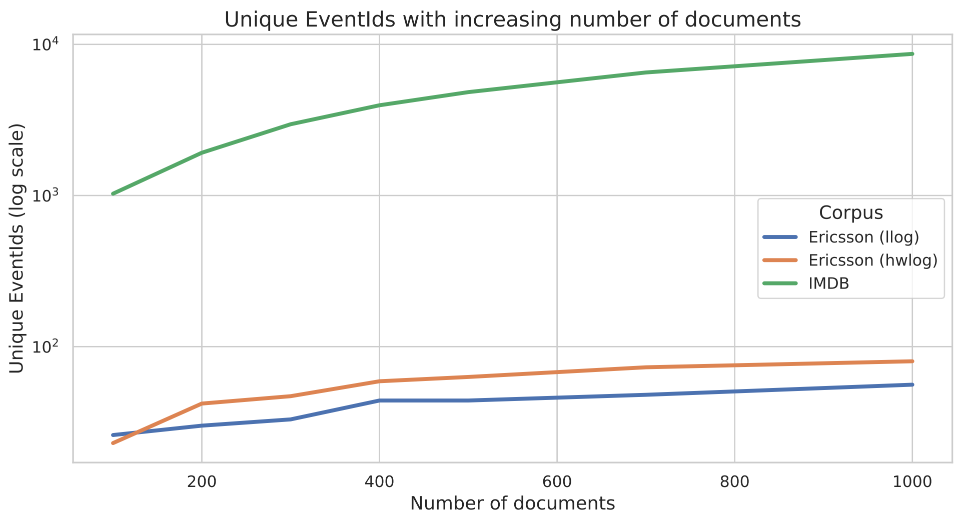


Figure 6.5: Number of unique EventIds (log scale) relative to the amount of documents

7 Conclusion

To classify the Ericsson baseband units using logs produced by their hardware- and software systems, we have shown that a data-driven NLP approach combined with an XGBoost classification model can match the current feature extraction and production model used by Ericsson. This was done by utilizing only the word-count BOW feature of each log type as concatenated input features to the XGBoost model. Attempts to incorporate feature representations of the sequence of the log entries, or parameter lists did not yield any improved results. In addition, while this word-count representation is relatively simple, it also provides interpretable predictions of the model when evaluated using SHAP values. These words can in turn also be used to find the actual location of specific lines in the original logs, to provide SMEs with practically insightful ways of interpreting the features and predictions. This data-driven approach to feature extraction also requires less domain knowledge compared to the existing manual data-mining feature-engineering in use at Ericsson. In practice this is then a more scalable method of generating features for ML, allowing for faster implementation of similar ML models for other products where logs are used for classification.

Furthermore, while larger deep learning models like Transformers combined with neural embeddings like Word2Vec can produce similar results, they are also comparatively complex in relation to the baseline solution. Our explanation of this is that the hardware- and system log data extracted from the baseband units do not show the same high variability in sentence structure, nor seem to depend on structures of sequences for different hardware- or software faults. We also claim that care should be taken when treating logs as texts found in other classical NLP tasks, like sentiment analysis, or document classification where the text is in fact directly generated by humans, as opposed to automatic logging systems.

References

[1] T. Hastie, R. Tibshirani, and J. Friedman, "The elements of statistical learning – data mining," in, second. 2013, pp. 587–602.

[2] S. M. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," in *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., Curran Associates, Inc., 2017, pp. 4765–4774. [Online]. Available: `http://papers.nips.cc/paper/7062-a-unified-approach-to-interpreting-model-predictions.pdf`.

[3] L. S. Shapley, *Notes on the n-Person Game II: The Value of an n-Person Game*, 1951.

[4] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16, San Francisco, California, USA: ACM, 2016, pp. 785–794, ISBN: 978-1-4503-4232-2. DOI: `10.1145/2939672.2939785`. [Online]. Available: `http://doi.acm.org/10.1145/2939672.2939785`.

[5] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," in *1st International Conference on Learning Representations, ICLR 2013 - Workshop Track Proceedings*, 2013.

[6] M. Du, F. Li, G. Zheng, and V. Srikumar, "DeepLog: Anomaly detection and diagnosis from system logs through deep learning," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2017. DOI: `10.1145/3133956.3134015`.

[7] 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 *IJCAI International Joint Conference on Artificial Intelligence*, vol. 2019-August, 2019. DOI: `10.24963/ijcai.2019/658`.

[8] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems*, vol. 2017-December, 2017.

[9] J. Devlin, M. W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *NAACL HLT 2019 - 2019 Conference of the North American Chapter of the Association for*

Computational Linguistics: Human Language Technologies - Proceedings of the Conference, vol. 1, 2019.

[10] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, *Language models are few-shot learners*, 2020.

[11] M. Du and F. Li, "Spell: Streaming parsing of system event logs," in *Proceedings - IEEE International Conference on Data Mining, ICDM*, 2017. DOI: `10.1109/ICDM.2016.160`.

[12] P. He, J. Zhu, Z. Zheng, and M. R. Lyu, "Drain: An Online Log Parsing Approach with Fixed Depth Tree," in *Proceedings - 2017 IEEE 24th International Conference on Web Services, ICWS 2017*, 2017. DOI: `10.1109/ICWS.2017.13`.

[13] S. Zhang, W. Meng, J. Bu, S. Yang, Y. Liu, D. Pei, J. Xu, Y. Chen, H. Dong, X. Qu, and L. Song, "Syslog processing for switch failure diagnosis and prediction in datacenter networks," in *2017 IEEE/ACM 25th International Symposium on Quality of Service, IWQoS 2017*, 2017. DOI: `10.1109/IWQoS.2017.7969130`.

[14] K. A. Nguyen, S. S. Im Walde, and N. T. Vu, "Integrating distributional lexical contrast into word embeddings for antonym-synonym distinction," in *54th Annual Meeting of the Association for Computational Linguistics, ACL 2016 - Short Papers*, 2016. DOI: `10.18653/v1/p16-2074`.

[15] M. Iyyer, V. Manjunatha, J. Boyd-Graber, and H. Daumé, "Deep unordered composition rivals syntactic methods for text classification," in *ACL-IJCNLP 2015 - 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing, Proceedings of the Conference*, vol. 1, 2015. DOI: `10.3115/v1/p15-1162`.

[16] N. Bosch, "Multimodal Fusion for System-Wide Anomaly Detection through Multiple Log Files," BA thesis, University of Groningen, Faculty of Science and Engineering, 2020. [Online]. Available: `https://fse.studenttheses.ub.rug.nl/22523/`.

[17] J. Zhu, S. He, J. Liu, P. He, Q. Xie, Z. Zheng, and M. R. Lyu, "Tools and Benchmarks for Automated Log Parsing," in *Proceedings - 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice, ICSE-SEIP 2019*, 2019. DOI: `10.1109/ICSE-SEIP.2019.00021`.

[18] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[19] J. Alammar, *The illustrated word2vec*, Accessed: 2021-02-03, 2019. [Online]. Available: `https://jalammar.github.io/illustrated-word2vec/`.

[20] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," *CoRR*, vol. abs/1603.02754, 2016. arXiv: `1603.02754`. [Online]. Available: `http://arxiv.org/abs/1603.02754`.

[21] C. Olah, “Understanding lstm networks,” *Colah’s Blog*, Aug. 27, 2015. [Online]. Available: `http://colah.github.io/posts/2015-08-Understanding-LSTMs/` (visited on 04/26/2021).

[22] S. Hochreiter and J. Schmidhuber, “Long Short-Term Memory,” *Neural Computation*, vol. 9, no. 8, 1997, ISSN: 08997667. DOI: `10.1162/neco.1997.9.8.1735`.

[23] T. Ichiishi, “6 - cooperative behavior and fairness,” in *Game Theory for Economic Analysis*, ser. Economic Theory, Econometrics, and Mathematical Economics, T. Ichiishi, Ed., San Diego: Academic Press, 1983, pp. 117–149, ISBN: 978-0-12-370180-0. DOI: `https://doi.org/10.1016/B978-0-12-370180-0.50011-1`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/B9780123701800500111`.

[24] S. M. Lundberg, G. G. Erion, and S.-I. Lee, “Consistent individualized feature attribution for tree ensembles,” *CoRR*, vol. abs/1802.03888, 2018.

[25] A. L. Maas, R. E. Daly, P. T. Pham, D. Huang, A. Y. Ng, and C. Potts, “Learning word vectors for sentiment analysis,” in *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, Portland, Oregon, USA: Association for Computational Linguistics, Jun. 2011, pp. 142–150. [Online]. Available: `http://www.aclweb.org/anthology/P11-1015`.

[26] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, *DistilBERT, a distilled version of BERT: Smaller, faster, cheaper and lighter*, 2019.

A

Appendix 1

A.1 Transformer Training History

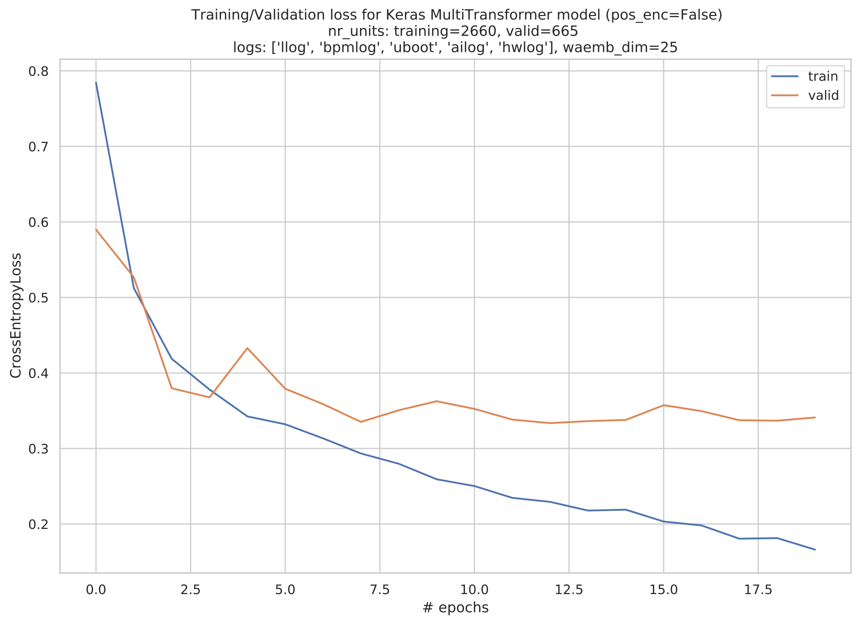


Figure A.1: Training and validation loss for MultiTransformer without positional encoding and WAEMB dimension 25

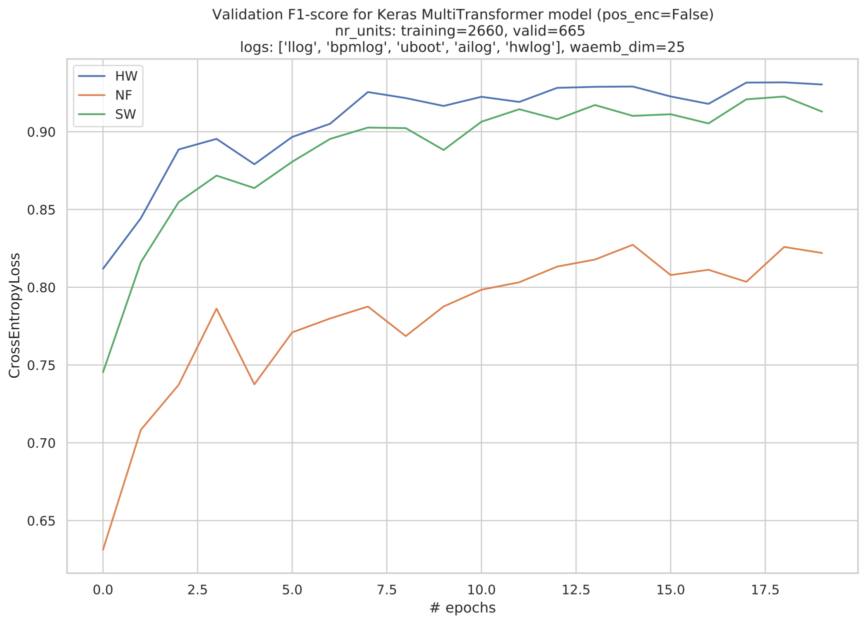


Figure A.2: F1-score on validation data for MultiTransformer without positional encoding and WAEMB dimension 25

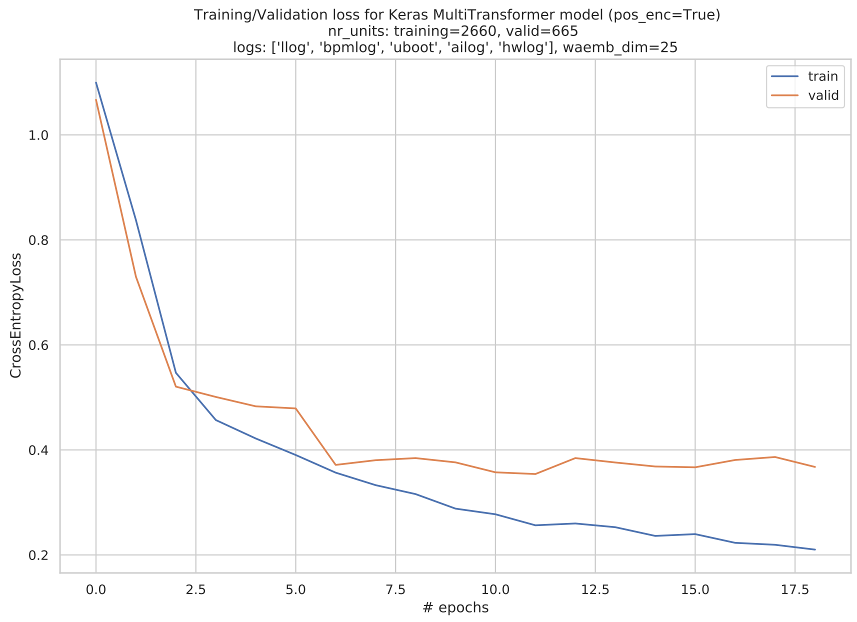


Figure A.3: Training and validation loss for MultiTransformer with positional encoding and WAEMB dimension 25

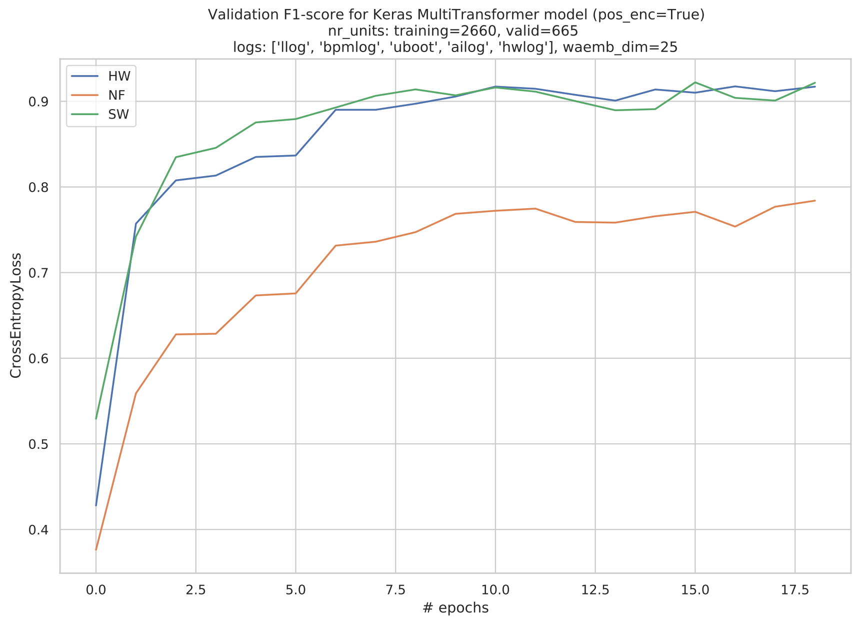


Figure A.4: F1-score on validation data for MultiTransformer with positional encoding and WAEMB dimension 25

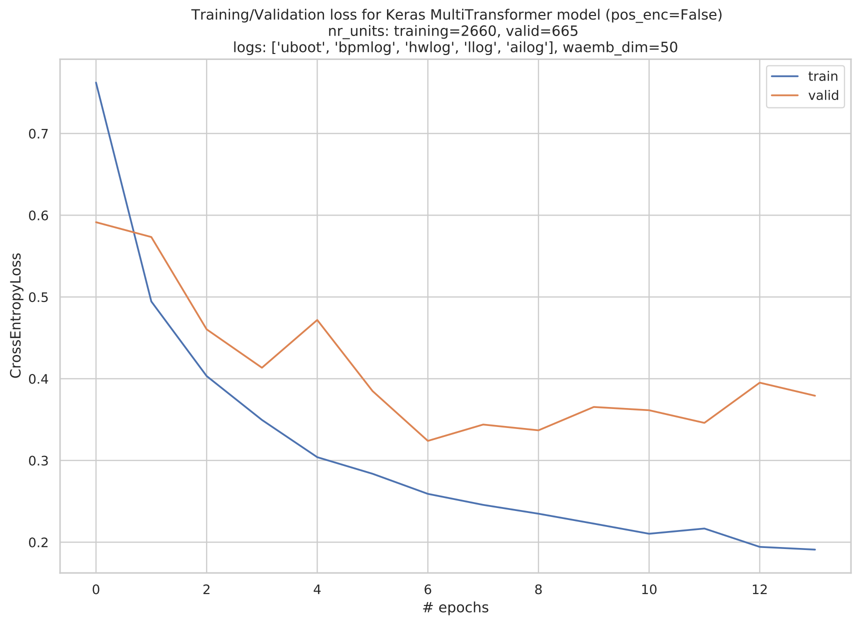


Figure A.5: Training and validation loss for MultiTransformer without positional encoding and WAEMB dimension 50

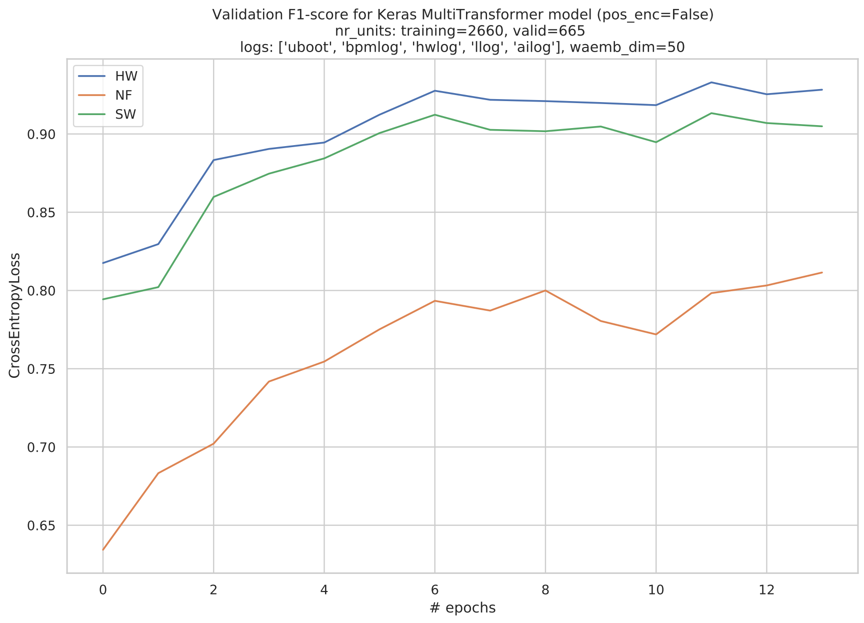


Figure A.6: F1-score on validation data for MultiTransformer without positional encoding and WAEMB dimension 50

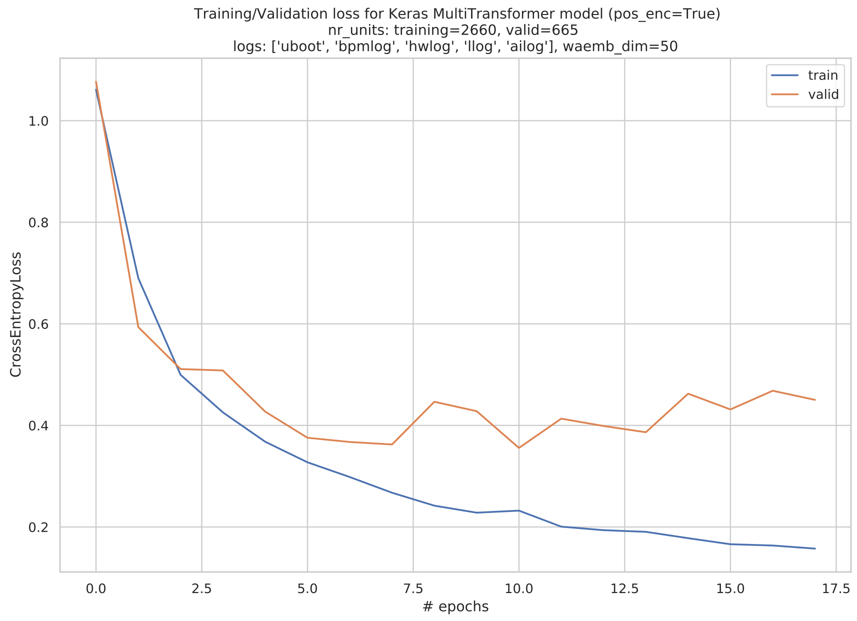


Figure A.7: Training and validation loss for MultiTransformer with positional encoding and WAEMB dimension 50

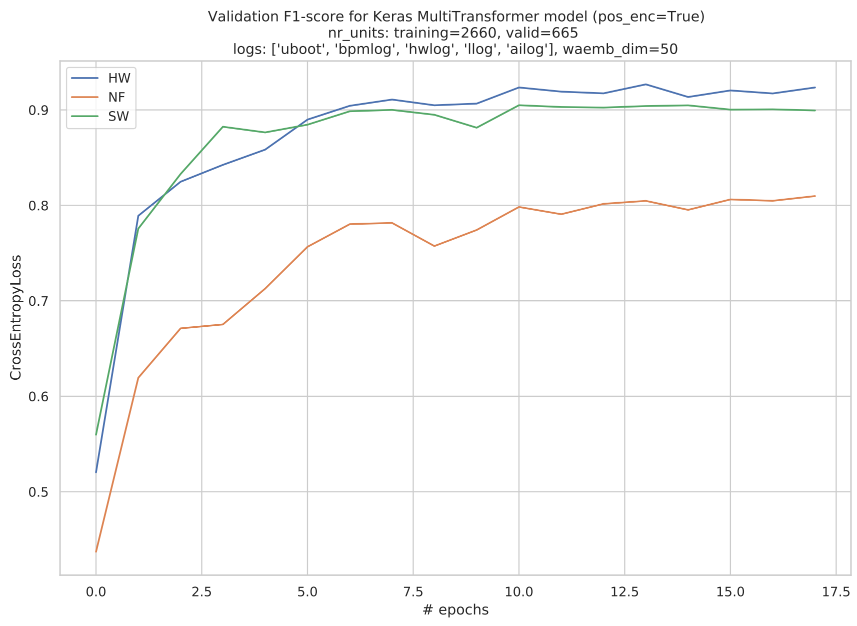


Figure A.8: F1-score on validation data for MultiTransformer with positional encoding and WAEMB dimension 50

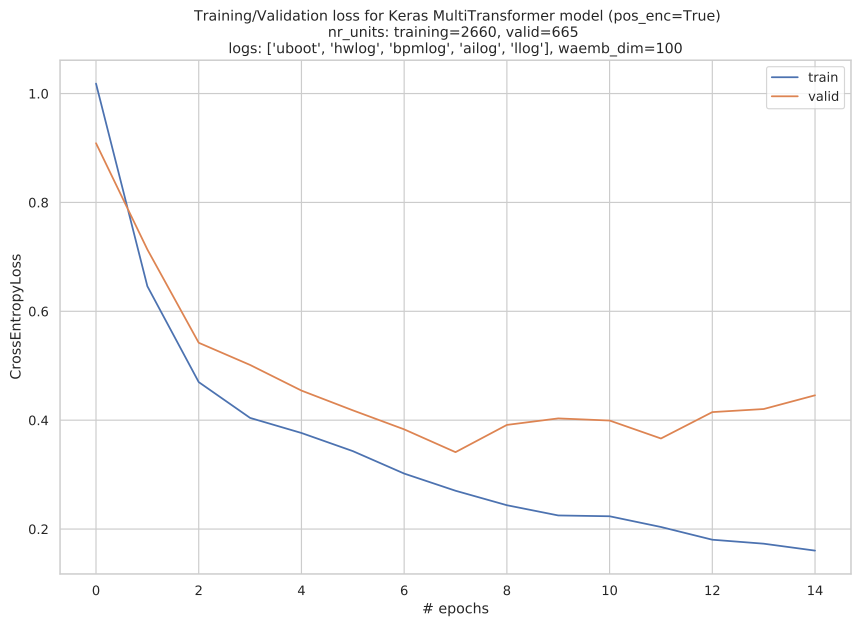


Figure A.9: Training and validation loss for MultiTransformer with positional encoding and WAEMB dimension 100

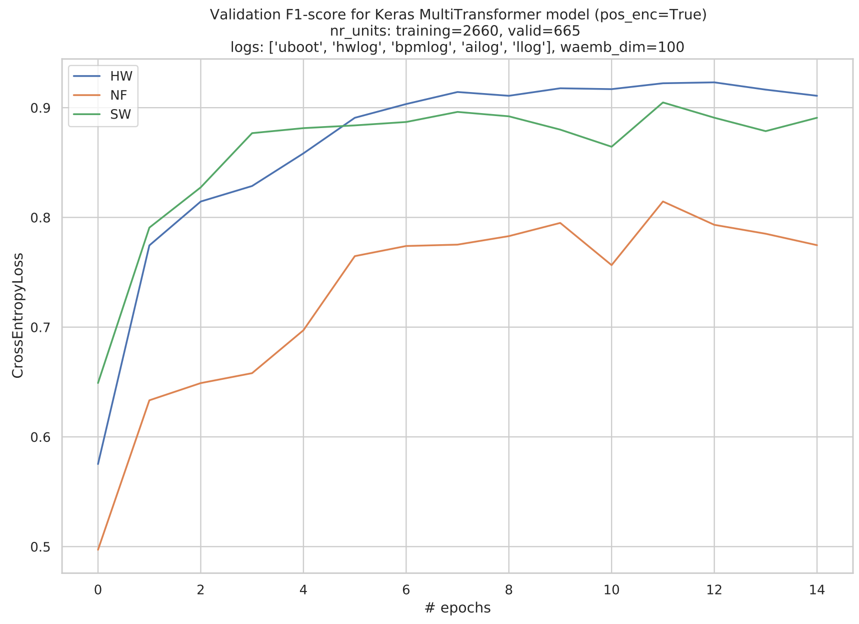


Figure A.10: F1-score on validation data for MultiTransformer with positional encoding and WAEMB dimension 100

A.2 LSTM Training History

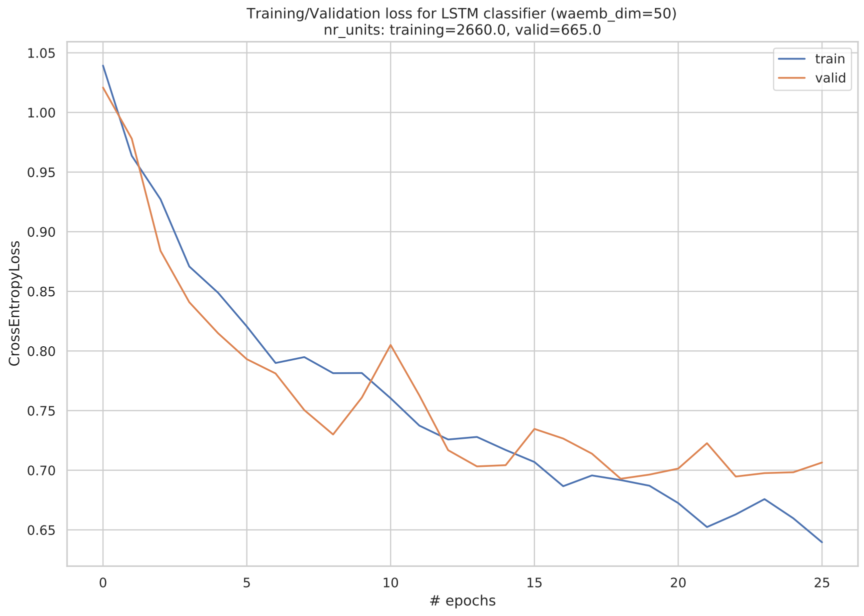


Figure A.11: Training and validation loss for LSTM model with WAEMB dimension 50

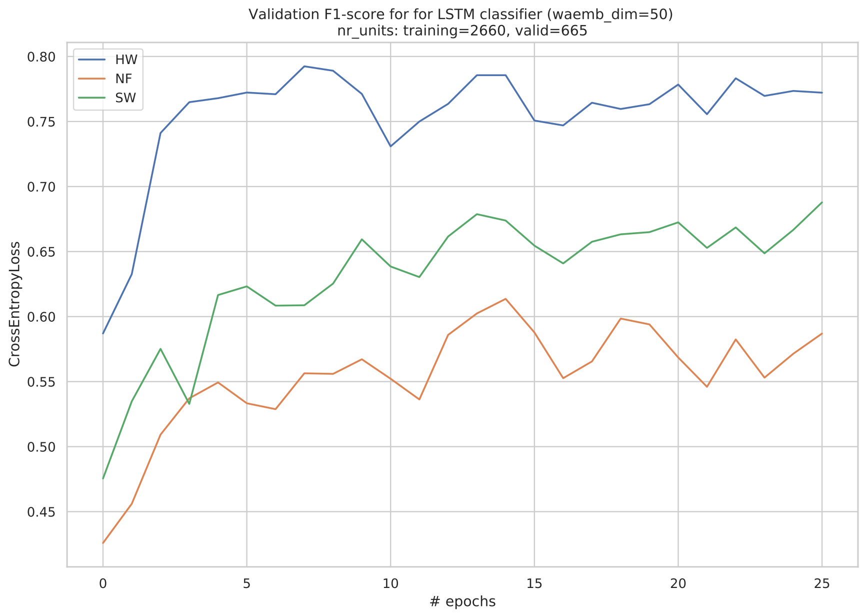


Figure A.12: F1-score on validation data for LSTM model with WAEMB dimension 50

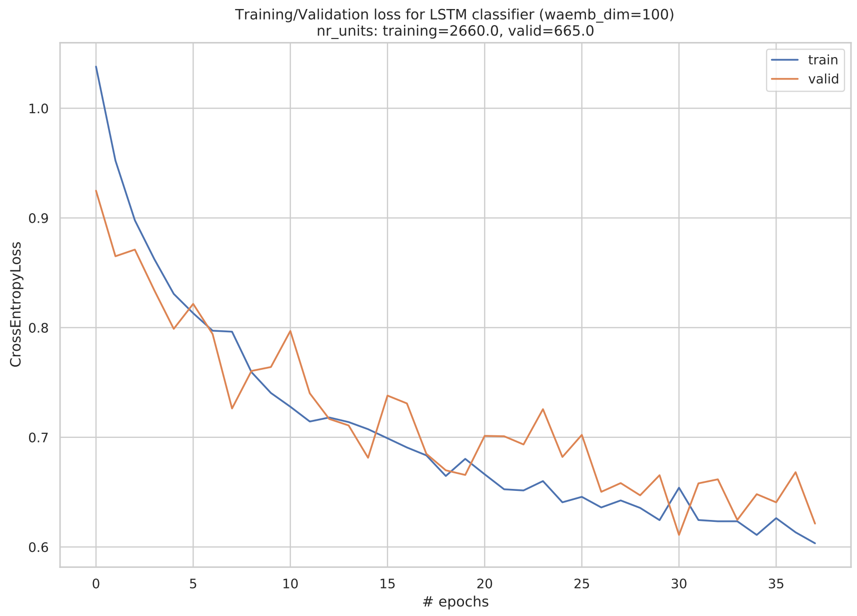


Figure A.13: Training and validation loss for LSTM model with WAEMB dimension 100

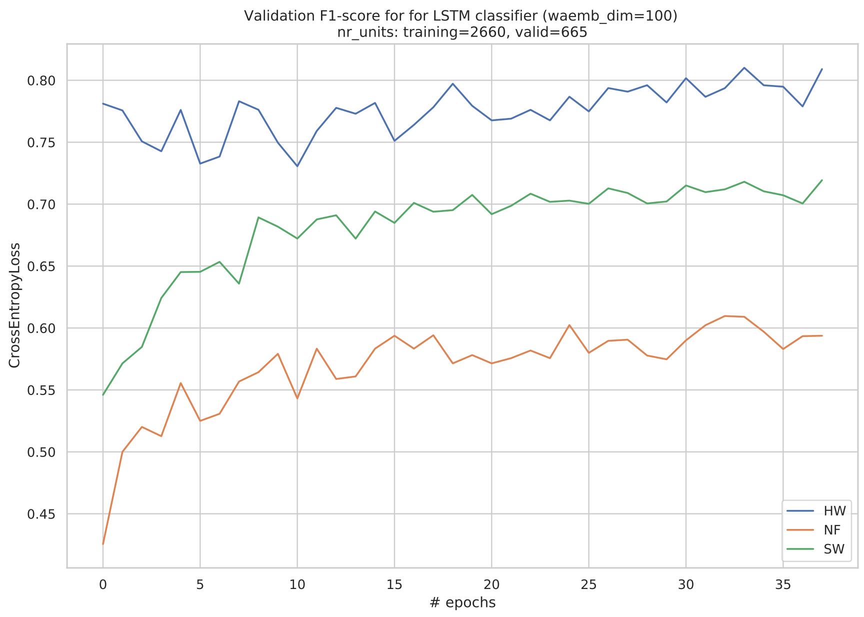


Figure A.14: F1-score on validation data for LSTM model with WAEMB dimension 100

A.3 DistilBERT Training History

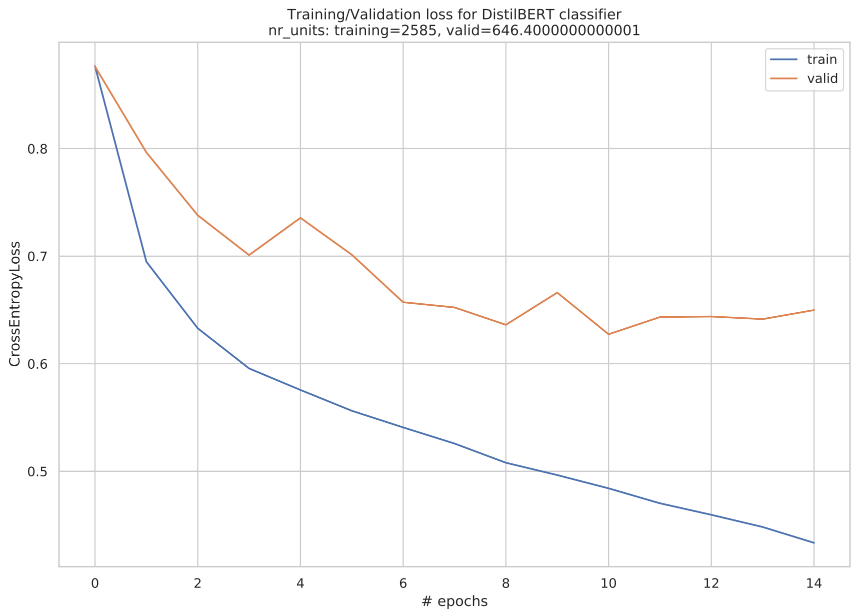


Figure A.15: Training and validation loss for DistilBERT model (only llog)

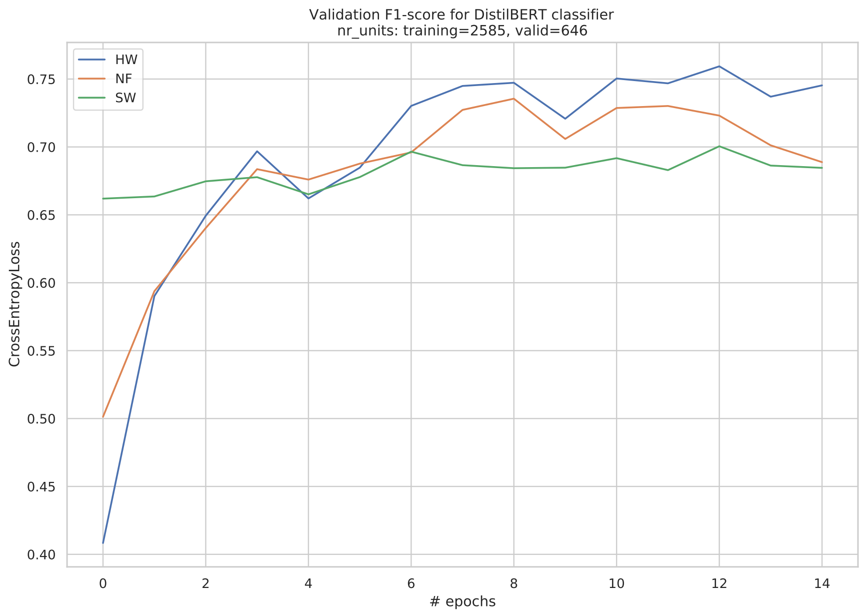


Figure A.16: F1-score on validation data for DistilBERT model (only llog)