MASTER'S THESIS 2024

Grouping of Duplicate Android Bug Reports through Log Anomaly Detection

Sofia Wahlmark

ISSN 1650-2884
LU-CS-EX: 2024-44

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2024-44

Grouping of Duplicate Android Bug Reports through Log Anomaly Detection

Gruppering av duplikat av Android buggrapporter genom detektering av onormala loggar

Sofia Wahlmark

Grouping of Duplicate Android Bug Reports through Log Anomaly Detection

Sofia Wahlmark
so0513wa-s@student.lu.se

June 28, 2024

Master's thesis work carried out at Volvo Car Corporation.

Supervisors: Qunying Song, qunying.song@cs.lth.se
Fredrik Henriksson, fredrik.henriksson@volvocars.com

Examiner: Emelie Engström, emelie.engstrom@cs.lth.se

Abstract

Bug solving is a critical aspect of modern software development, particularly within Android operating systems. Bug reports, containing extensive diagnostic information, are essential tools for analyzing and resolving issues in these systems. However, one significant challenge in bug resolution is the cost of identifying duplicate bug reports. Existing studies primarily focus on detecting duplicate reports based on e.g. title and description, potentially overlooking cases where bugs manifest differently visually or in log data. In this thesis, we utilize abnormal log sequences in the system logs to identify duplicate bug reports. This is done using the log parsing tool Drain [17] followed by the LSTM-based anomaly detector Deeplog [10] to extract abnormal logs which are used to measure the similarity between bug reports. The results demonstrate performance comparable to existing duplicate detection methods, with *RR@k*'s of 0.275, 0.471, and 0.588 for $k = 1, 3, 5$ respectively. These findings suggest the potential applicability of this method in industrial settings.

Keywords: Android bug report, log analysis, log parsing, anomaly detection, duplicate bug report detection, deep learning.

Acknowledgements

I would like to express my gratitude to everyone who has contributed to my research journey and offered their insights. Special thanks to my supervisor Qunying Song whose guidance and continuous feedback have been invaluable throughout the entire process.

I am also grateful to Volvo Cars for their support and interest in my research idea, providing me with the opportunity to explore and develop it further. I really appreciate all the help I have received from my supervisor Fredrik Henriksson and his support in navigating all company-related matters. I also appreciate the developers who were willing to participate in the interviews and share their thoughts in the preparation stage of this project.

Lastly, I would like to express my gratitude to my examiner Emelie Engström for their forthcoming evaluation and constructive feedback, which will undoubtedly enhance the quality of this work.

Contents

Chapter 1
Introduction

Bug solving is a central part of today's software development. For Android operating systems, a major tool available to analyze bugs and system behaviors is *bug reports*. These are usually captured after a bug has occurred and contain device logs, stack traces, and other diagnostic information that can be used to analyze and locate the cause of the behavior [2]. It is however common for Android bug reports to be very long and contain up to a million lines of logs and information, making more complex bug analysis a tedious process. The Android OS is a commonly occurring operating system for phones and tablets, and has in the later years also been developed for other areas such as cars. For this reason, it is of wide interest to make the process of bug resolution in this system more efficient.

One aspect of this is the handling of duplicate bug reports. At Volvo Cars bug reports are commonly attached to *tickets* describing the occurrence of a bug, these tickets are then assigned to software engineering teams so the bug can be solved. It is however not uncommon with duplicate tickets describing the same problem, and therefore the same bug just through a different report and logs. This is not always immediately discovered, especially if the description or attached visual representation of the bug is very different. This is an unnecessary waste of resources since it risks having two developers unknowingly working on solving the same bug as well as the time required to investigate the logs when the ticket is suspected to be a duplicate. It can also give a misleading picture of the backlog if multiple tickets are actually for the same bug. Finding duplicate tickets also contributes by grouping information about a bug from different tickets, as shown helpful in solving the bugs faster by Bettenburg et al. [6].

Existing studies on duplicate bug report detection mainly put the focus on the title and description [7][15][27][33][37]. While these approaches display efficiency, they risk missing the cases where a bug shows in visually diverging ways and therefore is described differently in the ticket. Studies have presented approaches where execution information is also included in the analysis with good results [35], making it interesting to further explore a more log-based duplicate detection approach. Additionally, detecting duplicate bug reports, and with that duplicate tickets, using logs would to some extent involve the detection of faulty log lines and log line sequences, as this needs to hold more weight to detect reports created for the same bug. There are several studies made in anomaly detection in system logs [10][19][24][39], mainly with the mutual aspect that they focus on abnormal log sequences.

The following subsections will give an introduction to how these aspects are applied. They will also present the research goals and how they are achieved, along with contributions to the field.

1.1 Research questions

At Volvo Cars it is not uncommon with duplicate bug reports and bug tickets and there is currently no internal system for automatically detecting and preventing duplicates. Previous research on the subject is mostly centered on text comparison of title, description, and other available parameters in the bug ticket [7][15][27][33][37], but this does not take account for incorrect system behaviors that are visually different but caused by the same underlying system error. The main goal of this thesis is therefore to contribute with a new approach by investigating how comparison could be done using the Android bug report logs. To structure the investigation of the problem, the following research questions were formulated:

- **RQ1**: Can log anomaly detection be used in duplicate detection between bug report logs in an industrial setting where the data is unlabeled?
- **RQ2**: How do the settings of an LSTM-based log anomaly detection model influence bug report duplicate detection?
- **RQ3**: What could the application of duplicate bug report detection look like in an industrial setting?

1.2 Research method

To answer the research questions, the Design Research Method (DRM) [31] is applied as seen in figure 1.1. Initially, the purpose and goals are formulated, which are presented through the research questions in section 1.1. A literature review and interviews are then performed to create a deeper understanding of the current situation. Based on this, a solution design is constructed. In this research, the proposed solution design entails duplicate detection done through comparison of abnormal logs given by log anomaly detection. A more extensive description of this can be found in section 2. Lastly, the results from the proposed solution are evaluated.

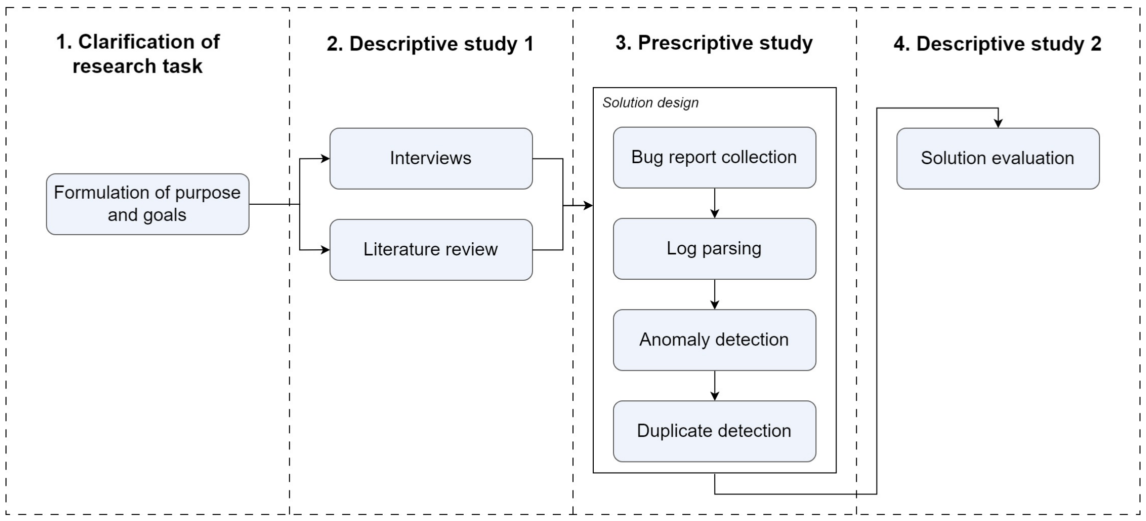


Figure 1.1: Application of the Design Research Method [31].

1.3 Result

The results shows that our approach is on the level of existing duplicate detection methods, with $RR@k$'s of 0.275, 0.471 and 0.588 for $k = 1, 3, 5$. Thus, the results show sufficient performance for this method to potentially be applied in an industrial setting.

It was shown that the hyperparameters used in the anomaly detection affect the final result of the duplicate detection and therefore need to be tuned based on the use case. Larger window size g showed a significant increase in precision, while top g probability had a smaller impact on the evaluation metrics. However, the combination of values for different hyperparameters matters and should be tuned in regard to each other for optimal performance in a given use case.

1.4 Contributions

The contributions of this research can be summarized as follows:

1. We propose a novel approach for duplicate detection by applying log anomaly detection to utilize abnormal logs in the duplicate predictions process.
2. The achieved results are in line with existing duplicate approaches, showing that the performance of our approach is comparable to the state-of-the-art.
3. We present how our approach can be applied in an industrial setting with only unlabeled data available, and how some hyperparameters in the anomaly detection model should be tuned in order to achieve sufficient performance for an unlabeled Android data set.

1.5 Scope and limitations

To counter the limitations in resources and time, some constraints are applied to the research scope. The main one is that only one method is tested for anomaly detection and the impact of other anomaly detection methods is therefore not evaluated in this research. Secondly, the impact of parameter values is only tested for a limited selection of parameters in limited intervals.

1.6 Chapter outline

The following sections in this thesis begin with the method in section 2. This section provides some background through a literature review, followed by a description of the performed interviews, and a more in-depth description of the research method and the duplicate detection pipeline. Next, section 3 presents the results achieved from the described method. The results are then discussed in section 4, which is used to present a conclusion in section 5.

Chapter 2
Method

For the method this thesis employs the Design Research Method (DRM) [31]. This method design involves understanding the current process and issues, designing a solution, and evaluating its effectiveness, which is done through the following four steps.

1. **Clarification of research task** - Define the purpose and objectives based on initial assumptions.
2. **Descriptive study 1** - Conduct a comprehensive literature review related to the purpose and objectives to create a deeper understanding of the current situation.
3. **Prescriptive study** - Use the new understanding of the situation to propose solutions for the identified problem.
4. **Descriptive study 2** - Perform an empirical study to evaluate the proposed ideas gained from step 3.

When using DRM, not all steps need to be applied and they do not need to be done in order. This method was chosen for this research since it fits well in scenarios where the goal is to improve current situations and generate solutions to an existing problem. How the steps are applied in this research can be seen in figure 1.1.

The following subsections will more in-depth describe the steps taken. The first phase involves performing a literature review and interviews with developers to achieve better insight into the research problem and goals. This is done in section 2.1 and 2.2 respectively. Next is the solution proposal in section 2.3 where the individual steps of the solution are explained.

2.1 Literature review

This section provides an overview of the background and related research through a brief literature review. The purpose is to collect the information needed to create a valid solution design. It begins with describing the Android bug reports and the general structure of the logs in section 2.1.1. This is aimed to provide the necessary background information about Android logs to understand the later sections. Section 2.1.2 then explains the concept of log parsers and three parsers that have achieved good results on Android logs. This is a prerequisite for many log anomaly detectors, which is discussed in section 2.1.3 while also outlining some existing approaches. Lastly, section 2.1.4 delves into duplicate detection and some existing methods for doing this.

2.1.1 Android bug reports

This section provides essential background information on Android bug reports, offering insights into their general structure and contents. Bug reports can be captured by users on Android platforms to gain information about the system behavior [3]. The purpose of the reports is to find and resolve bugs and do so by providing multiple files that are compressed into a zip file. This contains a detailed log of what has happened in the system during the time leading up to when the bug report is captured. This log file is very long and can sometimes extend to up to a million lines of log data.

The Android bug report logs are in turn organized into sections for the different parts of the system and different types of logs. The log sections all have their own format, except logs describing a timeline of events happening in the system. These generally follow a basic format of < **timestamp process-ID thread-ID log-level log-tag log-message** > with the log levels verbose (V), debug (D), information (I), warning (W), and error (E). Some examples of log sections in a timeline format are Crash, Kernel and Event [3]. A short example of the Event log is shown in figure 2.1. In user builds user ID is also present before the process ID.

```
------ EVENT LOG (logcat -b events -v threadtime -d *:v) ------
09-28 13:47:34.179   785  5113 I am_proc_bound: [0,23054,com.google.android.gms.unstable]
09-28 13:47:34.777   785  1975 I am_proc_start:
[0,23134,10032,com.android.chrome,broadcast,com.android.chrome/org.chromium.chrome.browser.precache.PrecacheServiceLauncher]
09-28 13:47:34.806   785  2764 I am_proc_bound: [0,23134,com.android.chrome]
...
```

Figure 2.1: Short section of Event log from example Android bug report as published on Android documentation website [3].

The different timeline log sections contain logs grouped by their own kind. As a consequence, these timelines run in parallel and are not chronological if the bug report logs are read from top to bottom. Additionally, the logs within a log section come from multiple processes and threads that run in parallel. To get a true event sequence the log lines therefore need to be separated by process ID.

2.1.2 Log parsing

The original format of logs is mostly unstructured. To be able to use them to train a model they first need to be structured [16], meaning key attributes are pulled out and used to turn the raw logs into a simplified stream of events. Logging frameworks usually have a common header format, for example as displayed in figure 2.1 which has a format of < **timestamp**

process-ID thread-ID log-level log-tag log-message >. This part is very easy to structure since it always follows the same format. The free text log message following the header is however not as simple. Usually, it is made out of a constant string combined with dynamic variables. During log parsing, the goal is to find and extract the constant string, as this is the generic event template the log line needs to be converted into. The dynamic variables are commonly replaced with a wild card token, generic character combination such as "*" or "<*>" [41], giving message templates that can look like *Service <*> took <*> ms in <*>*.

This can be achieved manually with regular expressions or Grok patterns [1]. However, this is both time-consuming and prone to error due to the amount of code and the update rate of code in modern systems [38]. Especially in the use of third-party components, it is difficult to have enough knowledge about the system to appropriately design template patterns [12]. It is therefore desirable to apply automated parsing to logs [41]. Parsing can be applied in an offline or online manner. Offline parsing is data-driven and entails that the model is pre-trained and immediately can extract the template from a raw log message [12][22]. This is possible since these types of parsing models have access to a larger amount of logs at a time that can be utilized for training before assigning log groups [13]. In other words, the log templates are in this case static and do not change over time within the session of the collected batch of logs. This is contrary to online parsing which is done in a streaming manner, where usually no training is done beforehand. Instead, log groups are dynamically added and evolving throughout the parsing process [9][17].

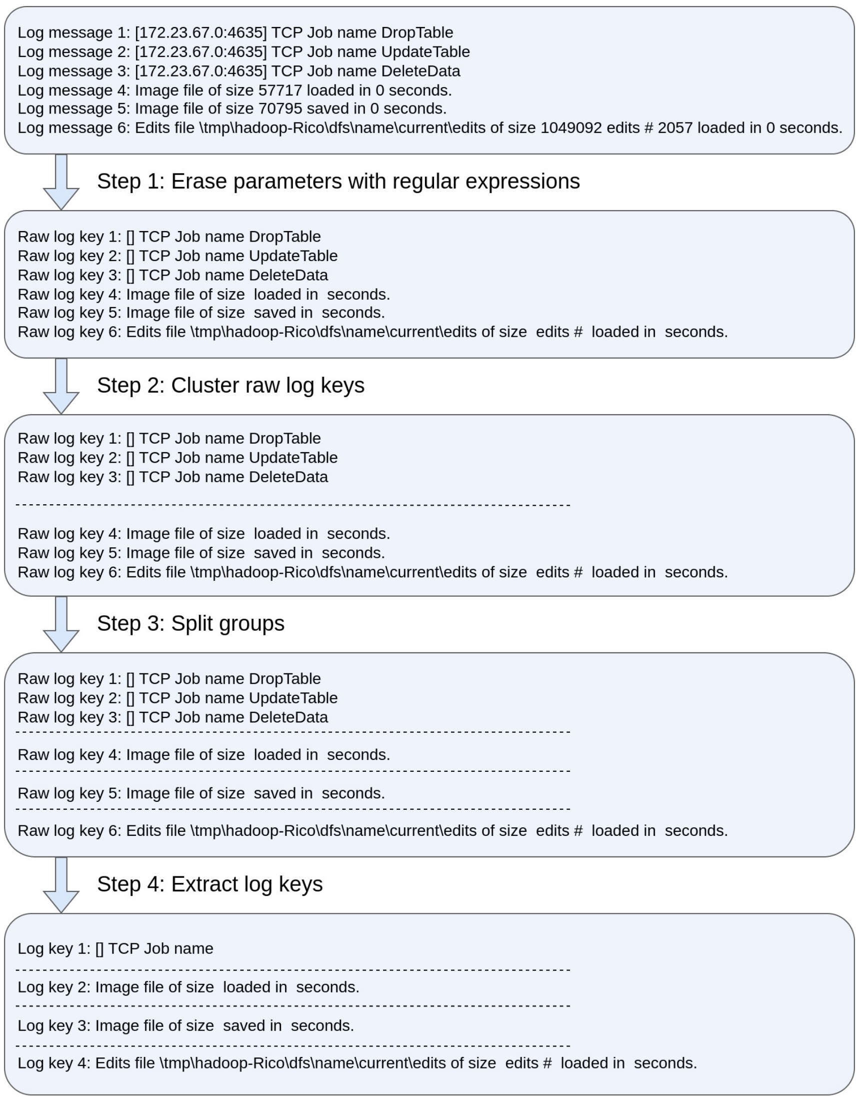
Zhu et al. [41] evaluates a range of log parsers on different data sets from the perspective of efficiency and accuracy, among other metrics. In the following subsections, the three log parsers with the highest accuracies on the Android data set are discussed.

LKE

Log Key Extraction, shortly referred to as LKE, is an offline log parser published by Fu et al. [12] in 2009 that uses hierarchical clustering and weight editing distance to assign event templates to log messages. Additionally, the authors propose to combine this with time information to detect low performance as well as using Finite State Automation (FSA) to present the normal workflow for different processes to detect anomalies, but this will not be the main focus in this section.

The process of the LKE parser is illustrated in figure 2.2. As visualized, the first step is to manually remove parts of the logs that are already known to be dynamic parameters using regular expressions. These parts can be IP addresses, URLs, text within brackets, et cetera. What is remaining is by the authors called raw log keys, and the next step is to group them using weighted edit distance and a threshold ς. Edit distance is a count of how many operations of either replacing, deleting or adding that need to be applied to a sentence before the sentences are equal. For the LKE parser, they use weighted edit distance to account for the nature of how developers write logs by valuing different parts of the logs differently. They mean that the words at the beginning of the message are more likely to be part of the template, making the dynamic variables more likely to be in the later part of the string.

The weighted edit distance is therefore calculated accordingly using $WED(rk_1, rk_2)$ as described in the Sigmoid equation 2.1. rk_1 and rk_2 respectively symbolize a raw log key where the number of operations needed for the log keys to be equal is EO, and the operations taken can be described as $OA_1, OA_2, \ldots, OA_{EO}$. x_i is the index of the word operated in


Step 1: Erase parameters with regular expressions
Step 2: Cluster raw log keys
Step 3: Split groups
Step 4: Extract log keys

```
Log message 1: [172.23.67.0:4635] TCP Job name DropTable
Log message 2: [172.23.67.0:4635] TCP Job name UpdateTable
Log message 3: [172.23.67.0:4635] TCP Job name DeleteData
Log message 4: Image file of size 57717 loaded in 0 seconds.
Log message 5: Image file of size 70795 saved in 0 seconds.
Log message 6: Edits file \tmp\hadoop-Rico\dfs\name\current\edits of size 1049092 edits # 2057 loaded in 0 seconds.
```

```
Raw log key 1: [] TCP Job name DropTable
Raw log key 2: [] TCP Job name UpdateTable
Raw log key 3: [] TCP Job name DeleteData
Raw log key 4: Image file of size  loaded in  seconds.
Raw log key 5: Image file of size  saved in  seconds.
Raw log key 6: Edits file \tmp\hadoop-Rico\dfs\name\current\edits of size  edits #  loaded in  seconds.
```

```
Raw log key 1: [] TCP Job name DropTable
Raw log key 2: [] TCP Job name UpdateTable
Raw log key 3: [] TCP Job name DeleteData
---------------------------------------------------------------------------
Raw log key 4: Image file of size  loaded in  seconds.
Raw log key 5: Image file of size  saved in  seconds.
Raw log key 6: Edits file \tmp\hadoop-Rico\dfs\name\current\edits of size  edits #  loaded in  seconds.
```

```
Raw log key 1: [] TCP Job name DropTable
Raw log key 2: [] TCP Job name UpdateTable
Raw log key 3: [] TCP Job name DeleteData
---------------------------------------------------------------------------
Raw log key 4: Image file of size  loaded in  seconds.
---------------------------------------------------------------------------
Raw log key 5: Image file of size  saved in  seconds.
---------------------------------------------------------------------------
Raw log key 6: Edits file \tmp\hadoop-Rico\dfs\name\current\edits of size  edits #  loaded in  seconds.
```

```
Log key 1: [] TCP Job name
---------------------------------------------------------------------------
Log key 2: Image file of size  loaded in  seconds.
---------------------------------------------------------------------------
Log key 3: Image file of size  saved in  seconds.
---------------------------------------------------------------------------
Log key 4: Edits file \tmp\hadoop-Rico\dfs\name\current\edits of size  edits #  loaded in  seconds.
```

Figure 2.2: Description of the steps taken during parsing in the LKE parser, based on the illustration by Fu et al. [12].

the operation OA_i and v controls the weight function. Thus, words occurring earlier in the sentence contribute more to the edit distance than words occurring later.

$$WED(rk_1, rk_2) = \sum_{i=1}^{EO} \frac{1}{1 + \mathrm{e}^{(x_i - v)}} \tag{2.1}$$

As mentioned, the weighted edit distance is used to cluster raw log keys into initial groups. Any two log keys with an edit distance less than ς get linked, and each connected cluster of logs becomes the initial group. ς can be set using a k-means clustering algorithm. This is done by calculating the difference for every possible pair of raw log keys using the weighted edit distance in equation 2.1. All edit distances are then divided into inner-class and inter-class, where inner-class distance implies that it is between two raw log keys with the same or similar log key, and inter-class oppositely means that their log keys are different. This is done using k-means clustering to cluster the raw log keys into two groups that correspond to the inner-class and outer-class where the distances are roughly the same as the edit distances. ς is then set as the greatest distance in the inner-class cluster.

The third step in the log key extraction is to split the groups further. Logs that belong to similar log keys can end up in the same groups using only the weighted edit distance. Therefore, a group-splitting algorithm is used to catch these cases. The most common word sequence in the group is used to find the *private* word sequence. In figure 2.2, a common word sequence for raw log key 4, 5 and 6 could be "**file**", "**of**", "**size**", "**in**", "**seconds**", and a private word sequence for raw log key 6 could be "**Edits**", $\varnothing$, $\varnothing$, "**edits # loaded**", $\varnothing$, $\varnothing$. We count how many different values there are for each part of the private sequences across all the raw log messages. If there are a lot of different values it might be a dynamic parameter, but if there are only a few different values it might be an indication of different log keys and the need to further split a group. This is decided with a threshold value ϱ. Groups are split repeatedly using this group splitting algorithm until no more splits can be made. Finally, the log key is extracted within each group from the common sequences.

Log keys are assigned to new logs by first using the regular expressions from the first step. The new log is then compared to existing log keys with the weighted edit distance, and if the smallest distance is less than a threshold σ the log is assigned to the corresponding log key. If the smallest distance exceeds the threshold the log message is considered an error message and its log key becomes the raw log key. Following this, abnormal sequences are detected using FSA.

As mentioned, LKE is an offline parser and needs to be trained before the real parsing is done, and consequently needs to be re-trained after system upgrades that affect the logs. The training is done on normal logs from different devices. The LKE parser has an accuracy of 0.909 on a test set of 2000 Android logs in the paper by Zhu et al. [41], but is taking days to parse their full test sets [20] which makes it slow compared to the other evaluated parsers.

Drain

Drain, fixed **d**epth t**r**ee b**a**sed onl**i**ne log parsi**n**g method, is an online parser published in 2017 by He et al. [17] that utilizes a fixed depth tree to efficiently parse logs and match them to a template format. The fixed depth tree makes sure all leaves are on the same depths and reduces the time needed to search it as it limits the number of nodes to be visited during the search.

The tree starts with a *root node* at the top of the tree and ends with *leaf nodes* at the bottom. Between the root node and leaf nodes are *internal nodes*. An illustration of the tree structure is shown in figure 2.3. The leaf nodes contain information about the log template and IDs for the log lines that are assigned to it. Before traversing the tree the log line is pre-processed. In this stage, some constants are already removed from the message using regular expressions. The regular expressions are provided by the user and are customized for the current domain. These regular expressions can entail for example numbers and IP addresses.

After this the tree is searched to find the appropriate leaf node, and with that log template, for the current log message. As seen in figure 2.3, the first layer of internal nodes describes the path to take based on the number of tokens, i.e. number of space-separated words, in the log. The next layer is based on the assumption that the first tokens in a log message are constant and hold possible values for the first word in a message, such as "Kernel" or "Detected". Depending on the set maximum tree depth, the following layers hold values for the

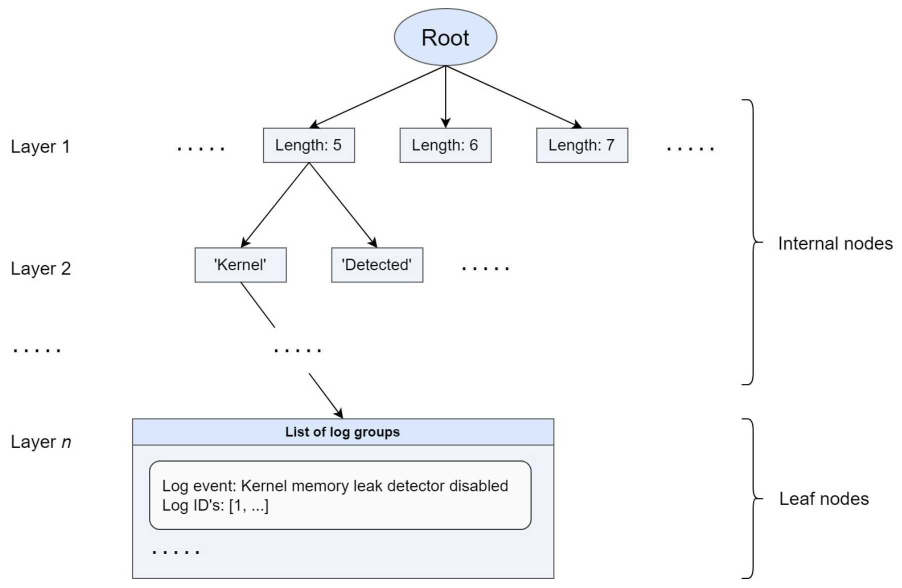


Figure 2.3: The tree structure used in the Drain parser, based on the illustration by He et al. [17].

second token, and so on. To avoid branch explosion, tokens containing digits all get summarized to the same node within that layer with a wild card token, e.g. "<*>", as value. The final layer, the leaf nodes, contains lists of log groups, which in turn hold information about the log template and the IDs for the logs assigned to it.

The log message is then compared to the log event templates inside the log groups using a similarity score that roughly measures the ratio of tokens they have in common. The highest similarity score is then compared to the similarity threshold. If the similarity is above the threshold the log message is assigned to that log group and the event template is updated. This is done by replacing any tokens in the template that do not match with the corresponding tokens in the log message with a wild card token. If the similarity instead is below the threshold, a new log group is created with a template based on the current message and the tree is updated.

As presented in the research by Zhu et al. [41], Drain is on average the most accurate log parser over the researched data sets, and the second most accurate for a test set of 2000 Android logs with an accuracy of 0.911. As mentioned previously, the tree structure makes the parsing fast compared to the other evaluated parsers [17][41]. For these reasons, it is the more commonly used log parser in research, as it is used in the pre-processing stage of multiple successful log anomaly detectors such as Deeplog [10], PLELog [39] and LogBERT [14].

Spell

Spell, short for Streaming Parser for Event Logs using LCS, is an online log parser based on the longest common subsequence computation created by Du et al. [9] in 2016. They investigate an at the time never before explored idea that the static part of a log message takes up the majority of the sequence, so two sequences that share many elements therefore likely share the same log print statement.

Initially the log entry e is divided into tokens, which are words separated by a set of delimiters such as space or equal sign. The log sequence is then used during the computation of the longest common subsequence between log entries as Σ = tokens from $e_1 \cap$ tokens from $e_2 \cap \ldots \cap$ tokens from e_L, where Σ are all possible tokens from $e_1, e_2, \ldots, e_L$. Given two sequences $\alpha = \{a_1, a_2, \ldots, a_m\}$ and $\beta = \{b_1, b_2, \ldots, b_n\}$, if γ is a common subsequence if it is a subsequence of both. The goal of the Longest Common Sequence problem is to find the longest possible such γ. The sequences γ does not need to be appearing uninterrupted in α and β, and can thus for example be $\gamma = \{1, 5, 7\}$ for $\alpha = \{1, 3, 5, 7, 9\}$ and $\beta = \{1, 5, 7, 10\}$.

For the process of the actual parsing, which can be seen in figure 2.4, the authors use a data structure that they call LCSMap, which in turn holds LCSObject data structures. The LCSObject contains information about the longest common sequence of a set of log messages and their IDs. When a new tokenized log sequence s is received, the longest common sequence $LCS(s, q_i)$ is calculated between s and the sequence q_i for every LCSObject i. The largest length of the longest calculated $LCS(s, q_i)$ as well as the index of the LCSObject is saved as l_i through the iterations. When the calculations are done, if $l_j = max(l_i's)$ is larger than a threshold τ, s is considered to belong to the same message template as q_i, otherwise a new LCSObject is created. As a default, $LCS(s, q_i)$ is expected to be at least half the length of s, thus the default threshold is $\tau = |s|/2$ where $|s|$ is the length of the tokenized log sequence.

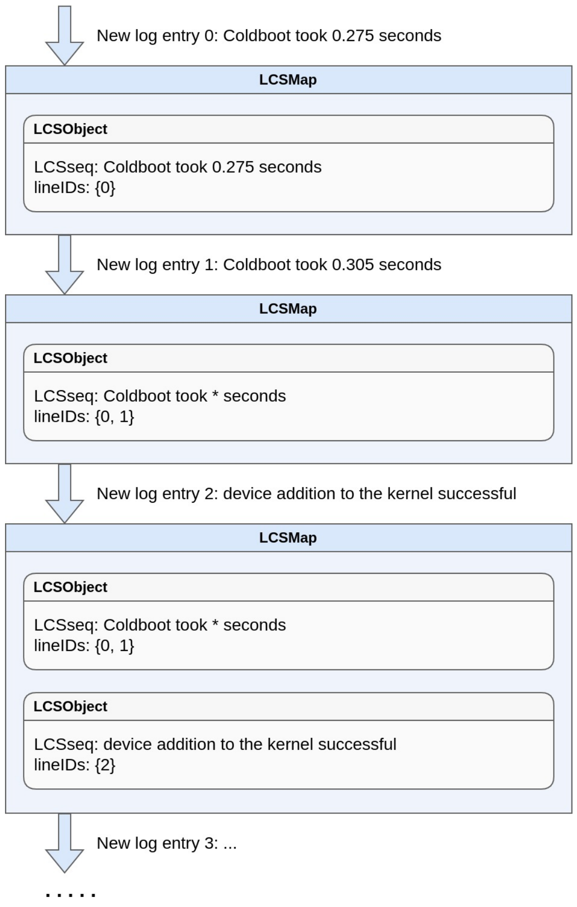

New log entry 0: Coldboot took 0.275 seconds
LCSMap
LCSObject
LCSseq: Coldboot took 0.275 seconds
lineIDs: {0}
New log entry 1: Coldboot took 0.305 seconds
LCSMap
LCSObject
LCSseq: Coldboot took * seconds
lineIDs: {0, 1}
New log entry 2: device addition to the kernel successful
LCSMap
LCSObject
LCSseq: Coldboot took * seconds
lineIDs: {0, 1}
LCSObject
LCSseq: device addition to the kernel successful
lineIDs: {2}
New log entry 3: ...
.....

Figure 2.4: The parsing process in Spell, based on the illustration by Du et al. [9].

If the threshold was reached, q_j is then updated to correctly represent the log sequences in the jth LCSObject and s. This is done by calculating the longest common sequence $LCS(s, q_j)$ and replacing tokens in q_j missing from the longest common sequence with "*". In the case of any consecutive "*"s they are merged into one.

The time complexity for this parsing method to process a log entry is linear in relation to the size of the log. For efficiency improvements, they apply optimization by detecting if the message type for a new log entry already exists. Spell has the highest accuracy of 0.989 compared to all evaluated parsers on a test set of 2000 Android logs in the research by Zhu et al. [41]. It performs well on average over the different data sets, however, specifically for the Android set the accuracy is impacted negatively as the log size grows.

2.1.3 Anomaly detection

As computer systems and applications increase in complexity, security and stability grow increasingly important. System logs play a big role in finding irregular behavior and bugs, however, due to the complexity of today's systems log analysis can be both time-consuming and error-prone when done manually. In recent years several deep learning models have been proposed to assist developers with this by analyzing the log data, a handful of which are evaluated by Le et al. [19]. The models have varying approaches to how this is done. One way is by analyzing the sequential order of the logs, commonly in a given window frame. Another is through quantitative vectors which implies counting the occurrence of each log event within a window frame. There are also cases where an embedding method like e.g. *word2vec* is used to create a semantic representation of the log templates [14][24]. In the following subsections, a more elaborate explanation is done of a few existing models.

Deeplog

Deeplog [10] is a neural network model utilizing Long Short-Term Memory (LSTM) to learn normal log execution orders to predict the next log key based on a given preceding sequence. Its base idea is that the complexity and control flows of logs from modern systems make system logs similar to natural languages, but with a smaller vocabulary and more structure. This makes an LSTM approach suitable [18]. The model is used to predict anomalies by testing if the next log key in a sequence matches the value predicted by the model.

The general architecture of the model can be seen in figure 2.5. As input it takes a sequence $seq = \{m_{t-h}, ..., m_{t-2}, m_{t-1}\}$ of window size h where each m_i is in a set of possible event keys $K = \{k_1, k_2, ..., k_n\}$, where $k_i = \{1, ..., n\}$. The output $Pr(m_t|seq)$ is a vector of the conditional probabilities of the next log key given the input sequence seq. This is achieved with a neural network with L LSTM layers, each of size α, followed by a linear layer of size equal to the number of possible event keys n. The proposed default values for L and α are 2 and 64 respectively.

The training of the neural network structure proposed in Deeplog is done with log sequences that are considered normal. For each input sequence of size h, the next coming log key is given as a label to adjust the weights to be able to predict the correct next log key. The model is trained as a multi-class classifier where the log keys in K define the classes. As previously mentioned, the model returns a probability distribution during prediction for the given input sequence. This is then employed by sorting the log keys by highest probability

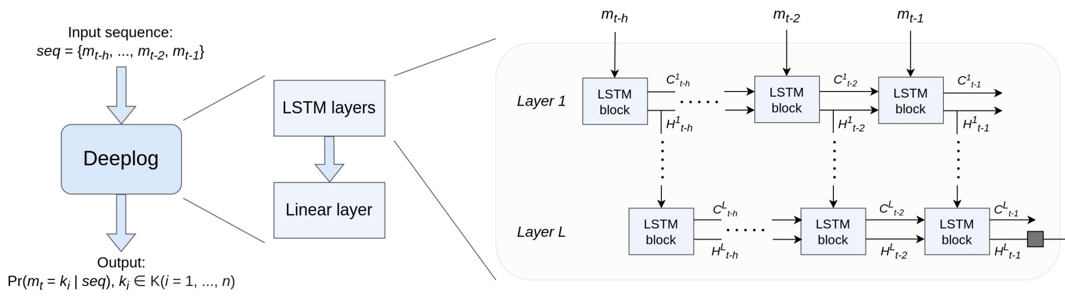


Figure 2.5: The inner structure of the Deeplog model based on the illustrations by Du et al. [10], illustrating the LSTM layer(s) followed by the linear output layer. The input is a sequence of log keys and the output is a vector of probabilities for each log key, given the input sequence.

and considering them as normal if they are among the tog g probability candidates. In the context of anomaly detection, the sequence within the window and the succeeding log key are considered an abnormal sequence if the next log key is not within the normal candidates. The default values in the paper by Du et al. [10] for window size h is 10 and top g candidates is 9.

The model is evaluated in the paper by Le et al. [19] where conclusions that were drawn included that it was a simple approach that held an advantage in only needing to be trained on normal logs. Its forecasting-based method also made it able to detect anomalies earlier than classification-based models. It does however perform less well on complex data sets and does not directly consider the semantic properties of the logs. On the other hand, it performs well on high data noise, i.e. high ratios of mislabeled logs. Another interesting finding is that Deeplog performs better on random training data than chronological, the reason being that in a random selection of data, the model gets to see future event logs during training and because of that can make better predictions.

LogAnomaly

Like Deeplog [10], LogAnomaly [24] is also based on an LSTM model and uses a forecasting-based approach for anomaly detection, where sequences diverting from the predictions are considered abnormal. In distinction to the state-of-the-art anomaly detection models at the time, LogAnomaly also uses quantitative vectors in addition to the sequential vectors during prediction.

During the training stage, they use a log parser to extract event templates from log lines. These are in turn converted into embeddings by combining word embeddings generated with Word2Vec [25] through weighted average, from which log sequences are then modeled as vectors of log template embeddings. During prediction, log lines are matched to log templates and turned into embeddings. If no matching template is found the log template is generated by the parsing tool and is turned into an embedding and matched to the template embedding vector with the highest similarity. This novel approach of using embeddings for the template vectors is called *Template2Vec* and is based on synonyms and antonyms constructed in the training stage.

The general structure of LogAnomaly as described in the paper is shown in figure 2.6.

As input it takes both a sequential vector $\mathbf{V}_j = (\mathbf{v}_{s_j}, \mathbf{v}_{s_{j+1}}, ..., \mathbf{v}_{s_{j+w-1}})$ as well as a count vector $\mathbf{C}_j, \mathbf{C}_{j+1}, ..., \mathbf{C}_{j+w-1}$, and with this it generates an output that is the most probable next log template $\mathbf{v}_{s_{j+w}}$ given the inputs. The original log sequence $\mathbf{S} = (s_1, s_2, \ldots, s_m)$ is stepped through with a window of size w, giving subsequences like $\mathbf{S}_j = (s_j, s_{j+1}, \ldots, s_{j+w-1})$. $\mathbf{V}_j$ is the embedded vector template sequence for $\mathbf{S}_j$. The count vector is generated by for each log message $s_i \in \mathbf{S}_j$, taking the sequence of logs leading up to it $(s_{i-w+1}, s_{i-w+2}, \ldots, s_i)$ and calculating the count vector $\mathbf{C}_i = (c_i(\mathbf{v}_1), c_i(\mathbf{v}_2), \ldots, c_i(\mathbf{v}_n))$. In the count vector $c_i(\mathbf{v}_k)$ is the number of $\mathbf{v}_k$ in the embedded template vector sequence $\mathbf{S}_i = (\mathbf{v}_{(s_{i-w+1})}, \mathbf{v}_{(s_{i-w+2})}, \ldots, \mathbf{v}_{(s_i)})$. This gives $\mathbf{C}_j, \mathbf{C}_{j+1}, ..., \mathbf{C}_{j+w-1}$ for $\mathbf{S}_j$ as input to the model. The authors propose the default settings of the model as two LSTM layers with 128 neurons and a window size of 20.

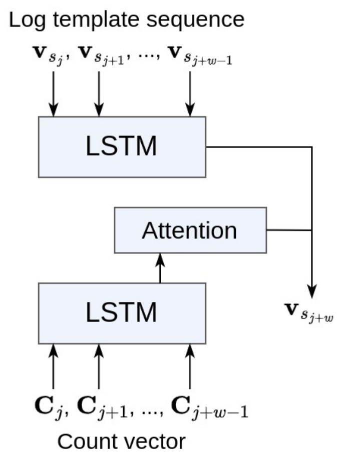

Log template sequence
$\mathbf{v}_{s_j}, \mathbf{v}_{s_{j+1}}, ..., \mathbf{v}_{s_{j+w-1}}$
LSTM
Attention
LSTM
$\mathbf{v}_{s_{j+w}}$
$\mathbf{C}_j, \mathbf{C}_{j+1}, ..., \mathbf{C}_{j+w-1}$
Count vector

Figure 2.6: Inner structure of the LogAnomaly model as proposed by Meng et al. [24]. Both the sequence of embedded log templates and count vector is taken as input.

In the paper by Le et al. [19] it is noted that Loganomaly [24], similarly to Deeplog [10], only requires normal data during training, can detect anomalies early and is less sensitive to mislabeled logs. However, it also holds similar downsides to Deeplog by performing better on simpler data and the fact that it is impacted by any potential log parsing errors. Additionally, the combination of sequential and quantitative vectors makes the model itself more complex. On the other hand, it possesses the strength of not immediately marking an unknown log line as an anomaly. Due to the feature of being able to match new log templates to existing ones, it can avoid some false positives.

PLELog

PLELog [39], named by its Probability Label Estimation properties, detects anomalies with an attention-based GRU network. Unlike the previously mentioned anomaly detection methods, this is a fully semantic approach. It also only requires labels for a small set of the normal logs in the training data set which it can use to estimate the labels of the remaining data set.

As shown in figure 2.7, the approach consists of three main stages; semantic embedding, probabilistic label estimation, and anomaly detection. The semantic embedding stage itself is also divided into three parts, the first being log parsing done with Drain [17] (see section 2.1.2) to extract log event templates. The log events are then stripped of any non-character tokens and stop words, and concatenated words such as NullPointerException that can occur in system logs are divided based on the Camel Case rule [8]. The semantic word vectors are then extracted for each of the words in the log events using the pre-trained Glove model [26]. Lastly, the word vectors are aggregated with consideration for the importance of each word which is done with TF-IDF [32].

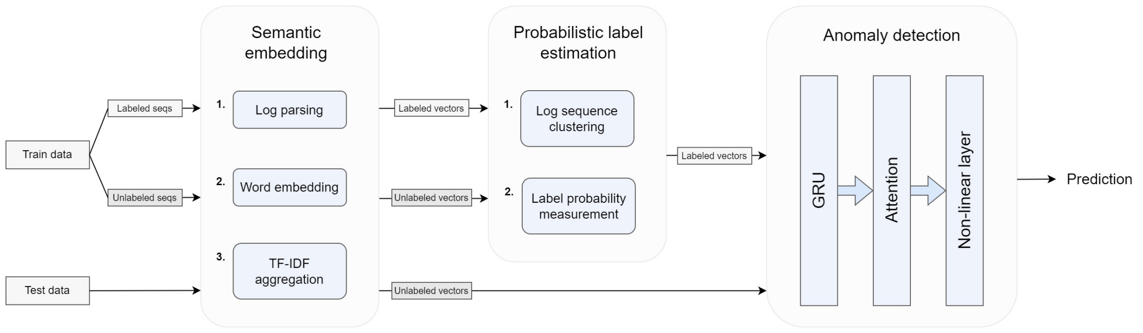


Figure 2.7: The process of using PLELog on training and test data based on the illustration by Yang et al. [39].

The second stage as shown in figure 2.7 is the probabilistic label estimation for unlabeled log sequences based on the known normal log sequences. This is done with the assumption that log sequences with similar semantics are likely to have the same label by doing semantic clustering of the summed semantic vectors from each log sequence. The labeled logs within each cluster are then used in combination with each unlabeled log sequence's certainty score produced by the clustering algorithm HDBSCAN [23] to determine their probability to be either normal or abnormal. This is only done on the training data.

The anomaly detection model is then trained on the pre-labeled and probabilistically labeled data. For this, it uses a GRU neural network combined with an attention-based masking layer. This makes the network assign larger weights to log events that have a stronger correlation with the anomaly detection result and smaller weights to noisy log events. A non-linear layer then predicts whether the log sequence is normal or abnormal.

Le et al. [19] evaluate PLELog as advantageous in the way that it does not require a fully labeled data set and can estimate labels. The use of semantic vectors and attention-based GRU also increases its effectiveness. However, it is slow compared to other anomaly detectors due to the clustering model. It is also not performing well in early anomaly detection, affecting its abilities in online scenarios. Surprisingly, according to Le et al. [19], it also performed worse on noisy data, something the attention layer meant to combat in the original paper.

2.1.4 Duplicate detection

The current duplicate detection tools mainly involve the comparison of titles, descriptions, and other structured information in the bug tickets. The approaches to doing this involve a range of methods such as dual-channel convolutional neural networks (dual-channel CNN), bidirectional long short-term memory (bi-LSTM), recurrent neural networks (RNN), and multilayer perceptron (MPL) [40]. They also vary in how the decisions about duplicates are made. One method being a *ranking* setting and another being a *classification* setting. The ranking setting involves comparing a ticket to multiple others and ranking their probability of being duplicates, while classification entails deciding on whether they are duplicates or not given a pair of tickets [40]. The following sections explain some existing duplicate detection models.

REP

REP [33] is a popular retrieval-based approach proposed over a decade ago by Sun et al. It uses textual features such as summary and description fields as well as the categorical features product, component, type, priority, and versions to compare the similarity between bug tickets. For the textual features, it calculates textual similarity with an extended version of BM25F for the summary and description field as bi-grams and uni-grams respectively. For product, component and type, the feature gets the value 1 if they are equal and 0 otherwise. Lastly, priority and version get the value of the inverse of the difference between the tickets. This is then applied through a retrieval function where the weights of the different features can be adjusted. The adjustable features can be optimized through stochastic gradient descent with past bug tickets.

Siamese Pair

Siamese Pair [7] is a duplicate detection model based on a combination of Siamese CNN and LSTM. It handles short descriptions, long descriptions and structured information, such as version and priority, separately. The descriptions are first turned into vector representations, and the short description is then encoded with a bi-directional LSTM and the long description with a CNN. The structured information is encoded using a feed-forward network. All features are then combined to represent the ticket in a comparable state that can be used either categorically or with ranking.

SABD

Soft Alignment Model for Bug Deduplication (SABD) [27] consists of two sub-networks for textual and categorical information. As input, it takes two bug tickets to be compared. The textual information, being the free text in the summaries and descriptions, is handled by the textual module. This module consists of an advanced architecture including textual embedding, soft alignment comparison, textual encoder, and textual comparison. The soft attention alignment in the second layer distinguishes this model by allowing for interaction between the text content from the tickets. It computes a similarity score between tokens from both tickets, enabling the selection of relevant features from each input. The categorical module includes an embedding layer, encoding and, finally, comparison. Instead of assigning the categories a binary value based on whether they are equal or not, embeddings are used to account for similar category inputs. The results of the comparisons from the two sub-networks are then combined and evaluated through classification.

HINDBR

A more recent approach is Heterogeneous Information Network based Duplicate Bug Report prediction (HINDBR) [37] which is a deep neural network (DNN) that utilizes a heterogeneous information network (HIN) to detect semantically similar bug tickets. HIN is a more complex type of network that can have multiple types of nodes and edges. This is used by creating nodes for possible features and letting tickets that share a node connect through it. For textual features like summary, an RNN is used for sequence embedding. Textual information and learned relations from the HIN are then embedded with a DNN to calculate the similarity scores.

DC-CNN

Another recent approach is DC-CNN [15] which uses a dual-channel CNN to determine if bug ticket pairs are duplicates. It uses the component, product, summary, and description from the ticket which are added to a text document and pre-processed, involving tokenization, stemming and removal of stop words and common industry words. The data is then turned into single-channel embedding matrices with word2vec which are compared using the DC-CNN model. The convolutional part of the network uses convolution and pooling to extract keywords in the document. The dual-channel property implies the use of two channels in the convolutional layers so two bug tickets can be processed at once. The extracted features are then merged to predict a similarity score between the two tickets and use the classification method to determine if they are duplicates based on a similarity threshold.

2.2 Interviews

To design a possible solution, an understanding of common approaches for manually analyzing bugs and identifying them as duplicates is needed. For research centered on understanding experiences and processes, interviews are an appropriate approach. They require less experience than questionnaires and also work well when the interviewer's experience on the topic is inadequate [28]. In the following sections, the structure and execution of the interview are explained.

2.2.1 Interview structure

In this interview process, a semi-structured approach was used. This is one of three interview categories and was picked due to it is suitability for novice researchers. The other interview types are unstructured and fully structured [30]. These differ in how strictly the interview is planned and how strongly the plan is followed. The semi-structured approach used for this interview is the most common and is a mix of both. It has a prepared plan of questions, but it does not have to be asked in the exact order and there is room for a natural conversation to develop similarly to the unstructured interview [28][30].

Additionally to how strictly the interview plan is executed, the interview was also divided into different phases. First, the participant was introduced to the background of the research and any other relevant information. After that, the participant was asked introductory questions to earn the necessary knowledge about their background. Then came the main interview part which took the majority of the interview time [30].

The interview questions consist of two subparts. The first being questions regarding general bug report log analysis and the second part is questions where the participant was directly asked how this could be applied in machine learning. The full questions can be found in Appendix A.

2.2.2 Participant selection

The goal of the interviews was to collect broad data to get a generalized understanding of current approaches.

The interviews were performed on four developers of different backgrounds and experience levels. The areas of work consisted of launcher and apps, Bluetooth and connectivity, testing, and park assist camera. Their experience of Android bug reports, measured in time, varied greatly between a year and ten years. The same could be observed for how often they analyzed Android bug report logs, which ranged from every day to once a week or less.

2.2.3 Interview process

All interviews were held in person, where the participant was first asked for permission to record the interview and given an introduction to the research. They were then asked questions about their background, followed by the prepared interview questions. Afterward, the participant was allowed to add any additional information they found relevant. Notes were taken during the interview process, which was then supplemented with complementary information from the recording of the session. The average length of the interviews was half an hour.

2.2.4 Interview results

The knowledge gained from the interviews indicated that different teams encounter duplicate bug reports with varying frequencies, where the answers differed between approximately 10% and 95% of bug reports having duplicates. In the specific case of 95% it however often involved a few common errors that could usually be found with a special GREP command.

To identify duplicates many look at the ticket title, description, and potential images and other media as a first step. After that they look in the log file, often narrowing it down to the log lines in the time interval surrounding the time shown in images or videos if available. Many interviewees found it hard to answer which log section they mainly look at, as that is hard to know when you are searching in the full file. However, common for all included teams seemed to be that they look at log lines commencing with a timestamp, with the structure described in section 2.1.1.

It was mentioned that challenges arising when identifying duplicate Android bug reports can be that it is hard to know what to look for in the logs in the case of less previous experience. There are also occasions where two tickets that look to belong to the same bug are actually caused by two different errors, as well as situations where an error creates two seemingly different bugs and tickets.

The consequence of duplicates of Android bug reports not immediately being identified is generally wasted time and energy by the developers. But it is usually discovered if two developers work on the same bug when it is discussed during daily meetings and it is realized that the core error seems to match. The time taken to discover this way that two Android bug reports are duplicates can vary and relies on the developer finding what causes the bug. When duplicates are found the tickets are linked and typically the newer one is closed.

The contribution of machine learning on duplicate identification of Android bug reports expressed by the interviewees can be summarized as that it would be good with a database that makes it possible to look for recurring bugs, and that it would be helpful to apply machine learning when the bugs are hard to find.

2.3 Solution design

Given the information gained from the literature review and interviews described in sections 2.1 and 2.2, the solution design in figure 2.8 is proposed. The first step involves bug report collection, which is presented in section 2.3.1. This part explains the bug report data used in the method and how it is retrieved. The pre-processing of the bug report log data is then described in section 2.3.2 where a description of the filtering and log parsing can be found. Section 2.3.3 presents the anomaly detection step, followed by section 2.3.4 where the process of the duplicate detection is explained. Lastly, section 2.3.5 details the evaluation metrics used when presenting the final results.

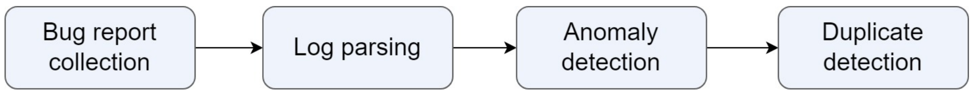


Figure 2.8: Solution proposal.

2.3.1 Bug report collection

The data to be used in this research consists of Android bug reports captured from Android-based devices at Volvo Cars. The collection of the bug reports was done through their customized version of Jira [5], which is a tool used for issue tracking and project management. Using the API offered by the Jira platform, a script was developed to download all bug reports from tickets matching a given search query, for example including assigned team or status. When downloaded, the bug reports are compressed zip files, where the data of interest is a bug report text file inside. The format of this file is more elaborately described in section 2.1.1.

The Jira API also provides information about the bug ticket. One example of this is information about which other tickets are connected to it as a duplicate or clone. This data is also retrieved to be able to connect which bug reports are duplicates.

For this research all available bug reports for Android version R and S assigned to two of the software engineer teams at Volvo Cars were downloaded, resulting in a total of 1351 bug reports at the time. The chosen teams were the Bluetooth team and the Launcher team due to convenience and closer collaboration throughout the research process. An additional reason for limiting the data to only two software developer teams was to constrain the scope of the research.

Out of all bug reports approximately 300 were part of a duplicate grouping. 51 of the among these most recently added to Jira were separated from the full data set and used as a test set in the final grouping stage. This leaves us with 1300 bug reports for training and validation during the anomaly detection stage. The reason for only using bug reports with guaranteed duplicates during the final duplicate prediction is due to the nature of the data set. Bug reports marked grouped as duplicates are sure to be duplicates, while bug reports that have not been marked as duplicates are not guaranteed to not be an undiscovered duplicate of another bug report.

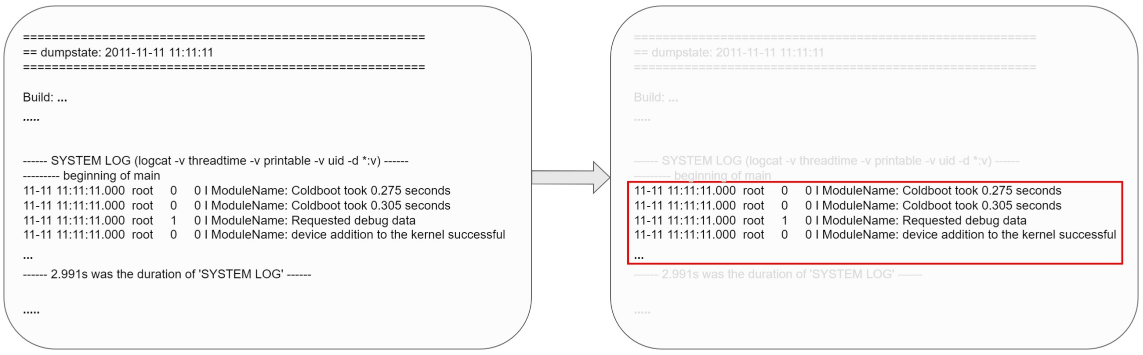
2.3.2 Pre-processing

To apply anomaly detection, the data need to be pre-processed. This involves filtering and parsing logs to extract event keys. The following sections describe how this is done.

Filtering

A bug report log file contains hundreds of thousands of lines of logs. It is therefore necessary to filter out log lines that are not important during comparison to accurately detect duplicate bug reports. As mentioned in the summary of the interviews presented in section 2.2.4, the logs predominantly analyzed among developers are kinds starting with a timestamp. For simplicity and compatibility with the used tools we primarily focus on these log types within this research.

Given this, the first step of pre-processing the data is to filter out only the lines of such format, as illustrated in figure 2.9. It was also found that some lines were missing user IDs, and these were removed as these exact lines were also present in other parts of the logs with a user ID included. This filtering step was necessary since the used log parsing method assumes that the log has a consistent format of < **timestamp user-ID process-ID thread-ID log-level log-tag log-message** >. The log lines of this format make up only about 5-20% of the total log lines, so this significantly reduces the amount of data and removes a lot of information that is not needed in this context.



```
========================================================
== dumpstate: 2011-11-11 11:11:11
========================================================

Build: ...
.....


------ SYSTEM LOG (logcat -v threadtime -v printable -v uid -d *:v) ------
--------- beginning of main
11-11 11:11:11.000  root    0     0 I ModuleName: Coldboot took 0.275 seconds
11-11 11:11:11.000  root    0     0 I ModuleName: Coldboot took 0.305 seconds
11-11 11:11:11.000  root    1     0 I ModuleName: Requested debug data
11-11 11:11:11.000  root    0     0 I ModuleName: device addition to the kernel successful
...
------ 2.991s was the duration of 'SYSTEM LOG' ------

.....
```

```
11-11 11:11:11.000  root    0     0 I ModuleName: Coldboot took 0.275 seconds
11-11 11:11:11.000  root    0     0 I ModuleName: Coldboot took 0.305 seconds
11-11 11:11:11.000  root    1     0 I ModuleName: Requested debug data
11-11 11:11:11.000  root    0     0 I ModuleName: device addition to the kernel successful
...
```

Figure 2.9: Illustration of the filtering step where only log lines following the format of < **timestamp user-ID process-ID thread-ID log-level log-tag log-message** > is kept.

Log parsing

To measure the similarity between bug reports we need to make log lines comparable. To do this we need to remove variable data from the constant strings in the log lines. This can be done manually with regular expressions or Grok patterns [1], but for this research, we utilize one of the multiple log parsing tools available for this purpose as discussed in section 2.1.2. The parser was chosen based on accuracy and efficiency for mainly Android logs in the study by Zhu et al. [41].

In this research, Drain [17] is used because it is the most accurate and among the more efficient parsers for Android logs as the data set grows. The Spell parser is the most accurate on a limited data set but performs poorly as the Android data set grew, converging to an accuracy of 0.2 [41]. This makes it unsuitable for the large data set used in this research. Further, the LKE parser was considered as unfit due to its parsing time making it unattainable to parse larger logs. It also needs to be re-trained between system updates due to its offline characteristics, which makes it less of a good choice from a long-term perspective.

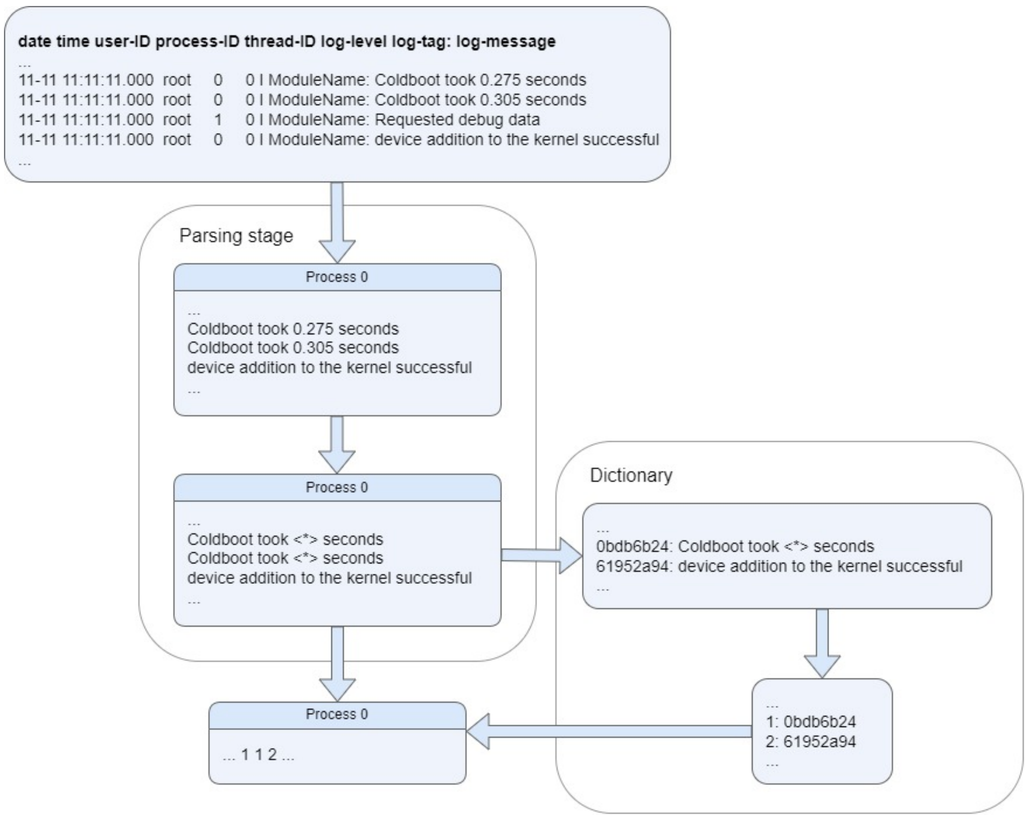
The implementation of Drain is based on the implementation provided in the toolkit [21] made for Zhu et al. [41] and the parameters were set according to their default values for Android logs as similarity threshold 0.2 and tree depth 6. The general process of the log parsing can be seen in figure 2.10. The full log file is given to the parser which extracts the log message and separates them based on process ID. Each process is then parsed and the logs are encoded and assigned event keys which are concatenated to a single line that describes the order of the logs within the process. In the final encoded file, each line represents a process and the string of numbers shows the order of the log events.

All processes between all parsed bug reports share the same dictionary and parsing tree which can be seen in figure 2.3. The decision to share the tree between all bug reports is to make the parsing more consistent between bug reports, as the strategy of the Drain parser causes two identical lines from different files to be parsed differently if similar lines are occurring with different frequencies between files. For this purpose the tree data structure is saved in a pickle [11] file to be loaded in later runs.

2.3.3 Anomaly detection

Abnormal sequences are detected in the logs using the encoded log event sequences acquired from Drain. To achieve this, the first step is to learn normal sequences. LSTM models are very good at learning sequences and have successfully been used to learn human languages which is an even more complicated task than learning log sequences. In this step, we therefore use the LSTM model approach Deeplog [10]. LogAnomaly [24] similarly uses LSTM modules in its model with the addition of also analyzing the quantitative vectors. However, Deeplog achieves similar results with a simpler structure and was, therefore, easier to find a correct implementation for. We used a public implementation of Deeplog [36] that matched the specifications in the paper [10].

The Deeplog model is used with the default parameters of 2 layers and layer size 64, as proposed in the original paper by Du et al. [10], and is trained for 10 epochs given the computational resources available. The paper proposes a top-g of the highest probability candidates to be considered normal during the prediction of the next event key. Different values of g and h are evaluated in the experiment. The default values provided for Deeplog



```
date time user-ID process-ID thread-ID log-level log-tag: log-message
...
11-11 11:11:11.000  root    0    0 I ModuleName: Coldboot took 0.275 seconds
11-11 11:11:11.000  root    0    0 I ModuleName: Coldboot took 0.305 seconds
11-11 11:11:11.000  root    1    0 I ModuleName: Requested debug data
11-11 11:11:11.000  root    0    0 I ModuleName: device addition to the kernel successful
...
```

```
Process 0
...
Coldboot took 0.275 seconds
Coldboot took 0.305 seconds
device addition to the kernel successful
...
```

```
Process 0
...
Coldboot took <*> seconds
Coldboot took <*> seconds
device addition to the kernel successful
...
```

```
...
0bdb6b24: Coldboot took <*> seconds
61952a94: device addition to the kernel successful
...
```

```
...
1: 0bdb6b24
2: 61952a94
...
```

```
Process 0
... 1 1 2 ...
```

Figure 2.10: Description of the parsing process and how log line belonging to different process ID's are separated for some example log lines.

are $g = 9$ and $h = 10$. These default values are therefore explored in this research along with a range of surrounding values for a comprehensive evaluation, with 7, 8, 9, 10, and 11 for g and 8, 10, 12, 14, and 20 for h. Window size 20 was added to learn about the impact of larger window sizes.

Given the data described in section 2.3.1, the bug report lines used for training contain both normal and abnormal sequences. The quantity of the data and the required knowledge make it impossible to accurately label the sequences, and the sequences therefore remained unlabeled. This played a key part in the choice of an appropriate model. The Deeplog model is preferably trained on normal sequences, but as this is not possible in this case we utilize the probability attribute in the model's design. We make the conjecture that abnormal sequences that occur rarely are not present among the top predictions and therefore do not make a too big impact on the anomaly detection result. It was shown by Le et al. [19] that Deeplog holds some tolerance to mislabeled logs and remains giving stable results as the ratio of mislabeled logs grows.

2.3.4 Duplicate detection

The first step of the duplicate detection stage is using the trained Deeplog model to predict the next event log key in a sequence. The event keys are iterated over with a sliding window of size h, one step at a time, as described in figure 2.11. For each sequence of h event keys the next is predicted with the trained model. If the actual next event key is among the top g probability event keys it is considered a normal sequence and we move on to the next iteration. However, if the actual event key is not among the top g probability event keys it is considered an anomaly and the sequence of h event keys and the following event key are saved as a list in a list of abnormal sequences for the given bug report. After this, the bug report is compared to other bug reports by calculating the ratio of how many sequences are matching between their abnormal event key sequences. The duplicate detection can then be done in two ways, as described in the following sections.

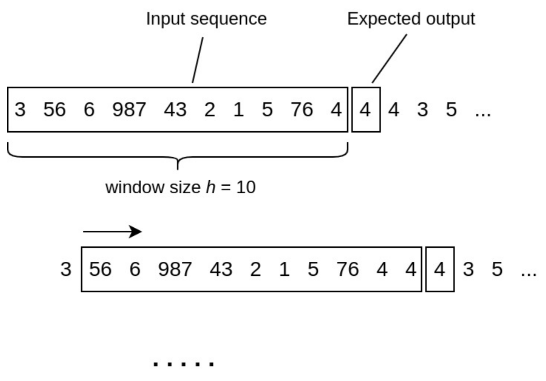


Figure 2.11: During training and prediction with the Deeplog LSTM model, the event sequences are iterated over with a sliding window with a set size h to learn or predict the correct next event key.

Classification

With the classification method, the bug reports in the test set are compared to all other bug reports in the test set this way by comparing their anomalies. If the match ratio is above a threshold t the bug reports are assumed to be duplicates of each other [40]. Based on observations, the value for t is set to 0.1 which requires that at least 10% of their abnormal logs overlap.

Ranking

The ranking method compares the anomalies of bug reports in the test set to all others in the test set. The k bug reports with the highest match ratio are assumed to be possible duplicates [40]. If any of the proposed matches is a duplicate of the current bug report, the guess is considered correct during evaluation.

2.3.5 Evaluation metrics

The effectiveness of the duplicate detection results with the classification method (see section 2.3.4) are measured using precision, recall and F-score. These evaluation metrics provide an assessment of the effectiveness of duplicate detection in accurately identifying and pairing duplicate bug reports. When defining these metrics positive and negative instances are used.

Positive instances entail a bug report being a duplicate to another, where true positive indicates a correct pairing and false positive an incorrect pairing. Similarly, negative instances mean a bug report is not a duplicate of the other, with true negative being a correct prediction of the bug reports not being duplicates of each other and false negative an incorrect prediction. The evaluation metrics for the classification method are defined as follows.

- **Precision** is the fraction between correctly predicted positive instances and all predicted positive instances, indicating the accuracy of positive predictions.

$$Precision = \frac{True\ positive}{True\ positive\ +\ False\ positive}$$

- **Recall** is the fraction between correctly predicted positive instances and all actual positive instances, giving a measure of how well we detect positive instances.

$$Recall = \frac{True\ positive}{True\ positive\ +\ False\ negative}$$

- **F-score** is the harmonic mean of the precision and recall, giving a combined balanced measure of both values.

$$F\text{-}score = 2 \cdot \frac{Precision\ \cdot\ Recall}{Precision\ +\ Recall}$$

The evaluation metric used for the ranking method is *Recall Rate@k*, shortly referred to as *RR@k*. It was chosen due to it being the most consistently used evaluation metric in research about duplicate detection [4][7][29][34][35]. This metric is based on the ranking of the potential duplicate bug reports based on the match ratio. It is defined as

$$RR@k = \frac{n_k}{m},$$

where n_k is the number of bug reports in the test set, i.e. the number of prediction instances, for which at least one correct duplicate is found in the top k positions in the ranking. m is the total number of bug reports in the test sets, thus the total number of performed predictions [40].

Chapter 3

Results

This section presents the results from the models from the method described in section 2, divided into classification and ranking evaluation methods as described in section 2.3.5. The baseline system had 2 LSTM layers of size 64 respectively and was trained for 10 epochs with varying values for window size h. This model was then used during prediction to find anomalies to be used during the duplicate detection. The results for the classification method are presented in section 3.1 and for the ranking method in section 3.2.

3.1 Classification

For the classification method, prediction was first done with the different window size models and default top g probability = 9 to see the impact of different values in window size h. This gave the results presented in section 3.1.1. Likewise, different values were then used for top g probability during prediction with the model trained with default window size h = 10 to see the impact of varying top g probability values. These results can be seen in section 3.1.2. Combined values for the results from a range of top g probabilities and window sizes to see the relation between the two is lastly presented in section 3.1.3.

3.1.1 Window size h

This section presents the results for varying values for window size h as can be seen in figure 3.1. The results are for h in the range 8, 10, 12, 14 and 20, with the default value 9 for top g probability. The information in the graph indicates that a growing widow size h increases the precision. The recall trend is however not as clear as the precision, but the collected data suggests a slight downward turn in recall as the window size increases to large values. Together, the precision and recall provide an F1-score that strongly follows the trend of the recall.

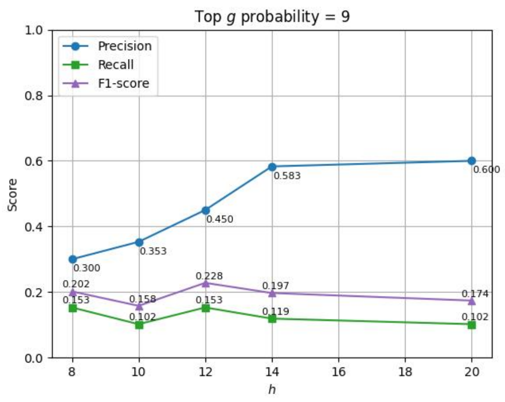


Figure 3.1: Evaluation metrics for different values for window size h for default top g probability = 9.

3.1.2 Top g probability

In this section, the results for varying values of top g probability in the range 7 to 11 are presented in figure 3.2. Default value 10 was used for window size h. One can see a slight downward trend for all evaluation metrics as g grows. Out of the range of values investigated, the smaller values 7 and 8 give the highest precision, recall and F1-score.

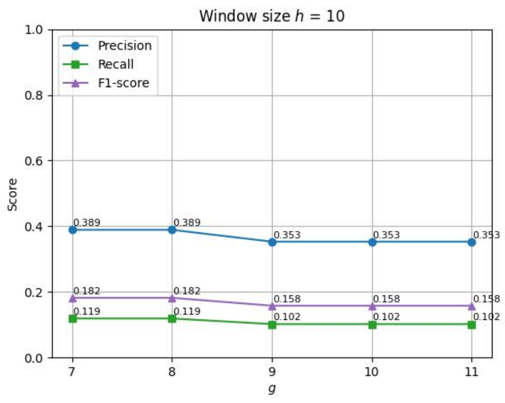


Figure 3.2: Evaluation metrics for different values for g top probability for default window size h = 10.

3.1.3 Combined top g probability and window size h results

This section presents a summary of values for precision, recall and F1-score for top g probability in the range 7 to 11, and window size h in the range 8, 10, 12, 14 and 20. This can be seen in the tables 3.1a-3.1c. Here one can see that a smaller g around 7 and 8 and potentially lower gives the best performance in all validation metrics regardless of the window size. For window size h, precision is the highest for $h = 14$, while recall is highest for $h = 8$ and F1-score for $h = 12$.

h g	8	10	12	14	20
7	0.312	0.389	0.421	**0.615**	0.600
8	0.333	0.389	0.450	**0.615**	0.600
9	0.300	0.353	0.450	0.583	0.600
10	0.258	0.353	0.421	0.583	0.600
11	0.310	0.353	0.421	0.583	0.600

(a) Precision.

h g	8	10	12	14	20
7	**0.169**	0.119	0.136	0.136	0.102
8	0.153	0.119	0.153	0.136	0.102
9	0.153	0.102	0.153	0.119	0.102
10	0.136	0.102	0.136	0.119	0.102
11	0.153	0.102	0.136	0.119	0.102

(b) Recall.

h g	8	10	12	14	20
7	0.220	0.182	0.205	0.222	0.174
8	0.209	0.182	**0.228**	0.222	0.174
9	0.202	0.158	**0.228**	0.197	0.174
10	0.178	0.158	0.205	0.197	0.174
11	0.205	0.158	0.205	0.197	0.174

(c) F1-score.

Table 3.1: Precision, recall and F1-score for combinations of values on top g probability and window size h.

3.2 Ranking

For the ranking method, a range of values was used for window size h and top g probability to see the results of different combinations. As earlier described in section 2.3.5, the evaluation metric $RR@k$ is used. These results can be seen in the tables 3.2a-3.2c with 1, 3 and 5 as values for k.

For $k = 1$ the highest $RR@k$ was 0.275 for $h = 8$. For $k = 3$, the $RR@k$ for $h = 8$ stays roughly the same, while it for other values has increased by approximately between 0.15 and 0.275. Here the window size $h = 20$ was best with $RR@k$ 0.471 for $g = 10$, followed by $h = 10$ with $RR@k$ 0.431 with $g = 11$. Finally, for $k = 5$ the top $RR@k$ is 0.588 for $h = 12$ with $g = 9...11$. In the step between $k = 3$ and $k = 5$, all value combinations had a noticeable increase in *RR@k* with around 0.1 or more. The top *RR@k* increased by 0.117, i.e. 11.7 percentage points.

h g	8	10	12	14	20
7	**0.275**	0.216	0.235	0.196	0.216
8	0.255	0.196	0.235	0.196	0.217
9	0.255	0.235	0.255	0.196	0.176
10	**0.275**	0.216	0.255	0.216	0.196
11	**0.275**	0.216	0.255	0.216	0.216

(a) $k = 1$.

h g	8	10	12	14	20
7	0.275	0.373	0.333	0.373	0.451
8	0.275	0.392	0.333	0.373	0.451
9	0.275	0.412	0.333	0.373	0.412
10	0.294	0.412	0.353	0.373	**0.471**
11	0.294	0.431	0.353	0.353	0.392

(b) $k = 3$.

h g	8	10	12	14	20
7	0.431	0.529	0.549	0.510	0.529
8	0.431	0.529	0.529	0.490	0.549
9	0.412	0.549	**0.588**	0.490	0.490
10	0.412	0.549	**0.588**	0.549	0.549
11	0.471	0.549	**0.588**	0.549	0.510

(c) $k = 5$.

Table 3.2: *RR@k* for combinations of values for top g probability and window size h for $k = 1, 3, 5$.

Chapter 4
Discussion

This section discusses the results presented in section 3. The findings are evaluated centered around the research questions in section 4.1, followed by a discussion of limitations and threats to validity in section 4.2 and potential future work in section 4.3.

4.1 Findings

This section discusses the answers to the research questions from section 1.1.

4.1.1 RQ1: Can log anomaly detection be used in duplicate detection between bug report logs in an industrial setting where the data is unlabeled?

Based on the results in section 3, there is potential for anomaly detection to be used together with duplicate detection in an industrial setting with unlabeled data. For such an application the ranking method is likely most appropriate [40], as it makes sense to return the k most likely duplicates to the user, rather than the user wanting to compare pairs of bug reports manually.

For $k = 1$, the highest *RR@k* was 0.275, which indicates a correct prediction roughly a fourth of the time. This is fairly good considering all the steps and parameters involved in the duplicate detection pipeline. This could be used in a real scenario, although a higher *RR@k* would likely be desirable. When k is increased to 3, a *RR@k* of 0.471 is achieved. This is a good recall given that the developer then only has to look through 3 bug reports to find the potential duplicate. For $k = 5$ the highest *RR@k* was 0.588. This is a good increase, but given the trade-off that the developer now has 5 candidates as possible duplicate candidates that will take time to evaluate and confirm. The chance of any of them being true being only slightly over half of the time is potentially not worth it.

The more appropriate value for k would therefore likely be 3. As mentioned, the developer would then only have to look further at 3 bug reports to find the potential duplicate. An almost 50% chance of finding a match among three candidates can be seen as sufficient performance for the application within an industrial setting, especially if viewed as a tool in addition to current methods, as expressed in the interviews in section 2.2.4.

This approach of using anomaly detection in duplicate detection can therefore be seen as a potential future tool. However, for a more conclusive answer, the method would need to be implemented and tested by developers. This was not done in this research due to time constraints, but the method of using anomaly detection for duplicate detection in an industrial setting on unlabeled data is with current results not presumed impossible.

To further evaluate the results they can be compared to current anomaly detectors evaluated by Zhang et. al. [40]. The maximum reached $RR@k$ over all evaluated models is 0.2-0.45 with $k = 1$, 0.35-0.6 with $k = 3$ and 0.4-0.7 with $k = 5$. The ranges of values show the spread of top $RR@k$ over different systems the bug reports are taken from. However, Android bug reports were not included in this study which does not make it fully comparable. Nevertheless, it gives an insight into reasonable expectations, and the results in this research are not very far off.

4.1.2 RQ2: How do the settings of an LSTM-based log anomaly detection model influence bug report duplicate detection?

As seen in the discussion of the results in section 3, the settings in the LSTM-based anomaly detection model have an impact on the performance of the duplicate detection. Generally, it appears that increased window size h in the model during training and prediction increases the precision when doing duplicate detection with the classification method, i.e. the actual duplicates among the guessed duplicates are increasing. This suggests that a larger window size allows the model to capture more information and thus make better predictions for anomalies, which in turn lets the duplicate detector make better predictions. The trends for the recall are however not as certain, but it has a slight decrease as the window size increases to large values, showing that less of the duplicates are found in the predictions. This might be because the model gets slightly more conservative in guessing and therefore while increasing the precision, also makes less positive guesses. The F1-score follows the recall and is therefore also hard to interpret.

For g, the results do not vary a lot between different top g probability values when using the classification method. There is a slight suggestion that a smaller top g probability, like 7 and 8, during the prediction can increase precision and recall. Since g is the number of log keys of the ranked top possibilities for being the next to appear in a sequence, an increased g decreases the number of sequences flagged as abnormal. The results indicates that as a consequence when g grows, less of the predicted duplicates are correct and less of the actual duplicates are found.

The combined results for h and g in the tables 3.1a-3.1c give a better view of the interaction between g and h. Within the limits of this research, one can see that a smaller g around 7 and 8 and potentially lower gives the best performance in all aspects regardless of the window size. For window size h, the precision increases for higher values and the recall increases for lower, likely because of what was earlier discussed regarding the window size. This highlights the importance of carefully selecting both window size h and top g probability to optimize the anomaly detection performance.

4.1.3 RQ3: What could the application of duplicate bug report detection look like in an industrial setting?

When applying duplicate detection as performed in this research, the ranking method would be most appropriate [40]. This is due to it being more efficient and more fulfilling of needs given the current workflows, that the developer is given k amounts of possible matches to a given bug report. The developer can then look at the logs and the associated bug ticket to determine if any of the bug reports are actual duplicates.

To avoid unnecessary time for analysis when there is no existing duplicate to a bug report, a match ratio threshold can be set to only show top-ranked potential matches if they are over a given threshold. Alternatively, each match ratio can be displayed to the developer who can choose to account for this in their analysis.

For a smooth workflow, the most optimal application would be an automatic script that continuously processes bug reports in recently created bug tickets. For companies like Volvo Cars which are using an instance of Jira, this could be done through the Jira API. For each newly added bug report, an anomaly prediction can be instantly made, followed by a duplicate detection with the ranking method toward all past bug reports. Alternatively, it can be compared to only bug reports captured within a reasonable time interval as proposed by Zhang et. al. [40]. This pipeline can then be hosted and display its results through a server that the developers can access.

Additional steps that need to be taken given the proposed method are that it needs to be updated to be able to handle new classes generated from the log parsing. Furthermore, the anomaly detection model needs to be re-trained sometimes with newly added bug reports to stay up to date with current trends in the system logs. This is important as the Deeplog [10] anomaly detection model assumes that all log sequences it has not encountered before during training are abnormal.

4.2 Limitations and threats to validity

The main limitation of this research was time. Even with sufficient resources the model was time-consuming to train on the full data set due to the amount of classes and the structure of the network. This affected the value combinations of h and g that could be tested as well as other parameters in the anomaly detection model, such as the number of epochs, batch sizes, and adjusted architecture settings such as LSTM size and number of LSTM layers. The time limitation also prevented us from experimenting with other anomaly detection models.

The choice of test data set was the 51 most recent bug reports with assigned duplicates in the Jira system. This was done to simulate a real situation by using new bug reports, and the duplicate criterion was made to ensure enough correct duplicate matches were possible.

However, this poses as a threat to validity as the results may be affected by the data set used for evaluation.

Additionally, the use of unlabeled logs diverts from the guidelines of the used Deeplog [10] model. This approach assumes that all log sequences used for training are normal, which is not true in our case. This is briefly discussed in section 2.3.3. While this should not interfere too much with the anomaly detection given that normal log sequences are more common than abnormal sequences, there is always a risk that this can potentially affect the result.

4.3 Future work

This research has many possibilities for future additional research. Besides further experimentation with settings in the anomaly detection network, this also includes assessing other anomaly detection approaches.

Further, for the application of the researched method in a real industrial setting, one must investigate the handling of new event keys that do not already exist in the vocabulary. Mainly this involves extending the set of output classes in the anomaly detection model, but also the frequency with which the model should be re-trained with new log sequences to stay accurate over time.

Additionally, it would be interesting to investigate the impact of applying an existing duplicate detection method for the parameters and text fields in the bug tickets as described in section 2.1.4, in combination with this approach of using anomaly detection in the system logs.

Chapter 5
Conclusion

In conclusion, the findings of this research show that the proposed approach has the possibility of being applied in an industrial setting. The achieved performance is within the range of good results achieved for other duplicate detection models and data sets [40], indicating that this approach has the potential to achieve state-of-the-art performance with correctly tuned parameters. Proper tuning of the parameters is important as the values for the hyperparameters have an effect on the performance of the duplicate detection and are use-case-dependent.

The difference between this approach and existing ones is that current approaches for bug report duplicate detection mainly imply duplicate detection using the information found in bug tickets, such as product, severity or descriptions [40]. This research therefore provides a new perspective on duplicate detection by utilizing anomaly detection applied on system logs in addition to similar antiquated approaches [35].

Despite the promising findings, it is also essential to acknowledge the limitations and threats to validity of this study. Time constraints impacted the extent of experimentation with parameter values and prevented the exploration of other anomaly detection models. Additionally, the use of an unlabeled log dataset deviates from the Deeplog [10] model guidelines, potentially affecting the validity of our results.

In conclusion, this study contributes valuable insights into the intersection of anomaly detection and duplicate detection, offering opportunities for future research and practical application in industrial settings.

References

[1] A beginners guide to logstash grok. `https://logz.io/blog/logstash-grok`. Accessed: 2024-01-30.

[2] Capture and read bug reports. `https://developer.android.com/studio/debug/bug-report`. Accessed: 2024-01-29.

[3] Read bug reports. `https://source.android.com/docs/core/tests/debug/read-bug-reports`. Accessed: 2024-01-29.

[4] Mehdi Amoui, Nilam Kaushik, Abraham Al-Dabbagh, Ladan Tahvildari, Shimin Li, and Weining Liu. Search-based duplicate defect detection: An industrial experience. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 173–182. IEEE, 2013.

[5] Atlassian. Jira. `https://www.atlassian.com/software/jira`. Accessed: 2024-05-10.

[6] Nicolas Bettenburg, Rahul Premraj, Thomas Zimmermann, and Sunghun Kim. Duplicate bug reports considered harmful... really? In *2008 IEEE International Conference on Software Maintenance*, pages 337–345. IEEE, 2008.

[7] Jayati Deshmukh, KM Annervaz, Sanjay Podder, Shubhashis Sengupta, and Neville Dubash. Towards accurate duplicate bug retrieval using deep learning techniques. In *2017 IEEE International conference on software maintenance and evolution (ICSME)*, pages 115–124. IEEE, 2017.

[8] Bogdan Dit, Latifa Guerrouj, Denys Poshyvanyk, and Giuliano Antoniol. Can better identifier splitting techniques help feature location? In *2011 IEEE 19th International Conference on Program Comprehension*, pages 11–20. IEEE, 2011.

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

[10] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 1285–1298, 2017.

[11] Python Software Foundation. pickle — python object serialization. `https://docs.python.org/3/library/pickle.html`. Accessed: 2024-04-15.

[12] Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *2009 ninth IEEE international conference on data mining*, pages 149–158. IEEE, 2009.

[13] Ana Gainaru, Franck Cappello, Stefan Trausan-Matu, and Bill Kramer. Event log mining tool for large scale hpc systems. In *Euro-Par 2011 Parallel Processing: 17th International Conference, Euro-Par 2011, Bordeaux, France, August 29-September 2, 2011, Proceedings, Part I 17*, pages 52–64. Springer, 2011.

[14] Haixuan Guo, Shuhan Yuan, and Xintao Wu. Logbert: Log anomaly detection via bert. In *2021 international joint conference on neural networks (IJCNN)*, pages 1–8. IEEE, 2021.

[15] Jianjun He, Ling Xu, Meng Yan, Xin Xia, and Yan Lei. Duplicate bug report detection using dual-channel convolutional neural networks. In *Proceedings of the 28th International Conference on Program Comprehension*, pages 117–127, 2020.

[16] Pinjia He, Jieming Zhu, Shilin He, Jian Li, and Michael R Lyu. An evaluation study on log parsing and its use in log mining. In *2016 46th annual IEEE/IFIP international conference on dependable systems and networks (DSN)*, pages 654–661. IEEE, 2016.

[17] Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R Lyu. Drain: An online log parsing approach with fixed depth tree. In *2017 IEEE international conference on web services (ICWS)*, pages 33–40. IEEE, 2017.

[18] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[19] Van-Hoang Le and Hongyu Zhang. Log-based anomaly detection with deep learning: How far are we? In *Proceedings of the 44th international conference on software engineering*, pages 1356–1367, 2022.

[20] LogPAI. Logpai/loghub. `https://github.com/logpai/loghub`. Accessed: 2024-04-04.

[21] LogPAI. Logpai/logparser. `https://github.com/logpai/logparser`. Accessed: 2024-02-21.

[22] Adetokunbo Makanju, A Nur Zincir-Heywood, and Evangelos E Milios. A lightweight algorithm for message type extraction in system application logs. *IEEE Transactions on Knowledge and Data Engineering*, 24(11):1921–1936, 2011.

[23] Leland McInnes and John Healy. Accelerated hierarchical density based clustering. In *2017 IEEE international conference on data mining workshops (ICDMW)*, pages 33–42. IEEE, 2017.

[24] Weibin Meng, Ying Liu, Yichen Zhu, Shenglin Zhang, Dan Pei, Yuqing Liu, Yihao Chen, Ruizhi Zhang, Shimin Tao, Pei Sun, et al. Loganomaly: Unsupervised detection of sequential and quantitative anomalies in unstructured logs. In *IJCAI*, volume 19, pages 4739–4745, 2019.

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

[26] Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[27] Irving Muller Rodrigues, Daniel Aloise, Eraldo Rezende Fernandes, and Michel Dagenais. A soft alignment model for bug deduplication. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 43–53, 2020.

[28] Jennifer Rowley. Conducting research interviews. *Management research review*, 35(3/4):260–271, 2012.

[29] Per Runeson, Magnus Alexandersson, and Oskar Nyholm. Detection of duplicate defect reports using natural language processing. In *29th International Conference on Software Engineering (ICSE'07)*, pages 499–510. IEEE, 2007.

[30] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering*, 14:131–164, 2009.

[31] Kristina Säfsten and Maria Gustavsson. *Forskningsmetodik: för ingenjörer och andra problemlösare*. Studentlitteratur ab, 2019.

[32] Gerard Salton and Chris Buckley. Term weighting approaches in automatic text retrieval. Technical report, Cornell University, 1987.

[33] Chengnian Sun, David Lo, Siau-Cheng Khoo, and Jing Jiang. Towards more accurate retrieval of duplicate bug reports. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 253–262. IEEE, 2011.

[34] Chengnian Sun, David Lo, Xiaoyin Wang, Jing Jiang, and Siau-Cheng Khoo. A discriminative model approach for accurate duplicate bug report retrieval. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 45–54, 2010.

[35] Xiaoyin Wang, Lu Zhang, Tao Xie, John Anvik, and Jiasu Sun. An approach to detecting duplicate bug reports using natural language and execution information. In *Proceedings of the 30th international conference on Software engineering*, pages 461–470, 2008.

[36] wuyifan18. wuyifan18/deeplog. `https://github.com/wuyifan18/DeepLog`. Accessed: 2024-04-24.

[37] Guanping Xiao, Xiaoting Du, Yulei Sui, and Tao Yue. Hindbr: Heterogeneous information network based duplicate bug report prediction. In *2020 IEEE 31st international symposium on software reliability engineering (ISSRE)*, pages 195–206. IEEE, 2020.

[38] Wei Xu. *System problem detection by mining console logs*. University of California, Berkeley, 2010.

[39] Lin Yang, Junjie Chen, Zan Wang, Weijing Wang, Jiajun Jiang, Xuyuan Dong, and Wenbin Zhang. Semi-supervised log-based anomaly detection via probabilistic label estimation. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1448–1460. IEEE, 2021.

[40] Ting Zhang, DongGyun Han, Venkatesh Vinayakarao, Ivana Clairine Irsan, Bowen Xu, Ferdian Thung, David Lo, and Lingxiao Jiang. Duplicate bug report detection: How far are we? *ACM Transactions on Software Engineering and Methodology*, 32(4):1–32, 2023.

[41] Jieming Zhu, Shilin He, Jinyang Liu, Pinjia He, Qi Xie, Zibin Zheng, and Michael R Lyu. Tools and benchmarks for automated log parsing. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 121–130. IEEE, 2019.

Appendices

Appendix A
Interview questions

Introductory questions about background of the participant:

1.1 How long have you worked at Volvo Cars and as a developer?

1.2 What area do you work with?

1.3 How often do you analyze bug reports?

Main interview:

2.1 Current bug report handling:

2.1.1 How often do you discover a bug report to be a duplicate?

2.1.2 How do you know if a bug report is a duplicate?

2.1.3 Where in the logs do you usually start looking when you analyze a bug report?

2.1.4 In a scenario where you suspect that a bug in an issue is triggered by the same thing as for another issue, what is your systematic approach to confirm this by looking at the bug reports?

2.1.5 Are there any specific challenges or difficulties you encounter when identifying duplicate bug reports?

2.1.6 What happens if a bug report or issue is not immediately identified as a duplicate and how often does that happen?

2.2 Bug report handling using machine learning:

2.2.1 If you were to let a machine learning model identify duplicates of bug reports it's helpful to know which parts of the reports that are important. Which parts would you consider important?

2.2.2 The other way around it's also helpful to remove as much unnecessary information as possible to simplify the problem. Which parts of bug reports would you consider removable?

2.2.3 What role do you think machine learning could play in improving the identification and handling of duplicate bug reports, and what potential challenges do you think can arise?

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2024-05-31

EXAMENSARBETE Grouping of duplicate Android bug reports through log anomaly detection
STUDENT Sofia Wahlmark
HANDLEDARE Qunying Song (LTH), Fredrik Henriksson (Volvo Car Corporation)
EXAMINATOR Emelie Engström (LTH)

Automatisk duplikatidentifiering av Android buggrapporter

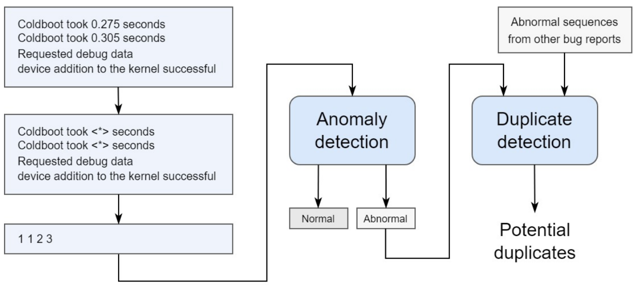
POPULÄRVETENSKAPLIG SAMMANFATTNING **Sofia Wahlmark**

Att lösa buggar är en central del av mjukvaruutveckling. Det händer dock att utvecklare står inför utmaningen att hantera flera buggrapporter för samma problem. Denna studie utforskar en ny metod för att identifiera dubbletter bland Android buggrapporter. Resultaten visar potential för att effektivisera bugghantering.

På Volvo Cars använder man sig av ett Android-baserat system. När fel uppstår tas en så kallad *buggrapport* som innehåller loggar över vad som skett i systemet inom den närmaste tiden. Buggen registreras sedan i ett internt system. Det händer dock ibland att det skapas flera ärenden för samma bugg. I brist på automatisk hantering av dubbletter måste dessa hittas och länkas till varandra manuellt, något som framförallt tar onödig tid från mjukvaruutvecklare.

I detta examensarbete har jag i samarbete med Volvo Cars undersökt hur man automatisk kan hitta dubbletter av Android buggrapporter för att hitta dubletter av buggärenden. De nådda resultaten visar potential för att metoden skulle kunna användas som verktyg för utvecklare för att effektivisera bugghanteringsprocesser. Om man skulle låta modellen förseslå tre buggrapporter som möjliga dubbletter kommer utvecklaren i ungefär hälften av fallen hitta minst en korrekt dubblett bland förslagen. Detta skulle minska arbetsbördan avsevärt jämfört med de tusentals buggrapporter som annars manuellt skulle behöva letas igenom för att hitta eventuella dubletter.

Tidigare metoder fokuserar främst på information i buggärendet, såsom titel och beskrivning. Min metod skiljer sig genom att istället använda sig av avvikande sekvenser i systemloggarna. Loggarna omvandlas först till unika ID:n för sin loggtyp, följt av detektering av avvikande sekvenser. De avvikande sekvenserna jämförs sedan mellan buggrapporter för att föreslå potentiella dubbletter.


```
Coldboot took 0.275 seconds
Coldboot took 0.305 seconds
Requested debug data
device addition to the kernel successful
```
```
Coldboot took <*> seconds
Coldboot took <*> seconds
Requested debug data
device addition to the kernel successful
```
```
1 1 2 3
```

Genom att på detta sättet minska behovet av manuell hantering av dubbletter kan utvecklarna fokusera mer på att lösa nya problem och förbättra användarupplevelsen för sina kunder. Resultaten visar potential för att den föreslagna metoden skulle kunna användas som ett verktyg inom mjukvaruutveckling hos Volvo Cars och liknande företag.