Machine Learning And Deep Learning Based Approaches For Detecting Duplicate Bug Reports With Stack Traces

Neda Ebrahimi Koopaei

A Thesis

In

The Department

Of

Electrical And Computer Engineering

Presented in Partial Fulfillment of the Requirements

For the Degree of

Doctor Of Philosophy (Electrical And Computer Engineering) at

Concordia University

Montreal, Quebec, Canada

July 2019

© Neda Ebrahimi Koopaei, 2019

CONCORDIA UNIVERSITY

SCHOOL OF GRADUATE STUDIES

This is to certify that the thesis prepared

By: Neda Ebrahimi Koopaei

Entitled: Machine Learning and Deep Learning Based Approaches for Detecting Duplicate Bug Reports with Stack Traces

and submitted in partial fulfillment of the requirements for the degree of

Doctor Of Philosophy (Electrical and Computer Engineering)

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

______________________________ Chair
Dr. Catherine Mulligan

______________________________ External Examiner
Dr. Ghizlane El Boussaidi

______________________________ External to Program
Dr. Olga Ormandjieva

______________________________ Examiner
Dr. Anjali Agarwal

______________________________ Examiner
Dr. Yan Liu

______________________________ Thesis Supervisor
Dr. Abdelwahab Hamou-Lhadj

Approved by

Dr. Rastko R. Selmic, Graduate Program Director

September 12, 2019 ______________________________

Dr. Amir Asif, Dean
Gina Cody School of Engineering & Computer Science

Abstract

Machine Learning and Deep Learning Based Approaches for Detecting Duplicate Bug Reports with Stack Traces

Neda Ebrahimi Koopaei, Ph.D. candidate

Concordia University, 2019

Many large software systems rely on bug tracking systems to record the submitted bug reports and to track and manage bugs. Handling bug reports is known to be a challenging task, especially in software organizations with a large client base, which tend to receive a considerable large number of bug reports a day. Fortunately, not all reported bugs are new; many are similar or identical to previously reported bugs, also called duplicate bug reports.

Automatic detection of duplicate bug reports is an important research topic to help reduce the time and effort spent by triaging and development teams on sorting and fixing bugs. This explains the recent increase in attention to this topic as evidenced by the number of tools and algorithms that have been proposed in academia and industry. The objective is to automatically detect duplicate bug reports as soon as they arrive into the system. To do so, existing techniques rely heavily on the nature of bug report data they operate on. This includes both structural information such as OS, product version, time and date of the crash, and stack traces, as well as unstructured information such as bug report summaries and descriptions written in natural language by end users and developers.

In this thesis, we propose new approaches for automatically detecting duplicate bug reports with the objective to help triaging teams and software developers in the provision of fixes. These techniques are based on machine learning and stochastic processes, namely automata, Hidden Markov Models and deep learning algorithms. While the majority of approaches focus on textual parts of bug reports, we use stack traces. The use of stack traces is desirable in situations where bug report descriptions are deemed to be unreliable due to the inherit imprecision and ambiguity of natural language. Moreover, stack traces have the apparent advantage of decreasing the required number of preprocessing tasks, such as those associated with processing bug report comments and descriptions using natural language processing techniques. We evaluate the approaches presented in this thesis by applying them to bug reports of two large open source systems, namely Firefox and GNOME and comparing them to the state-of-the-art approaches.

Acknowledgement

I would like to express my special thanks of gratitude to Dr. Wahab Hamou-Lhadj who gave me the golden opportunity to do this wonderful research by accepting to enroll me as a Ph.D student. He pushed me when I needed to be pushed and complimented me on jobs well done. I thank him for his trust, for keeping his door always open for helpful feedback and conversation. More than anyone else, his influence has contributed to my development as a scientist.

I would like to express my thank to Dr. Abdelaziz Trabelsi for imparting his knowledge and expertise in this study. I am also grateful to thank my committee members for taking the time to read this dissertation and to serve on my thesis committee.

I would like to thank everyone at the Research Lab of Professor Hamou-Lhadj for their friendship, encouragement, and stimulating discussions.

I would also like to thank NSERC (Natural Science and Engineering Research Council of Canada), the Gina School of Engineering and Computer Science, and the Faculty of Graduate Studies for their generous financial support.

I want to thank my lovely parents, brothers and friends who helped me a lot in ups and downs, successes and failures with their encouragements and warm support. None of these could be possible without them.

Table of Contents

List of Figures x

List of Tables xii

Chapter 1. Introduction 1

1.1. Problem and Motivation 1

1.2. Key Terminology 3

1.3. Research Contributions 4

1.4. Thesis Organization 6

1.5. Related Publications 7

Chapter 2. Background and Literature Review 8

2.1. Bug Tracking Systems 8

2.2. On the Use of Stack Traces 11

2.3. Text-based Approaches for Detecting Duplicate Bug Reports 15

2.4. Stack Traces Based Approaches for Detecting Duplicate Bug Reports 18

2.5. Discussion 20

Chapter 3. Dataset Generation 22

3.1. Firefox Dataset 22

3.2. GNOME Dataset 29

Chapter 4. CrashAutomata: Detection of Duplicate Bug Reports based on Generalizable Automata 32

4.1. Introduction 32

4.2. CrashAutomata Approach 33

4.2.1 Training Phase: Automata Construction 34

4.2.2 Testing (Detection) Phase 40

4.2.3 Evaluation Metrics 41

4.3. Evaluation 41

4.4. Comparison with Crash Graphs 48

4.5. Discussion 54

4.6. Chapter Summary 55

Chapter 5. An HMM-Based Approach for Automatic Detection and Classification of Duplicate Bug Reports 57

5.1. Background on HMMs 57

5.2. Overall Approach 60

5.2.1 Training Phase 60

5.2.2 Validation Phase 60

5.2.3 Testing Phase 61

5.2.4 Evaluation Metrics 61

5.3. Evaluation 62

5.3.1 Firefox and GNOME Datasets 62

5.3.2 Comparison 66

5.3.3 Discussion 70
5.5. Chapter Summary 70
Chapter 6. Towards a Deep learning Approach for Detecting Duplicate Bug Reports 72
6.1. Background 72
6.1.1 Recurrent Neural Networks 72
6.1.2 LSTM 75
6.1.3 GRU 76
6.2. Approach 78
6.2.1 Building the Networks 78
6.2.2 Testing Phase 82
6.2.3 Evaluation Metrics 83
6.3. Experimental Results and Comparison 83
6.4. Discussion 88
6.5. Chapter Summary 89
Chapter 7. Conclusions and Future Directions 90
7.1. Summary of Contributions 90
7.2. Threats to Validity and Limitations 91
7.2.1 Threats to External Validity 91
7.2.2 Threats to Internal Validity 92
7.2.3 Threats to Construct Validity 93

7.2.4 Limitations and Opportunities for Future Research 93

7.4. Closing Remarks 95

Bibliography 96

List of Figures

Figure 1.1. Research Contributions 4

Figure 2.1. Life cycle of a bug report in Bugzilla [52] 11

Figure 3.1. An example of Mozilla crash report 24

Figure 3.2. A master bug report and one of its duplicate bug reports in Bugzilla 25

Figure 3.3. Sample converted stack trace to relevant IDs and removing unknown functions...... 28

Figure 3.4. Number of duplicates in duplicate bug report groups in Firefox dataset. 29

Figure 3.5. Sample master bug report in GNOME 30

Figure 3.5. Number of duplicates in a duplicate bug report group in GNOME dataset. 31

Figure 4.1. CrashAutomata Overall Approach 34

Figure 4.2. The n-grams extracted using Algorithm 1 applied to T1, T2, and T3 36

Figure 4.3. The automaton that is extracted from T1, T2, and T3 with $\alpha = 0.6$ 40

Figure 4.4. Average F-measure by varying α for Firefox dataset. 46

Figure 4.5. Average F-measure by varying α for GNOME dataset. 46

Figure 4.6. F-measure boxplot for Firefox 47

Figure 4.7. F-measure boxplot for GNOME 47

Figure 4.8. An example of a graph created by Crash Graphs to model traces [10] 49

Figure 4.9. Boxplot of Precision, Recall and F-measure for all groups in Firefox and GNOME CrashAutomata (α=0.5) and Crash Graphs. 51

Figure 4.10. Average Precision, Recall and F-measure for CrashAutomata and Crash Graphs when applied to the Firefox dataset 52

Figure 4.11. Average Precision, Recall and F-measure for CrashAutomata and Crash Graphs when applied to the GNOME dataset 52

Figure 4.12. Comparison of variation of False Negative Rates for CrashAutomata and Crash Graphs for Firefox dataset 53

Figure 4.13. Comparison of variation of False Negative Rates for CrashAutomata and Crash Graphs for GNOME dataset..... 53

Figure 5.1. Typical topology of an HMM..... 58

Figure 5.2. MAP obtained with different HMM state numbers for Firefox 64

Figure 5.3. MAP obtained with different HMM state numbers for GNOME 66

Figure 5.4. Comparison of Recall Rate@k between Crash Graphs and HMM 68

Figure 5.5. Comparison of MAP between Crash Graphs and HMM..... 69

Figure 6.1. Unrolled recurrent neural network (taken from [87])..... 73

Figure 6.2. Long Short-Term Memory [91]..... 75

Figure 6.3. A Gated Recurrent Unit [91] 77

Figure 6.4. Example of the embedding converting IDs to vectors 78

Figure 6.5. Process of passing information through the layers..... 79

Figure 6.6. The selected network model for Firefox 80

Figure 6.7. Accuracy and cost in Firefox model by number of epochs 85

Figure 6.8. Train vs Test accuracy and cost in GNOME model by number of epochs 86

Figure 6.9. Firefox Recall Rate@k comparison of HMM and our deep NN model..... 87

Figure 6.10. GNOME Recall Rate@k comparison of HMM and deep NN model 87

List of Tables

Table 2.1. An example of stack trace in Mozilla crash reporting system 12

Table 3.1. Characteristics of the Firefox and GNOME datasets.. 27

Table 4.1. The E matrix constructed with CrashAutomata from traces T1, T2, and T3 39

Table 4.2. An example of a classification result of CrashAutomata for nine duplicate bug report groups in Firefox 43

Table 4.3. Classification result of CrashAutomata for duplicate bug report groups in GNOME. 44

Table 5.1. Median of Recall Rate@k with different HMM state numbers using Firefox dataset. 63

Table 5.2. Median of Recall Rate@k with different HMM state numbers using GNOME dataset. 65

Table 6.1. The attributes of the combined model 80

Table 6.2. Results for Recall Rate@k for Firefox and GNOME datasets 83

Chapter 1. Introduction

1.1. Problem and Motivation

Software debugging and maintenance is known to consume up to 70% of the software cost [1]. Many large software systems rely on bug tracking systems to record the submitted bug reports and to track bugs. Developers are then responsible for identifying and fixing bugs usually in a semi-manual way. However, the amount of received bug reports a day might be considerably high [2], [3]. Fortunately, not all reported bugs are new; many are similar or identical to previously reported bugs (duplicates) which can result in work being done multiple times by the development team. Studies on Eclipse and Firefox have shown that approximately 20% of bug reports in Eclipse and 30% of bug reports in Firefox are duplicates [4].

Detecting duplicate bug reports can help the software maintenance process in three ways.First, it reduces the manual triaging effort spent by triagers by preventing them from returning to the same bug multiple times [3], [5]–[7]. In addition, the information inside duplicates can help improve the triaging process such as assigning the bug reports to appropriate developers [7]. Developers can use duplicate bug reports to have a better idea about the bug [7]. There exist tools and algorithms to automatically detect duplicate bug reports as soon as they arrive in the system,;each adapted to the nature of the data available (e.g., [8]–[10]). Stored information inside bug reports plays a significant role in detecting duplicate bug reports.

If we divide the approaches of detecting duplicate bug reports according to the type of data

they use, we can define two general groups. The first group involves approaches that use unstructured textual information inside bug reports such as summaries and descriptions (e.g., [4], [11]–[28]), and the second group encompasses approaches that use stack traces (e.g., [3], [10], [29]–[31]). The former methods consume a vast amount of time preprocessing text, preparing it for natural language processing, which is highly dependent on the quality of the written information by users/developers.

In this thesis, we propose approaches for automatically detecting duplicate bug reports from their early arrival in the system by leveraging solely stack traces. Stack traces carry vital information about the running system when a crash occurs. Using stack traces can have several advantages over other bug report information:

- Crash reporters collect stack traces from user's machine automatically without human intervention, which makes them reliable.
- Stack traces carry information about the running functions when the crash occurs, where usually the bug is located.
- Stack traces can be made available without a written bug report, which does not require the time-consuming process of writing and reading problem descriptions.

This thesis makes the following statement:

We show that stack traces alone are sufficient to detect duplicate bug reports. In fact, using stack traces decreases the required number of preprocessing tasks, such as those associated with processing bug report comments and descriptions using natural language processing techniques. The use of stack traces also addresses the problem of low-quality written bug reports.

1.2. Key Terminology

In this thesis we use the following terms and definitions:

- *Fault/ Defect:* "A fault/defect is introduced into the software as the result of an error. It is an anomaly in the software that may cause it to behave incorrectly, and not according to its specification" [32].

- *Error:* "An error is a mistake, misconception, or misunderstanding on the part of a software developer" [32]. "An error typically is a symptom for the existence of a fault or defect in some artifact" [33].

- *Failure:* "A failure is the inability of a software system or component to perform its required functions within specified performance requirements" [32], [34].

- *Crash:* A crash is when a fault causes a software, application or operation system to crash and exit functioning.

- *Software Bug:* "A software bug is an error, flaw, failure, defect or fault in a computer program or system that causes it to violate at least one of its functional or non-functional requirements". [35]

- *Crash Report:* A crash report stores the software's behavior before crashing. Crash reports are usually reported automatically (crash reporting systems are implemented as part of the software application). A crash report contains data (that can be proprietary) to help developers understand the causes of the crash (e.g., memory dump).

- *Bug report:* A bug report describes a behavior observed in the field and considered

abnormal by the reporter [36]. Bug reports are submitted manually to bug report systems (e.g., Bugzilla/Jira). There is no mandatory format to report a bug. Nevertheless, a bug report should have the version of the software system, OS, and platform, steps to reproduce the bug, screenshots, stack trace and anything that could help a developer assess the internal state of the software system.

1.3. Research Contributions

In this thesis, we make four contributions that are depicted in Figure 1.1, and discussed in more details in what follows.

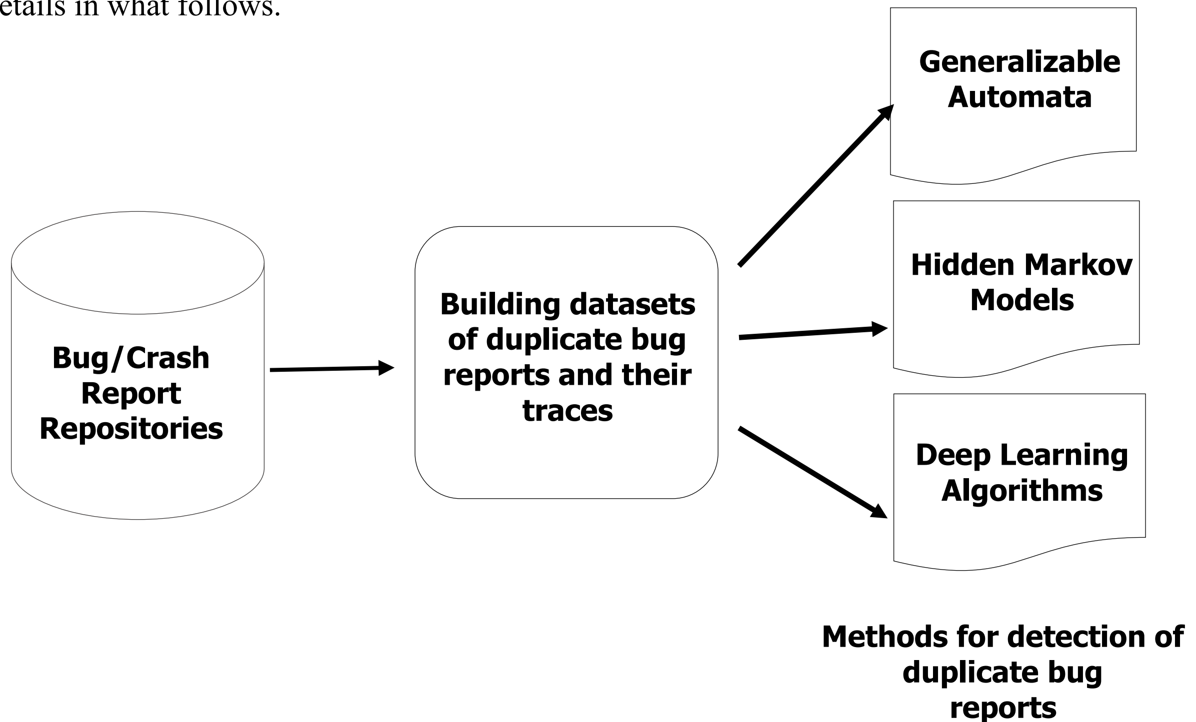


Figure 1.1. Research Contributions

Contribution 1: Dataset Generation

We generate new datasets of duplicate bug reports including master and duplicate bug reports from Firefox and GNOME bug reports (Chapter 3). Building datasets is an important activity in software engineering researchers to allow other researchers and practitioners to experiments with

new algorithms. The main purpose of this contribution is to generate datasets from two large public open bug repositories associated with the Firefox and GNOME projects. The datasets contain stack traces related to duplicate bug reports. These traces are organized in a way that those associated with a bug report including its duplicates are put in one single duplicate group. The resulting groups can be readily use for training, validation, and testing.

Contribution 2: An approach based on generalizable automata for detecting duplicate bug reports

In this thesis, we propose an approach for detecting duplicate bug reports using generalizable automata (Chapter 4). The automaton is a state machine that receives a sequence of inputs and determines if the input should be accepted or not. The contribution of this work is that for each duplicate group, we generate a single automaton that is the representation of stack traces inside a duplicate bug report group. The automaton is designed in a way that it is able to generalize for unseen cases during training by varying a given threshold. We use the generated automata to detect whether a new bug report is a duplicate of existing ones or not. To our knowledge, this the first time generalizable automata are used for detecting duplicate bug reports.

Contribution 3: An Approach based on Hidden Markov Models (HMMs) for Detecting Duplicate Bug Reports

An HMM is a statistical Markov model widely used to analyze the behavior of a system over time [37]. A more straightforward Markov process typically assumes that the states are directly visible to the observation data produced in the system. While in HMM, the states are hidden, but the output of each hidden state (i.e., the state transition probability) is dependent on the observation data. In this contribution (Chapter 5), we use discrete HMMs to provide the state transitions

between function calls of the stack traces. An HMM models the duplicate stack traces of a duplicate bug report group. Therefore, when a new bug report arrives in the system, we compare the associated stack trace with the HMM model of each duplicate group, instead of comparing it with every single existing stack trace, to detect whether it is a duplicate of an existing trace. To the best of our knowledge, this is the first time that HMMs are used in detecting duplicate bug reports.

Contribution 4: Towards a Deep Learning Approach for Detecting Duplicate Bug Reports

In this contribution, we investigate the use of deep learning techniques that are based on neural network models and stack traces to detect and assign duplicates to the relevant duplicate bug report groups (Chapter 6). Deep learning techniques are machine-learning approaches that have proven superior to classical machine learning in image and text processing in recent years. We propose an approach for generating deep neural networks out of the stack traces collected from duplicate bug reports. There exist approaches that apply deep neural networks to textual parts of duplicate bug reports. To our knowledge, this is the first time that deep neural network techniques have been applied to stack traces alone without considering the textual parts.

1.4. Thesis Organization

The thesis organization is as follows; in Chapter 2, we provide background information about the crash reporting and bug tracking systems and present studies related to ours. Chapters 3, 4, 5 and, 6 are dedicated to the main contributions of this thesis we mentioned in the Section 1.3. Finally, we conclude the thesis in Chapter 7, following with future directions and closing remarks.

1.5. Related Publications

- **Neda Ebrahimi**, Abdelaziz Trabelsi, Md Shariful Islam, Abdelwahab Hamou-Lhadj, and Kobra Khanmohammadi. "An HMM-based approach for automatic detection and classification of duplicate bug reports," *Elsevier Journal of Information and Software Technology (IST)* 113 (2019): 98-109. [38]

- **Neda Ebrahimi,** Md Shariful Islam, Abdelwahab Hamou-Lhadj, and Mohammad Hamdaqa. "An effective method for detecting duplicate crash reports using crash traces and hidden Markov models," in *Proceedings of the 26th Annual International Conference on Computer Science and Software Engineering (CASCON 2016)*, pp. 75-84. IBM Corp., 2016. [39]

- **Neda Ebrahimi,** and Abdelwahab Hamou-Lhadj. "CrashAutomata: an approach for the detection of duplicate crash reports based on generalizable automata," in *Proceedings of the 25th Annual International Conference on Computer Science and Software Engineering (CASCON 2015)*, pp. 201-210. IBM Corp., 2015. [40]

Other publication:

- Kobra Khanmohammadi, **Neda Ebrahimi**, Abdelwahab Hamou-Lhadj, Raphaël Khoury**:** " Empirical study of android repackaged applications," *Springer Journal of Empirical Software Engineering*, 2019. [41]

Chapter 2. Background and Literature Review

Mining bug repositories is an area of research that continues to attract increased attention in recent years [42] [43]. This is due to the importance of this field in understanding bugs and their causes, in order to reduce the time and effort in fixing them. The purpose of this chapter is to provide background information needed to understand the rest of this thesis. We also review the literature of mining software repositories research with an emphasis on studies that leverage stack traces for the detection of duplicate bug reports.

2.1. Bug Tracking Systems

Software bugs are inevitable. Software developers manage and keep track of bugs and relevant changes to fix bugs by adopting bug tracking systems such as Bugzilla[1] and Jira[2]. A bug tracking system is a software application that keeps track of reported software bugs in software development projects. Many bug tracking systems have the option for the end users to report the bugs they encounter. Also, developers use bug tracking systems to triage, track, comment and discuss the possible solutions with other developers on the bugs that are reported [44], [45]. A bug tracking system is a necessary component of software development products specifically for large and critical systems.

Information about reported bugs include the time when a bug is reported, its severity, details on how to reproduce the bug (if available), as well as the reporter and developer assigned to fix the bug. Typical bug tracking systems follow the life cycle for a bug and allow for configuring the

[1] https://www.bugzilla.org/

[2] www.atlassian.com/jira

status of a bug upon the changes. The lifecycle of a bug report in Bugzilla, the main bug tracking system used in this thesis (see Chapter 3) is shown in Figure 2.1 [46].This life cycle records the status, resolution, summary and changed date and time of bug reports.

At each stage, the status of a bug changes until a final resolution is found. When a bug is submitted by an end-user, it is set to the UNCONFIRMED status until its presence is confirmed and receives enough votes or a user with proper permissions modifies its status to NEW. It remains in the NEW status until it is ASSIGNED to a developer to be fixed. When a fix is determined by the assignee, the report moves to the RESOLVED state. Developers have five different options to resolve a report. The status can be one of FIXED, DUPLICATE, WONTFIX, WORKSFORME, and INVALID [47].

- RESOLVED FIXED: If a developer modifies the source code and commits the fixes to the system.
- RESOLVED DUPLICATE: A previously submitted report is detected. The report is marked as a duplicate of the original report.
- RESOLVED WONTFIX: When developers decide that there is no proper solution for this bug and a given report will not be fixed.
- RESOLVED WORKSFORME: If after some attempts by the developers, it is not possible to reproduce the bug according to the problem descriptions in the report.
- RESOLVED INVALID: If the report is not related to the software itself.
- RESOLVED REOPEN: A bug is reopened if the tester is not satisfied with the fix or a

formerly resolved report may be reopened at a later date (e.g., due to an ineffective fix).

- In such cases, the status of the bug report changes to REOPENED [4], [5], [48]–[50].
- RESOLVED VERIFIED: When a tester verifies the fixes.

Finally, the report is CLOSED after it is resolved and does not occur again. A report can be reopened (sent to the REOPENED state) if the initial fix is not adequate and the tester is not satisfied with the fix or a formerly resolved report may be reopened at a later date (e.g., due to an ineffective fix). Reports can be reopened multiple times.

The fixing time is computed as the number of days between the date when bug report is marked as new and the date when it is marked as a resolved. During the life cycle of a bug report, developers have the option to add extra information about the bug report when analyzing it. Other developers may use this information for fixing this bug or other relevant bugs. Also, Status and Resolution fields of the bug report may be updated depending on the changes developers make.

Software developers keep track of the severity of the bug report as well. The severity implies the level of impact of the bug on the software execution and is a critical factor in detecting the priority of a bug for fixing [51]. The possible severities are:

- Blocker: blocks development and/or testing work.
- Critical: crashes, loss of data, severe memory leak. Software will not run.
- Major: major loss of function.
- Normal: regular report, some loss of functionality under specific circumstances.

Figure 2.1. Life cycle of a bug report in Bugzilla [52]

2.2. On the Use of Stack Traces

In this thesis, we use stack traces to detect duplicate bug reports. A stack trace contains memory dump of function calls when the system crashes. An example of a stack trace in shown in Table 2.1. The trace contains five frames; each containing information about the executed function. A frame signature is defined as the method signature, which is composed of both the function and the module name. In this example, the program crashed at Frame 0 (function signature mozglue.dll/mozalloc_abort). More formally, a stack trace can be defined as an ordered set of

frames $F=\{f_0, f_1, \ldots, f_n\}$, where n is the number of functions and f_0 is the top frame.

Table 2.1. An example of stack trace in Mozilla crash reporting system

Frame	Module	Signature
0	mozglue.dll	mozalloc_abort(char const* const)
1	mozglue.dll	mozalloc_handle_oom(unsigned int)
2	mozglue.dll	moz_xmalloc
3	xul.dll	mozilla::net::CacheFileMetadata::WriteMetadata(unsigned int, mozilla::net::CacheFileMetadataListener*)
4	xul.dll	mozilla::net::CacheFile::WriteMetadataIfNeededLocked(bool)
5	xul.dll	mozilla::net::CacheFile::DeactivateChunk(mozilla::net::CacheFileChunk*)

Stack traces have been used in various areas of software maintenance including bug prioritization [53]–[55], fault location, bug reproduction [56], etc. Schröter et al. [57] conducted an empirical study on the usefulness of stack traces for bug fixing. They showed that the lifetime of a bug containing stack traces is shorter that that of a bug report without a stack. In addition, they showed that in 60% of the fixed bugs they analyzed, the bugs were located inside functions found in stack traces.

Just et al. [58] surveyed the required information and common problems developers tend to

face when bug reporting systems. They argued that there is a long way to go before bug tracking systems reach the expected performance. Among their findings, the authors the most helpful information that developers need in order to fix a bug consists of the bug reproduction steps, which are usually easy to construct from stack traces and test cases.

Kim et al. [59] argued that most bug reporting systems prioritize their bugs based on the number of included crash reports. It might take a long time for a top crash to collect enough submitted crash reports, which also may cause some difficulties for users in case of data loss. The authors proposed a technique to predict top crashes earlier. The technique first collects stack traces stored in top and bottom crash reports. Then, they extract methods from stack traces. A feature selection method records the feature value for each trace. This information is used in training a machine learner, which is used to predict if a crash is a top or a bottom crash. Although the method is able to automatically predict if a new crash report would be a top or bottom, the approach has many limitations. If a top crash has never occurred in the training set, the technique is not applicable.Also, at least two versions of a product should be available for feature selection. By applying the approach on the crash databases of Firefox and Thunderbird, the authors' approach achevieda prediction accuracy of 75-90% on a small dataset.

Wang et al. [3] investigated the possibility of identifying correlated crash types using stack traces. The authors performed an empirical study on open source bug repositories, Firefox and Eclipse. They defined *"a group of crash types related to identical or correlated bugs, as a crash correlation group (CCG)"* [3]. A crash type can belong to one or several crash correlation groups. For example, if a crash type CT1 shares a bug with a crash type CT2 and another bug with a crash type CT3 then CT1 belongs to two crash correlation groups, i.e., {CT1,CT2} and {CT1,CT3}. According to the investigated correlations, the authors proposed a bug localization method called

Buggy Files Finder (BFFinder) to locate and rank buggy files from the stack traces in crash reports. BFFinder uses three rules to identify correlated crash types: Rule 1: crash signature comparison, Rule 2: top frame comparison and Rule 3: closed ordered sub-set comparison. Using a Bayesian Belief Networks, BFFinder computes and ranks files from stack traces based on their probability of being buggy.

CraTer [60] is a tool introduced by Gu et al. [50] to automatically predict if a crashing fault resides in stack traces or not. The authors extracted 89 features from the stack traces and source code (such as type of the exception, the number of frames, and features from the top and bottom frames) to train their model. By leveraging the predictive model, CraTer can reduce the search space for crashing faults and help prioritize crash localization efforts. CraTer achieves high accuracy (over 90%) when predicting the residence of newly-submitted crashes.

Stack traces have also been found useful in bug reproduction. The studies that fit in this category can be divided into two distinct parts: (A) on field record and in-house replay [61]–[63] and (B) on house crash explanation [31], [56], [64]. These two categories yield heterogeneous results depending on the chosen approach and are mainly differentiated by the needs of instrumentation. Indeed, the first category needs to oversee – by means of instrumentation – the execution of the targeted system on the field in order to reproduce the crashes in-house, while tools and approaches belonging to the second category only use materials produced by the crash such as the crash stack trace or the core dump at crash time. In the first category, tools record a different type of evidence such as invoked methods [62], try-catch exception [65] or objects [66]. In the second category, the tools and approaches aim to understand what happens only using material produced by the crash itself, such as crash stack [31], previous – and controlled – execution [64] or model checking [56].

2.3. Text-based Approaches for Detecting Duplicate Bug Reports

Several approaches have been proposed in the literature to support the automatic detection of duplicate bug reports. This section presents previous studies related to our work. We divide the duplicate detection approaches into two main categories based on the type of bug report information used: 1) Text-based approaches and 2) Execution information-based approaches.

Typically, developers and users submit information related to the crash in the summary and textual description section of a bug report. The aforementioned unstructured information can then be used by researchers to investigate the similarities between existing bug reports and an incoming bug report for classification purposes [3], [9], [11], [12], [15], [24]–[27], [67]–[71]. Using natural language sections, information retrieval (IR) techniques are widely used to calculate the similarity scores between queries and the retrieved data. In this context, Wang et al. [69] studied similar terms in duplicate bug reports of OpenOffice for text similarity measurements. Then, they proposed a new technique to extract similar terms in bug report descriptions that cannot be obtained via a general-purpose thesaurus. They showed that their method improves upon existing methods by 58%. Rakha et al. [24] conducted a study on the effort required to manually identify duplicate issue reports for Firefox, SeaMonkey, Bugzilla, and Eclipse-Platform. They observed that more than 50% of duplicate bug reports can be detected within 24 hours after their submission even when only one developer is involved. Their classification model achieves an average precision and recall of 68% and 60% respectively.

The performance of different IR models (e.g., Log-Entropy based weighting systems) compared with topic-based modes (e.g., LSI, LDA, and Random Projections) has been studied by Kaushik et al. [67]. By applying different heuristics to data retrieved from Eclipse and Firefox,

they observed that word-based models outperform topic-based models with 60% and 58% recall rates, respectively. Their results suggest that the project's domain and characteristics play a crucial role in improving the performance of heuristic models.

Sun et al. [44] proposed a supervised approach based on a discriminative model. The model uses information retrieval to extract textual features from both duplicate and non-duplicate bug reports. It is then trained and tested using a support vector machine (SVM) classifier. All pairs of duplicate bug reports have been formed and considered as positive samples, while all other pairs of non-duplicate bug reports have been treated as negative ones. By applying the method on bug reports from Firefox, Eclipse, and OpenOffice, the authors' method was able to achieve a recall as high as 65% on all datasets.

In another study [27], Nguyen et al.improved their model by extending the well-known BM25 ranker to provide a ranked list of duplicates. It should be noted that BM25 is a function for calculating term frequencies and similarities among bug reports. By taking advantage of IR-based features and topic-based features, Nguyen et al. extended the work of Sun et al. [38] by combining BM25F with a specialized topic model. The authors considered words and term occurrences inside bug reports in an effort to improve bug localization performance. The algorithm shows recall rates between 37% and 71% and Mean Average Precision of 47%.

Sureka and Jalote [26] proposed a character-level n-gram approach to further improve the accuracy of automatic duplicate-bug-report detection. The technique calculates the text similarity between the user's query and existing title and description information of bug reports in character-level. The character-level n-grams are language independent and thus, they save languages specific pre-processing time. According to their experiments, however, their approach has been of modest

performance for which about 21% and 34% recall rates have been achieved for top 10 and top 50 recommendations, respectively.

Another set of supervised approaches build a model based on a training data and use it to analyze a pair of bug reports to predict whether they are duplicate. In addition to textual and categorical features (description, component, priority, etc.), Alipour et al. [11] suggested using contextual features to detect duplicate bug report pairs. They showed that domain knowledge of software engineering concept plays a compelling role in detecting duplicates between bug report pairs. When applied to a bug repository of the Android ecosystem, the approach achieves a recall of up to 92%.

Deshmukh et al. [21] applied Convolutional Neural Networks (CNN), and Long Short Term Memory (LSTM) on short and long descriptions of bug reports extracted from the dataset presented by Lazar et al. [72]. They showed that their approach could achieve accuracy and recall rate of 90% and 80%, respectively.

Budhiraja et al. [22] applied word embedding to the summary and descriptions of bug reports. After tokenizing, training and vectorization of bug reports, all the bug reports in the dataset are converted into vectors. For a new bug report, a similar process is performed to convert it to a vector. Then the similarity between the vector of the new bug report and all the reports is calculated. The similarity between new and existing bug reports is reported using the top k rank metric. They concluded that word embedding has potential in detecting duplicate bug reports.

The study by Poddar et al. [17] classifies duplicates by leveraging a neural network and learning latent topics (vectors) and cluster them into latent topics. The method measures the topic similarity modeling (semantic space) between the topics of two bug reports;if both are talking

about a similar topic their θ must be close to one another. With this loss calculation, the method learns the duplicates from the same topic and non-duplicates from different topics. The final component focuses on the important words by creating memory vectors ϕ. The approach achieves a Recall between 76% and 96% in used datasets. Comparing the obtained results on different types of datasets with non-technical annotators indicates that their approach outperforms the state-of-the-arts techniques.

Chaparro et al. [18] argued the reformulating queries for retrieval of duplicate bug reports. They include the software's observed behavior in bug reports and the bug report title to reformulate the initial query. The authors hypothesized that these selected features contain more bug relevant information in the queries and is able to retrieve more duplicate bug reports and improve the quality of duplicate bug report retrieval between 56% and 78%.

Continuous querying bug reports in an approach that was introduced by Hindle et al. [19], and which alerts bug reporters about the possibility of the existence of duplicate bug reports as they type in their bug report. The main goal of this study is to use IR-based measures to deduplicate the bug reports as they are reported rather than reformulating the queries to find the duplicate bug reports afterwards. By leveraging TF-IDF and cosine distance to find similar documents, Hindle et al. [19] could improve the performance of the state-of-the-art duplicate bug report detection techniques.

2.4. Stack Traces Based Approaches for Detecting Duplicate Bug Reports

Wang et al. [71] applied natural language processing techniques to both stack traces and bug report descriptions and observed that there is an improvement of 25% over approaches that only use bug report descriptions. The authors, however, did not model the temporal order of sequence

of calls in stack traces. Instead, they treated stack traces as text with stack trace functions treated as words. This approach detects 67%-93% of duplicate bug reports of Firefox.

Lerch et al. [30] proposed an approach to identify stack traces in bug reports by transforming stack traces into a set of methods and then using term frequency to compute and rank the similarity between method sets. The authors' method, when applied to Eclipse bug reports, achieves the same results as state-of-the-art approaches, but with fewer requirements. This approach, however, does not consider the temporal order of sequences of function calls in stack traces.

Kim et al. [10] proposed a crash graph-based model which captures the crashes reported and stored in a bucket. A graph of stack traces in a bucket (a group of related bug reports) is constructed to aggregate multiple traces. Instead of comparing an upcoming stack trace with every single trace in a bucket, their model only compares with the graph. To evaluate their model, the authors used graph similarity as a metric. When applied to crash reports of Windows systems, their approach achieves a maximum precision of 71.5% and recall of 64.2%. To our knowledge, this is the only approach that uses temporal order of sequences of functions calls of stack traces to detect duplicates. Our approach achieves a better recall rate and MAP than Kim et al.'s approach as discussed in Section 4.4.

To improve the accuracy of bucketing in the Windows Error Reporting system (WER), Rebucket was proposed by Dang et al. [73] for clustering crash reports based on call stack similarity. Rebucket measures the similarity between call stacks in WER and assigns crash reports to buckets according to similarity values. This approach is not used to detect duplicates, but to group related crashes together.

Sabor et al. [29] used package names in stack traces instead of method names to detect

duplicate bug reports in Eclipse. Their method then generates n-gram features from sequences of package names. The extracted features are then used for measuring the similarity between new stack traces of new bug reports and stack traces of historical bug reports. The objective of their paper is to reduce the computation time needed to process large traces.

Castelluccio et al. [74] proposed a tool, integrated into Socorro to find statistically significant properties in groups of Mozilla crash reports and present them to analysts (developers and triaging teams) to help them analyze the crashes and understand the causes. The tool is based on contrast-set learning, a data mining approach [75]. The authors applied the tool to data collected from the Mozilla crash reporting system and bug tracking system. Their findings show that the tool is very effective in analyzing related crash groups and bugs.

Furthermore, the tool, which is now integrated with the Mozilla crash reporting system, received favorable feedback from Mozilla developers. Although this tool does not tackle the problem of duplicate bug reports, it could be useful in our research. We can use it to extract the most meaningful properties of crash traces and use them to improve the HMM models. The tool can also be used to improve the grouping of crashes in Mozilla, and use the resulting grouping to identify crash reports that are related to duplicate bug reports.

2.5. Discussion

As described in the previous sections, the majority of studies focus on detecting duplicate bug reports using the textual parts of bug reports such as summaries and descriptions. Most of the techniques that use stack traces treat their content as document and leverage natural language processing techniques. These techniques do not take advantage of the temporal order of function calls in stack traces, which is the main feature that characterizes the execution of a system.

We conjecture in this thesis that using the full potential of stack traces (i.e., by considering the execution paths depicted by the function calls) are an excellent alternative to bug report descriptions for the detection of duplicate bug report, especially in cases where the quality of bug report descriptions and comments is deemed to be poor. To our knowledge, the only technique that truly leverages stack traces for the problem of duplicate bug report detection is the one proposed by Kim et al. [10] by leveraging graph theory techniques. This motivated us to compare the techniques proposed in this thesis with Kim et al.'s approach.

Chapter 3. Dataset Generation

In this chapter, we describe the datasets we used in our research. We also describe the details, process, and characteristics of the datasets that are collected.

To test our methods, we need datasets that contain a decent number of duplicate bug reports with their associated stack traces. Once we have these groups, we can split the data into training, validation, and testing steps, to run experiments with machine learning techniques. The datasets should also be publicly available so as to allow other researchers to reproduce the proposed studies as well as to compare their approaches with ours.

Keeping these criteria in mind, we chose to conduct our studies on bug reports of two open source datasets: Firefox and GNOME. These two projects have been widely used in similar studies (e.g., [2], [4], [6], [13], [23], [71], [76]–[79]). Also, both projects rely on Bugzilla's bug tracking system for managing bug reports, which reduces the time spent on parsing and processing bug reports from various bug tracking systems.

3.1. Firefox Dataset

To generate the Firefox dataset, we have to refer to two related resources: Mozilla Crash Reporting system (also called Socorro)[3] and Bugzilla Mozilla[4]. Mozilla Socorro manages the Firefox crash reports, while Bugzilla keeps track of bug reports.

[3] https://crash-stats.mozilla.com/

[4] https://bugzilla.mozilla.org/

When a Mozilla system (such as Firefox) crashes, a dialog box appears to the user enabling him or her to provide comments about the observed failure. Mozilla Socorro also records additional information about the current status of a user's machine including the stack trace, priority, product, component, OS version, and date of the underlying system [74]. An example of a partial screen shot of a crash report is shown in Figure 3.1. In this example, crash report ID `2c8e0719-c725-4310-882e-59e700190707` classified as `OOM|small` signature reported on 2019-07-06. This crash belongs to the Firefox version 69.0b2 and was running on a Windows 7. This crash report contains more than 60 crashing threads. However, for the ease of presentation, we only show the top three frames of the crashing thread 0 with module names, signature, and related source file links. The source file may link to the original code where the signature is located. A crash report might have several other features than what presented here that are out of scope of our research.

When a crash report is submitted to Mozilla crash report system, Socorro first groups it according to the top method signature of the associated stack trace, and other existing grouping algorithms [74]. The resulting groups are called "crash buckets", and their purpose is to group all similar crash reports together so that when one case is fixed, the solution will apply to the rest of the group as well.

It should be noted that Socorro may result in many duplicate reports being spread over multiple buckets. This happens when the top frames of stack traces are different. Even worse, many unrelated crashes may be grouped together, which defeats the purpose of having a bucketing system in the first place. This faulty bucketing process can cause further difficulties for developers when attempting to understand the problem by looking at crash traces that may, in fact, be unrelated.

ID: 2c8e0719-c725-4310-882e-59e700190707
Signature: OOM | small

Signature	OOM | small
UUID	2c8e0719-c725-4310-882e-59e700190707
Date Processed	2019-07-07 19:10:05 UTC
Uptime	21,684 seconds (6 hours, 1 minute and 24 seconds)
Last Crash	52,307 seconds before submission (14 hours, 31 minutes and 47 seconds)
Install Age	116,061 seconds since version was first installed (1 day, 8 hours and 14 minutes)
Install Time	2019-07-06 10:46:06
Product	Firefox
Version	69.0b2
Build ID	20190704152356 (2019-07-04)
OS	Windows 7
OS Version	6.1.7601 Service Pack 1
Build Architecture	x86
CPU Info	GenuineIntel family 6 model 37 stepping 2
CPU Count	4

Crashing Thread (0)

Frame	Module	Signature	Source	Trust
0	mozglue.dll	mozalloc_abort	memory/mozalloc/mozalloc_abort.cpp:33	context
1	mozglue.dll	mozalloc_handle_oom(unsigned int)	memory/mozalloc/mozalloc_oom.cpp:51	cfi
2	mozglue.dll	moz_xmalloc	memory/mozalloc/cxxalloc.h:33	cfi

Figure 3.1. An example of Mozilla crash report

When the number of crash reports in a bucket reaches a threshold, the triaging team creates a bug report, assigns to it a bug report ID, and submits it to Bugzilla for further investigation. This point marks the beginning of a bug report in Bugzilla.

Bug 1062929

While idling on Command and Conquer:Tiberium Alliances, Firefox crashes with OOM small

RESOLVED DUPLICATE of ~~bug 1053934~~

Status

Product: Firefox
Component: Untriaged
Type: defect
Status: RESOLVED DUPLICATE of ~~bug 1053934~~
Opened: 5 years ago
Updated: 5 years ago

People (Reporter: bytehead, Unassigned)

Tracking

Version: 32 Branch
Target: ---
Platform: x86 Windows 7
Points: ---

Firefox Tracking Flags (Not tracked)

Details

Whiteboard: ---
Votes: 0 votes
Crash Signature: [@ OOM | small]

Bug 1053934

Crash in gfxContext::PushNewDT (OOM) with reproducible test-case (reddit.com)

VERIFIED FIXED in Firefox 33

Status

Product: Core
Component: Graphics
Type: defect
Importance: -- critical
Status: VERIFIED FIXED
Opened: 5 years ago
Updated: 4 years ago

People (Reporter: aaronmt, Assigned: mattwoodrow)

Tracking

Version: Trunk
Target: mozilla34
Platform: ARM Android
Keywords: crash, reproducible
Points: ---
Blocks: 1060856, ~~Woodduck~~
Dependency tree / graph
Duplicates: ~~1058567~~, ~~1058661~~, ~~1059346~~, ~~1059762~~, ~~1060581~~, ~~1062929~~

Firefox Tracking Flags (firefox32 wontfix, firefox33+ fixed, firefox34 fixed, b2g-v2.0 verified, b2g-v2.0M verified, b2g-v2.1 verified, fennec+)

Details

Whiteboard: ---
Votes: 0 votes
QA Whiteboard: [QAnalyst-Triage+]
Crash Signature: [@ OOM | small]
URL: reddit.com/r/askscience/comments/28va...

Figure 3.2. A master bug report and one of its duplicate bug reports in Bugzilla

We implemented a web crawler to look for available bug reports from bug report ID 1 to bug report ID 1,299,999 from the Bugzilla Mozilla website. We then stored the bug reports in XML format. To extract appropriate bug report information, we simply parse the XML files. Firefox offers various products, including all the Mozilla products used in our data collection. These products highlight issues with Firefox web pages and other Mozilla software. The status of a bug report is either NEW, INVALID, RESOLVED, VERIFIED, and etc. and is set by the developer. Among all available bug reports, we selected the bug reports with "RESOLVED DUPLICATE" or "VERIFIED DUPLICATE" status.

Figure 3.2 shows the available information in a bug report page with the links to the duplicates (if applicable) and crash signature(s). A Master Bug Report (MBR) is a bug report that is considered to be a parent for upcoming duplicate bug reports. Therefore, if a bug report is detected similar or identical to the MBR, the bug is called the duplicate of the MBR.

Each bug report may or may not have links to the crash signature, its group of crash reports, and the stack traces inside the crash reports identified in Mozilla. In our case, we have only considered bug reports with a valid link to the crash signatures that also contain stack traces. It is worth noting that Mozilla's crash reporting system preserves crash reports from one year before the current date. Therefore, any signature no longer containing any crash reports was removed from our dataset. In Mozilla, crash reports may have than one crashing thread, starting from Thread 0, which is the main thread. We have implemented a web crawler to retrieve all available threads in a crash report. We only considered Thread 0 for the current study. Considering more threads may yield scalability issues when training with a large set of bug reports. Moreover, we used a maximum of 200 stack traces associated with each crash signature. We stopped at 200 because we noticed that adding more traces did not affect much the results.

Stack traces of an MBR and its duplicates form a duplicate group with the MBR's ID as its label. A duplicate group contains a group of stack traces of an MBR and at least one duplicate. For example, in Figure 3.2, bug ID "1053934" is an MBR with seven duplicates: "1058567", "1058661", "1059346", "1059762", "1060581", and "1062929". Therefore, we generated a duplicate group labelled "1053934", which contains stack traces of MBR and all its duplicates.

We also eliminated repetitive stack traces from each duplicate group. Since we need at least two stack traces for training, one for validation, and one for testing, we selected duplicate groups with at least four stack traces. After filtering for these conditions, we collected 103 duplicate bug report groups (see

Table **3.1**). While processing the traces, we noticed that some function names start with @0x, which seem to be memory addresses. We removed these functions from the traces since we cannot ascertain that they belong to Firefox. They could be caused by Windwos configuration system.

Finally, we convert the function names in stack traces to function IDs to speed up processing. For example, the stack trace in Figure 3.3 contains 7 frames (F0- F6). However, F6 is an invalid function – calls a memory address - that will not convert to a valid function ID. At the end, this stack trace is converted into the sequences of numbers: 1 2 3 4 5 6. This process is done for every single stack trace in our dataset and identical function IDs are given the same functions.

Table 3.1. Characteristics of the Firefox and GNOME datasets

Dataset	**Total number of duplicate bug report groups**	**Total number of traces**	**Average #of stack traces in each DG**
Firefox	103	2,883	28

GNOME	182	4,600	25

Frame	Stack Frames	Function ID
F0	nsLineLayout::ReflowFrame(nsIFrame*, unsigned int&, mozilla::ReflowOutput*, bool&)	1
F1	nsBlockFrame::ReflowInlineFrame(mozilla::BlockReflowInput&, nsLineLayout&, nsLineList_iterator, nsIFrame*, LineReflowStatus*)	2
F2	nsBlockFrame::DoReflowInlineFrames(mozilla::BlockReflowInput&, nsLineLayout&, nsLineList_iterator, nsFlowAreaRect&, int&, nsFloatManager::SavedState*, bool*, LineReflowStatus*, bool)	3
F3	nsBlockFrame::ReflowInlineFrames(mozilla::BlockReflowInput&, nsLineList_iterator, bool*)	4
F4	nsBlockFrame::ReflowLine(mozilla::BlockReflowInput&, nsLineList_iterator, bool*)	5
F5	nsBlockFrame::DrainSelfOverflowList()	6
F6	@0xbb3afff	

Figure 3.3. Sample converted stack trace to relevant IDs and removing unknown functions.

Figure 3.4 shows the distribution of available duplicates in the generated duplicate bug report groups. We can see that 78 out of 103 duplicate groups have only two duplicate bug reports available. The number of duplicate bug reports in each duplicate bug report group in our Firefox

dataset does not exceed 8.

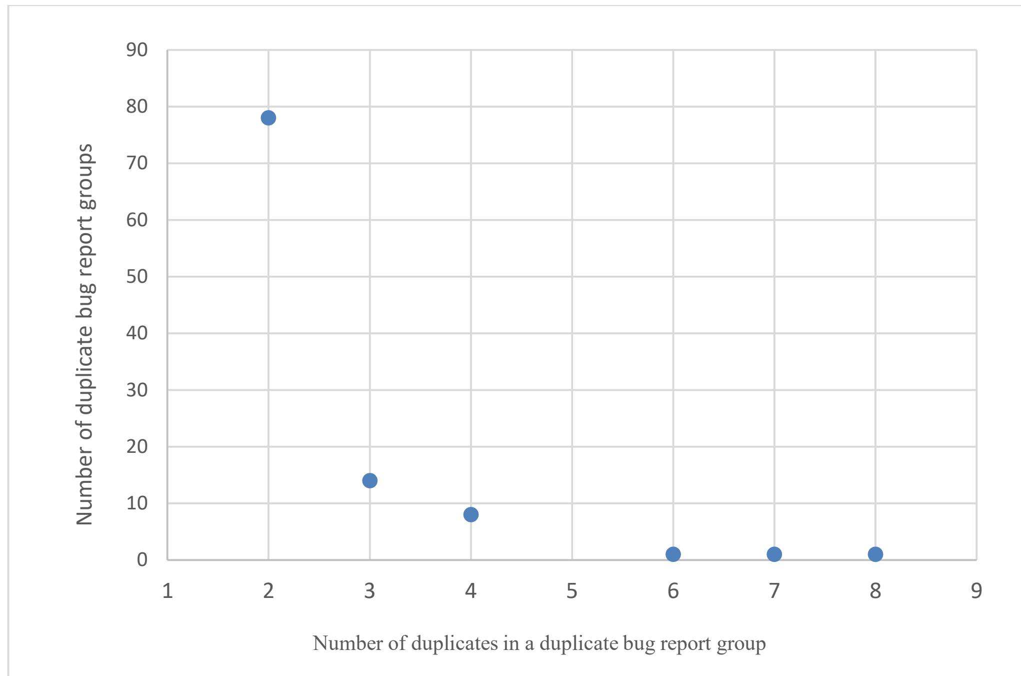


Figure 3.4. Number of duplicates in duplicate bug report groups in Firefox dataset.

3.2. GNOME Dataset

Developed by the GNOME project, GNOME is a graphic user interface and a set of desktop applications on Linux. To track bug reports, GNOME also uses the Bugzilla bug tracker[5]. Unlike Mozilla, GNOME stack traces are integrated into the bug report description parts. Figure 3.5 shows a sample bug report ID “69271” with duplicates and stack traces. To collect the bug report data, we implemented a web crawler to retrieve all related GNOME bug reports from the Bugzilla bug repository.

[5] https://bugzilla.gnome.org/

GNOME New Browse Search Search [?] Reports Help New Account Log In Forgot Password

Bug 69271 - Embossing image crashes background capplet [gdk_pixbuf_finalize]

Status: RESOLVED FIXED

Product: gnome-control-center
Component: Background
Version: unspecified
Hardware: Other other

Importance: High critical
Target Milestone: ---
Assigned To: Control-Center Maintainers
QA Contact: Control-Center Maintainers

URL:
Whiteboard:
Keywords:

Duplicates: 62460 63159 66773 69221 71151 71776 75051 75158 75361 80135 83248 90121 97563 101109 (view as bug list)
Depends on:
Blocks:
Show dependency tree

Reported: 2002-01-21 13:45 UTC by scott
Modified: 2004-12-22 21:47 UTC (History)
CC List: 1 user (show)

See Also:
GNOME target: ---
GNOME version: 2.0

Attachments
Add an attachment (proposed patch, testcase, etc.)

scott 2002-01-21 13:45:47 UTC Description

```
Package: control-center
Severity: normal
Version: 1.4.0.1
Synopsis: Embossing a particular image crashes
Bugzilla-Product: control-center
Bugzilla-Component: background

Description:
Every time I try embossing this image, it crashes the applet.

Hopefully, I can attach the image. If not, give me an e-mail and I
forward it.
(no debugging symbols found)...(no debugging symbols found)...
(no debugging symbols found)...0x4065b079 in wait4 () from /lib/l:
```

+ Trace 31649

Trace 16541 (Quality: 2.0) on Bug 69271

```
#0 0x406ab8f9 in wait4 () from /lib/libc.so.6
#1 0x407241f0 in __check_rhosts_file () from /lib/libc.so.6
#2 0x40136286 in gnome_init () from /usr/lib/libgnomeui.so.32
#3 0x40634848 in sigaction () from /lib/libc.so.6
#4 0x40679ac3 in free () from /lib/libc.so.6
#5 0x405f7226 in gdk_pixbuf_finalize () from /usr/lib/libgdk_pixbuf.so.2
#6 0x405f71f2 in gdk_pixbuf_finalize () from /usr/lib/libgdk_pixbuf.so.2
#7 0x405f711b in gdk_pixbuf_unref () from /usr/lib/libgdk_pixbuf.so.2
#8 0x0804d136 in copyState ()
```

Figure 3.5. Sample master bug report in GNOME

By December 2016, we collected all available bug reports starting from bug ID 1 to bug ID 753,300. Then, a parser was used to extract bug reports with the label “RESOLVED DUPLICATE”. Like Firefox, duplicate groups are formed by gathering and linking the stack traces of master and duplicate bug reports into a single duplicate group. To keep the valid method names, a preprocessing has been performed on stack traces to remove unknown and meaningless methods generated by either calls to Java libraries or setting up of debugging parameters.

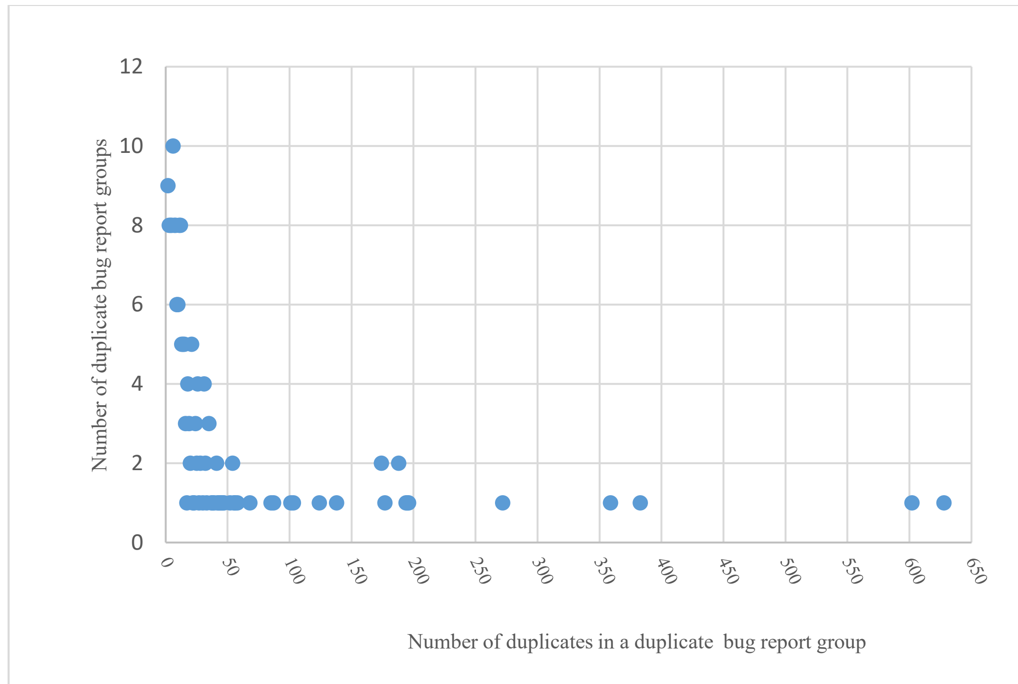


Figure 3.6. Number of duplicates in a duplicate bug report group in GNOME dataset.

Similar to Firefox, we only kept the duplicate groups that included more than two duplicate bug IDs and at least four stack traces. We found 675 duplicate groups and thus, to reduce the computation time, we selected randomly 182 duplicate groups for further analysis. Figure 3.6 and

Table **3.1** illustrate the attributes of the generated GNOME dataset. A wide range of duplicates ranging from 2 to 628 is available in each duplicate group. As can be observed in Figure 3.6, 156 of the 182 (about 86%) duplicate groups contain between 2 and 50 duplicates. Since GNOME preserves stack traces for a longer time compared to Firefox datasets, GNOME datasets are more diversified than Firefox in terms of the number of duplicates. There are even some duplicate groups with over 200 duplicate bug reports.

Chapter 4. CrashAutomata: Detection of Duplicate Bug Reports based on Generalizable Automata

4.1. Introduction

As mentioned in Chapter 2, existing techniques for detecting duplicate bug reports can be divided into two categories. The first category encompasses techniques that rely solely on the description of the reports, expressed in natural language [6], [15], [23], [26], [27], [44], [68]. These techniques suffer from the imprecision and ambiguity of natural language, making them unreliable when working with poor-quality descriptions. To address this, researchers have turned to more formal bug report data, such as stack traces [10], [71], [73], [76]. These studies model information in historical stack traces and use the resulting models to classify incoming reports.

Perhaps one of the most effective approaches is Crash Graphs. Introduced by Kim et al. [10], Crash Graphs are used to detect duplicate bug (and crash) reports in WER (Windows Error Reporting System). The approach aggregates multiple stack traces in the same group by constructing a graph where the nodes represent the stack trace functions and the edges represent the calling relationship. When applied to bug reports of two Windows products, Crash Graphs achieves 71.5% precision and 62.4% recall [10].

The problem with Crash Graphs and similar approaches is that they generate training models that are too rigid, making them hard to generalize to unseen cases. This may explain their low recall. What we mean by generalization is the capacity to model stack traces while considering possible calls that are not necessarily in the training set. Take for example the following fictive stack trace ABCDED used for training. A model that represents this trace should classify traces

ABABCDED or AAABCDED as similar because they only differ from ABCDED due to the contiguous repetition of AB and A respectively. Contiguous repetitions can be due to loops in the program. They did not appear in the stack trace used for training because the loop was not exercised during the scenario that led to the crash. We will see in the rest of this chapter that generalization goes beyond considering just contiguous repetitions.

In this chapter, we propose an approach called CrashAutomata that uses a combination of varied-length n-grams and automata to model stack traces of duplicate bug report groups. CrashAutomata is inspired by the work of Jiang et al. [80], who developed an algorithm for anomaly detection that is made generalizable by controlling a certain threshold variable α. We adopted the algorithm to model stack traces and detect duplicate bug reports. We experimented with various values to determine the most suitable value of α that yields the best detection accuracy. We also compared CrashAutomata with Crash Graphs. The results show that our approach has a better recall than Crash Graphs while keeping the same level of precision.

4.2. CrashAutomata Approach

Figure 4.1 shows an overview of the CrashAutomata approach. This approach is divided into two phases: The training and testing (detection) phases. In the training phase, we use the historical stack traces of duplicate bug report groups, identified in Chapter 3 for each dataset. We extract varied-length n-grams from the stack traces of each group. These n-grams will be used to construct an automaton for each group. To control the level of generalization of the automaton, we use a threshold α, which regulates the number of extracted n-grams (see Section 4.2.1.). The testing phase consists of assessing the effectiveness of the model in classifying new bug reports.

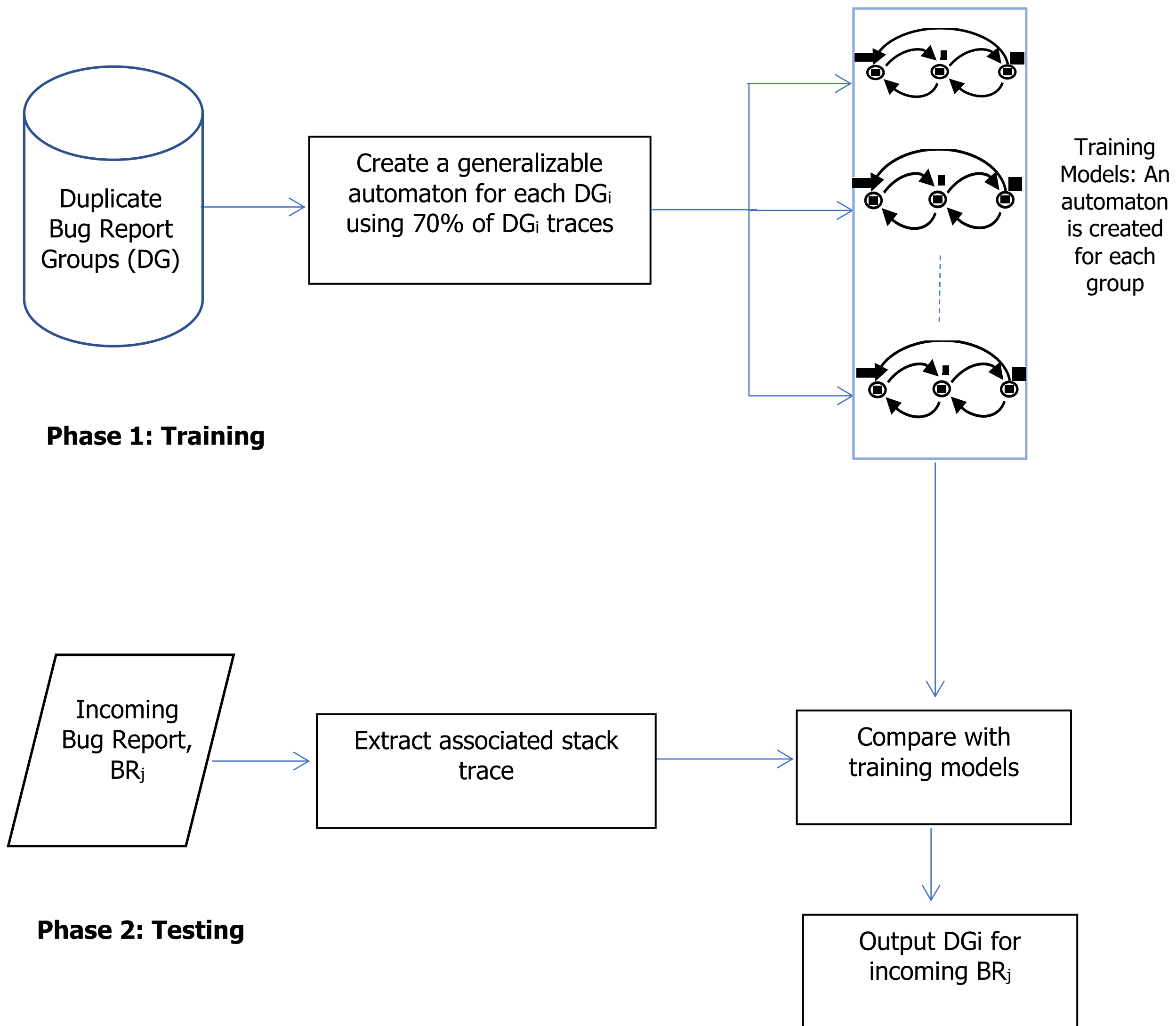


Figure 4.1. CrashAutomata Overall Approach

4.2.1 Training Phase: Automata Construction

We build an automaton from the stack traces of each bug report group, DGi. One way to achieve this is to simply consider each frame signature as a state in the automaton. A transition from one state to another occurs between two consecutive frame signatures. This method, however, suffers from two limitations. First, it may result in large automata, which may impact the scalability of the approach. The second and perhaps most important limitation is that it tends to be too rigid, meaning that the resulting automata cannot easily generalize to unseen cases.

We address the scalability problem by using an n-gram extraction technique that identifies the common sub-sequences or patterns within a sequence of events in a stack trace, where the length of the patterns can vary from 1 to *n* (the number of frame signatures in a trace). To this end, we adopt the algorithm presented by Jiang et al. in [80], used to detect anomalies in large datasets. This algorithm analyzes the training stack trace event sequences and extracts frequent patterns as n-grams according to a threshold α. The threshold is used to control the level of generalization of the resulting automaton.

To illustrate the steps of the algorithm (see Algorithm 1), we use the following three sequences (taken from [80]): T1; ABCDE, T2; CDEA and T3; CDEBA. For the current study, these sequences represent three stack traces of the same duplicate bug report group, where A, B, C, D, and E are distinct frame signatures. In the beginning of the process, the algorithm extracts all unique frame signatures from the stack traces and labels them as 1-gram.

In the consecutive steps, two n-grams of length k (C_k^i and C_k^j) are combined to make an n-gram of length k+1. The new pattern, which we refer to here as "p_{k+1}", is retained in the list of final n-grams if the frequency of "p_{k+1}" is greater than α multiplied by the minimum frequency of C_k^i and C_k^j. Otherwise, it is pruned. From the previous example, take α = 0.6. If we combine the two valid 1-grams A and B, we obtain AB. However, the frequency of AB in all traces is always 1 (it only appears in T1), which is less than α (= 0.6) * minimum frequency of A and B (= 1). Therefore, AB will be pruned from the final list of n-grams that will form the automaton. The pattern CD, on the other hand, which is a composition of two valid 1-grams C and D, is retained because its frequency (which is 3) is greater than α (= 0.6) * minimum frequency of C and D (= 3). The process of constructing n-grams continues this way until there are no n-grams to construct.

In our case, the 3-gram CDE is the last n-gram to be constructed (k would be 3). The final list of n-grams output by the algorithm when using traces T1, T2, and T3 is shown in Figure 4.2.

K1	K2	K3
A (3) ✓ B (2) ✓ C (3) ✓ D (3) ✓ E (3) ✓	AB (1) ✗ BA (1) ✗ BC (1) ✗ CD (3) ✓ DE (3) ✓ EA (1) ✗ EB (1) ✗	CDE (3) ✓

Figure 4.2. The n-grams extracted using Algorithm 1 applied to T1, T2, and T3

Note that the value of α varies from 0 to 1. A smaller α constructs a more generalized model, whereas when α is closer to 1, the model becomes more rigid. If $\alpha \rightarrow 0$, the longest n-gram is the trace length itself, whereas when $\alpha \rightarrow 1$ the n-grams are all 1-grams in the sequence. The challenge is to find an appropriate α that yields the best accuracy when classifying incoming reports.

An automaton that is too general (when α converges with 0) will lead to many false negatives. On the other hand, an automaton that is too strict (when α converges with 1) will result in many false positives. In the current case study, we experiment with values of α that vary from 0 to 1 to determine which values of α lead to the best accuracy.

Algorithm 1

Input: the set of unique traces
Output: the sets of varied-length n-grams.

$C_1 = \{$the set of single components c_1^i with $f(c_1^i) > 0\}$.
$k = 1$
do
  for each two elements c_k^i, c_k^j from the set C_k,
    if the last $k-1$ component sequence of c_k^i equals
      the first $k-1$ component sequence of c_k^j,
      then generate a new sequence
        $s = c_k^i$ plus the last component of c_k^j;
        count $f(s)$, the number of times that s appears in the trace data;
        if $f(s) > \alpha \cdot min(f(c_k^i), f(c_k^j))$,
          then put s into the set C_{k+1}.
  $k = k+1$.
while C_k is not empty
return all C_j, for $1 \leq j \leq k-1$

Algorithm 1. Algorithm for varied-length n-gram extraction (from [80])

To build the automaton from the list of the varied-length n-grams previously extracted, we adopt Jiang et al.'s algorithm in [80] (see Algorithm 2). The output of the algorithm is a state transition matrix E where the rows and columns represent the n-grams extracted from the previous step. The algorithm starts with the n-gram set that has the longest length k (in the previous example, k = 3). Within each set of k-grams, it processes the k-grams in the descending order of their frequency (i.e., the k-gram that has the highest frequency is processed first). These rules aim to minimize the final number of n-grams and edges in the final automaton. The next steps of the algorithm are straightforward. For each element of a set of k-grams, we search in the trace to see if it exists, and if so, it is replaced by a state number (that can be saved in a table along with their

corresponding elements). When applied to traces T1, T2, and T3, the resulting E matrix with $\alpha = 0.6$ is shown in

Table 4.1. Figure 4.3 shows the automaton extracted from this matrix. As we can see from this figure, the resulting automaton generalizes to sequences that are not in T1, T2, and T3. For example, the sequence ABCDEBCDE would be considered as a valid sequence.

Algorithm 2. Automaton Construction

for α *from* 0 *to* 1 step 0.1

Input: the set of unique traces and the sets of n-grams

Output: the automaton *E*

set $E[m][n] = 0$ for any two n-grams m, *n*

for each trace *T*

set $k = L$ and $l = T's$ length

do

for each k-gram C_k^i selected from C_k according to the sorted order (the most frequent one first),

search and replace all C_k^i in *T* with the assigned state number;

if the length of the replaced part equals *l*,

then break from the inner loop.

$k = k - 1.$

while the length of the replaced part $\neq l$ and $k \geq 1$.

from left to right, set $E[m][n] = 1$ if an n-gram *n* follows another n-gram *m* contiguously in the trace *T*

remove the unused n-grams/states from *E*

return the matrix *E*

Algorithm 2. Automaton construction algorithm used in CrashAutomata (taken from [80])

Table 4.1. The E matrix constructed with CrashAutomata from traces T1, T2, and T3

N-gram	A	B	C	D	E	CD	DE	CDE

A	0	1	0	0	0	0	0	0
B	1	0	0	0	0	0	0	1
C	0	0	0	0	0	0	0	0
D	0	0	0	0	0	0	0	0
E	0	0	0	0	0	0	0	0
CD	0	0	0	0	0	0	0	0
CDE	1	1	0	0	0	0	0	0

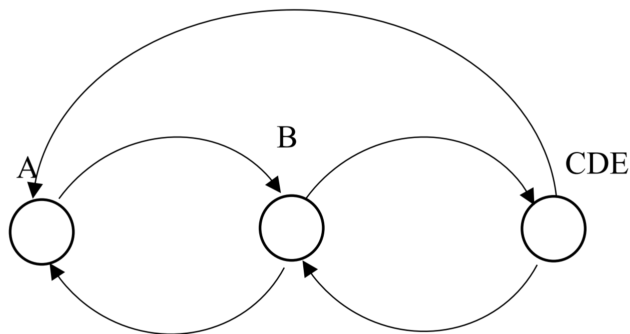


Figure 4.3. The automaton that is extracted from T1, T2, and T3 with $\alpha = 0.6$

4.2.2 Testing (Detection) Phase

Once we construct the automata for the duplicate bug reports in the dataset, we use them to classify incoming bug reports according to their stack traces. For this, we need to change the sequences of an incoming stack trace into the extracted n-grams identified previously. If the trace contains non-defined n-grams (due, for example, to new functions that were not encountered during the training phase), we simply assign to them a new ID. We compare the sequence of n-

grams in the new trace with the ones in the corresponding automata. The automaton with the highest similarity is output as a potential duplicate group of the incoming bug report.

4.2.3 Evaluation Metrics

This study measures the effectiveness of CrashAutomata using precision, recall, and the F-measure; which are defined using true positive (TP), false positive (FP), and false negative (FN) [81]. For a duplicate bug report group DG_i, we measure TP, FP, and FN as follows:

- TP_{DGi} = The number of traces that are correctly classified
- FP_{DGi} = The number of traces of other duplicate bug report group that are classified as parts of bug report group DG_i
- FN_{DGi} = The number of traces of bug report group DG_i that were classified as belonging to other bug report groups other than DG_i

We derive precision and recall for each duplicate bug report group, DG_i, as follows. Note that a high FP will reduce precision, whereas a high FN will reduce recall:

$$Precision\ (DG_i) = \frac{TP_{DGi}}{TP_{DGi} + FP_{DGi}}$$

$$Recall\ (DG_i) = \frac{TP_{DGi}}{TP_{DGi} + FN_{DGi}}$$

$$F_{measure}(DG_i) = 2 * \frac{Precision(DG_i) * Recall(DG_i)}{Precision(DG_i) + Recall(DG_i)}$$

4.3. Evaluation

The objective of the case study is to assess the accuracy of CrashAutomata using the Recall and F-measure on both Firefox and GNOME datasets. We also determine the most suitable α value that yields best accuracy. In addition, we compare CrashAutomata to Crash Graphs. We chose Crash Graphs because (a) it relies only on stack traces, and (b) provides the best accuracy so far compared to other known techniques.

For each duplicate bug report group, we chose 70% of the stack traces for training and used the remaining 30% for testing, since this is a typical practice in machine learning [82]. We then constructed an automaton for each duplicate bug report group with the 70% of traces and run CrashAutomata using different values of α to determine the most suitable α. Table 4.3 and Table 4.4 show the true positive, false positive, and false positive values of CrashAutomata for a randomly selected set of duplicate bug report groups for Firefox and GNOME respectively when using $\alpha = 0.9$ and $\alpha = 1$ (the full confusion matrices are shown in Table 4.2). The rows refer to the individual duplicate bug report groups, while the columns provide the classification details. The first column provides the duplicate bug report group ID, followed by the number of stack traces in each group. They are further identified as either true positive, false positive, and false negatives. For example, DG 92215 in Firefox dataset has 11 stack traces that with $\alpha = 0.9$, with 10 of the 11 stack traces classified correctly, and thus, no false positives and only 1 false negative. However, when $\alpha = 1$, none of the stack traces were classified correctly, and 25 stack traces belonging to other groups were assigned to this group incorrectly (i.e., a false positive). Therefore, in general, the number of false positives with $\alpha = 1$ is higher than with $\alpha = 0.9$. This is because $\alpha = 1$ results in the finer granularity of n-grams and therefore more variety of n-grams exist in the CrashAutomata. So, when a stack trace arrives in the system there exists more n-grams similarity between the stack's n-grams and n-grams of irrelevant groups than the relevant group's n-grams.

DG 92215 in Firefox is an example of a high number of false positives resulting from this fact.

Based on the predicted labels, we calculated the Recall and F-measure for all duplicate bug report groups with the average Recall and F-measures. Figure 4.4 and Figure 4.5 show the average F-measure obtained from all of Firefox and GNOME's duplicate bug report groups by varying the value of α from 0 to 1 with a 0.1 step. As shown in Figure 4.4, the best F-measure of 90% accuracy for Firefox is obtained when α = 0.9. Meanwhile, an accuracy of 92% is obtained for GNOME when α = 0.5 (see Figure 4.5).

Figure 4.6 and Figure 4.7 show the F-measure boxplots for the duplicate bug report groups in Firefox and GNOME by varying α. Since the Precision and Recall for α = 1 are low and close to 0 in many groups, the F-measure drops dramatically from α = 0.9 to α = 1.

Table 4.3. An example of a classification result of CrashAutomata for nine duplicate bug report groups in Firefox

		α = 0.9			*α = 1*		
DG_{ID}	**Total Stack Traces in Test set**	**TP**	**FP**	**FN**	**TP**	**FP**	**FN**
1001195	3	0	0	3	0	0	3
1011391	17	17	0	0	0	0	17
1034254	13	10	0	3	0	0	13
1041489	13	6	0	7	0	0	13
1046231	3	3	0	0	0	0	3
1046285	5	0	0	5	0	0	5
1049138	14	2	0	12	0	0	14
105275	2	0	0	2	0	0	2
1053934	13	9	0	4	1	0	12
1077858	2	2	0	0	0	0	2
1115929	25	24	0	1	0	0	25
1133405	11	11	0	0	0	0	11
1149498	11	6	0	5	0	0	11

1191635	17	15	0	2	0	0	17
1191756	2	2	0	0	0	0	2
1193043	2	2	0	0	0	0	2
121055	26	23	0	3	0	0	26
1215944	6	0	0	6	0	0	6
1215992	11	10	0	1	0	0	11
462778	9	9	0	0	0	6	9
480345	2	2	0	0	0	1	2
65838	2	2	0	0	0	2	2
77716	1	0	0	1	0	2	1
87223	3	1	0	2	2	18	1
92215	11	10	0	1	0	25	11

Table 4.4. Classification result of CrashAutomata for duplicate bug report groups in GNOME

		α = 0.9			***α = 1***		
DG_ID	**Total Stack Traces**	**TP**	**FP**	**FN**	**TP**	**FP**	**FN**
138022	36	36	32	0	0	0	36
115940	6	4	13	2	0	0	6
147004	2	0	11	2	0	0	2
156997	13	13	11	0	0	0	13
109874	4	4	9	0	0	0	4
146676	7	2	9	5	0	0	7
119695	6	6	6	0	0	0	6
130205	5	0	6	5	0	0	5
152782	6	5	6	1	0	0	6
130246	3	3	5	0	0	0	3
146844	6	5	5	1	0	0	6
157737	10	10	5	0	0	1	10
131500	24	24	4	0	0	0	24
141523	12	1	4	11	0	0	12
159783	25	25	4	0	0	0	25
105680	14	14	3	0	0	0	14
110992	5	1	3	4	0	0	5
113666	5	5	3	0	0	0	5
116371	4	4	1	0	0	0	4
130291	6	5	0	1	1	0	5
141738	18	16	0	2	1	0	17
90862	6	6	0	0	1	79	5
100328	3	3	0	0	0	0	3

101109	2	2	0	0	0	0	2
101110	2	2	0	0	0	0	2

Table 4.5. Confusion matrices for Firefox and GNOME with Firefox and GNOME with α=0.9 and α=1.0

Firefox α=0.9	Positive	Negative
Positive	637	147
Negative	63	-

Firefox α=1.0	Positive	Negative
Positive	6	778
Negative	767	-

GNOME α=0.9	Positive	Negative
Positive	1013	219
Negative	140	-

GNOME α=1.0	Positive	Negative
Positive	10	1222
Negative	1221	-

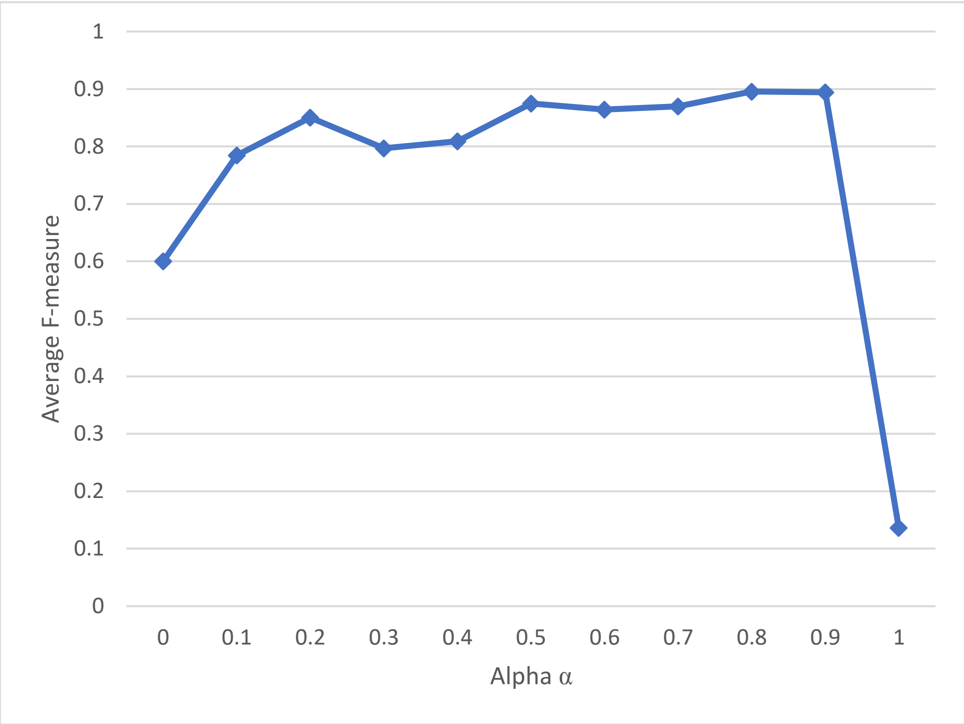


Figure 4.4. Average F-measure by varying α for Firefox dataset.

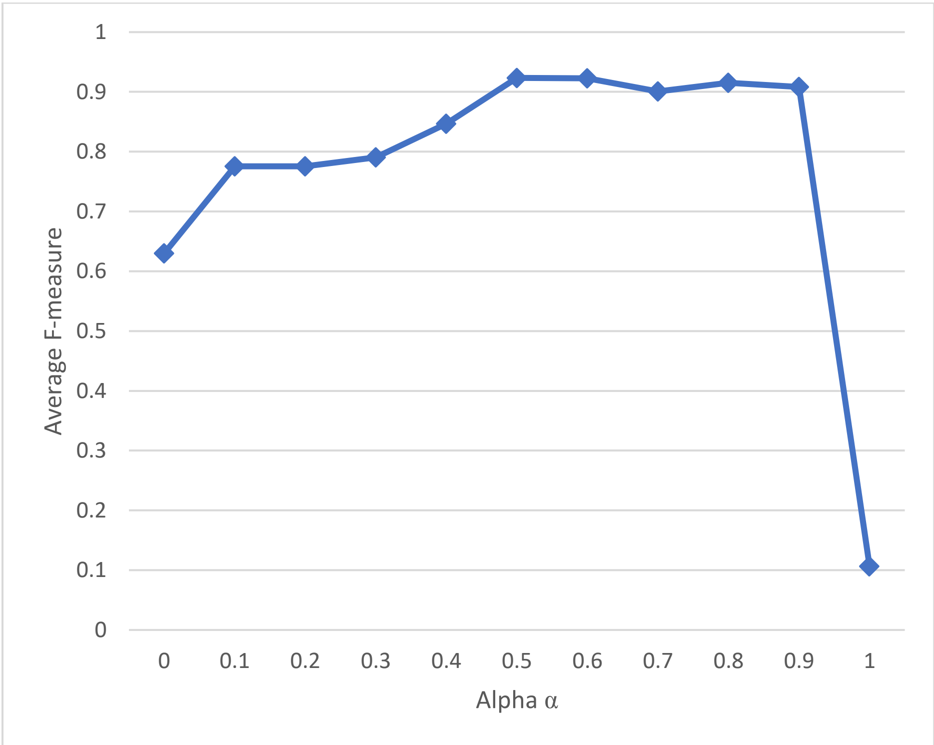


Figure 4.5. Average F-measure by varying α for GNOME dataset.

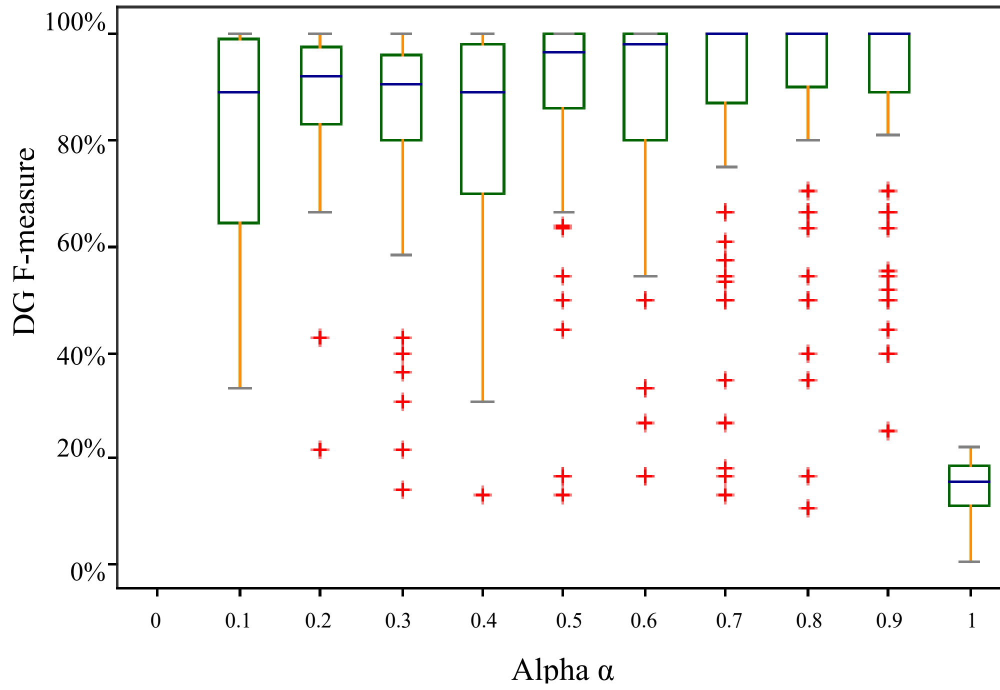


Figure 4.6. F-measure boxplot for Firefox

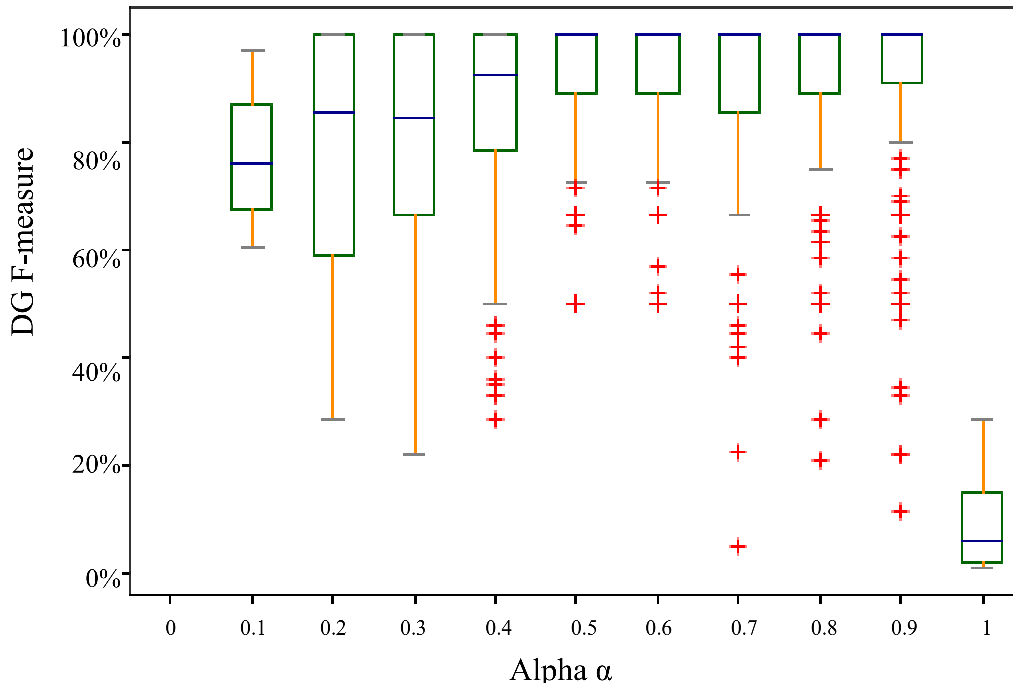


Figure 4.7. F-measure boxplot for GNOME

4.4. Comparison with Crash Graphs

Kim et al. [10] introduced Crash Graphs to detect duplicate crash (bug) reports in the WER (Windows Reporting System). The approach aggregates multiple stack traces (called crash traces in [10]) in the same group by constructing a graph where the nodes represent the stack trace functions and the edges represent the calling relationship.

Figure 4.8, taken from [10], shows an example of three stack traces ABCD, AFGD, and CDFG, where A, B, C, D, F, G are distinct functions. In this example, a graph is created by taking a 2-gram representation of trace elements and combining them.

The authors use the aggregated graph to model stack traces of each bucket (a group of related crash reports in WER) to predict if a new crash trace should belong to the bucket (this is equivalent to duplicate bug report group in our study) or not. A similarity metric is used to determine the extent to which an incoming stack trace is deemed similar to those modeled in the graph. When applied to detect duplicate bug reports of two Windows products, the best accuracy achieved by Crash Graphs is 71.5% precision and 62.4% recall using a 90% similarity, which we also used to run the experiments with Crash Graphs.

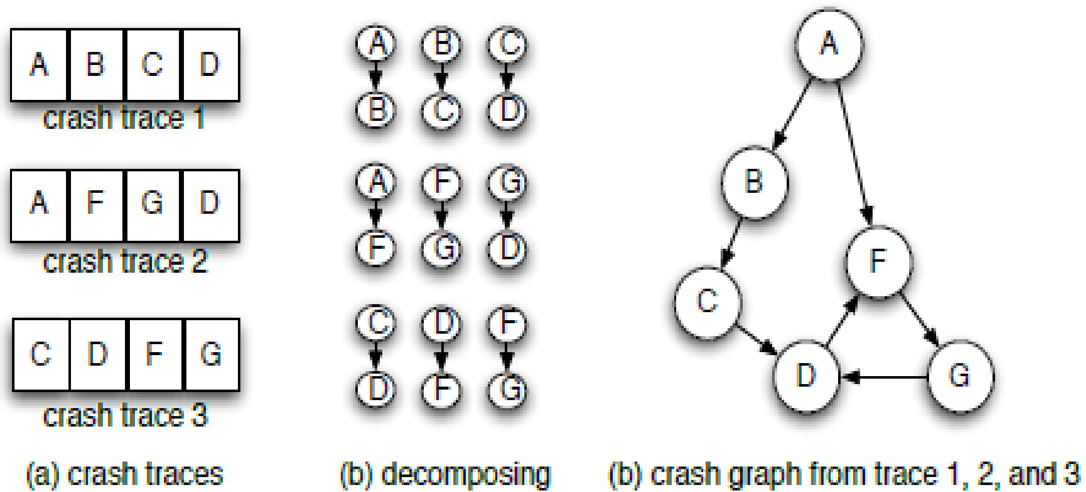


Figure 4.8. An example of a graph created by Crash Graphs to model traces [10]

We implemented Crash Graphs and applied it to our datasets with the objective of comparing its accuracy with CrashAutomata. Figure 4.9 compares the precision, recall, and F-measure boxplots achieved for all of Firefox and GNOME's duplicate bug report groups when using CrashAutomata and Crash Graphs. These metrcis show higher values in the GNOME dataset compared to the Firefox dataset. As shown in the boxplots, in all measures the CrashAutomata outperforms the Crash graphs.

We also examined the average precision, recall, and F-measure of both approaches on each dataset. The results are shown in Figure 4.10 and Figure 4.11. As we can see, CrashAutomata's precision, recall, and F-measure are significantly higher compared to Crash Graphs. We attribute this to the generalization ability of CrashAutomata.

Furthermore, we studied the number of false negatives for both approaches to pinpoint the buckets that caused the low recall. Figure 4.12 and Figure 4.13 compare the False Negative rates for both CrashAutomata and Crash Graphs in Firefox and GNOME. The results show that in

almost all buckets, CrashAutomata performs better than Crash Graphs. In Crash Graphs, many stack traces are classified incorrectly, increasing the rate of the false negatives. CrashAutomata shows the median false negative rates is closer to zero, whereas this percentage is around 50% using Crash Graphs and Firefox, and even more disappointingly, 40% in GNOME. These findings show that increasing the generalization of CrashAutomata reduces the number of false negatives significantly, leading to a much better recall.

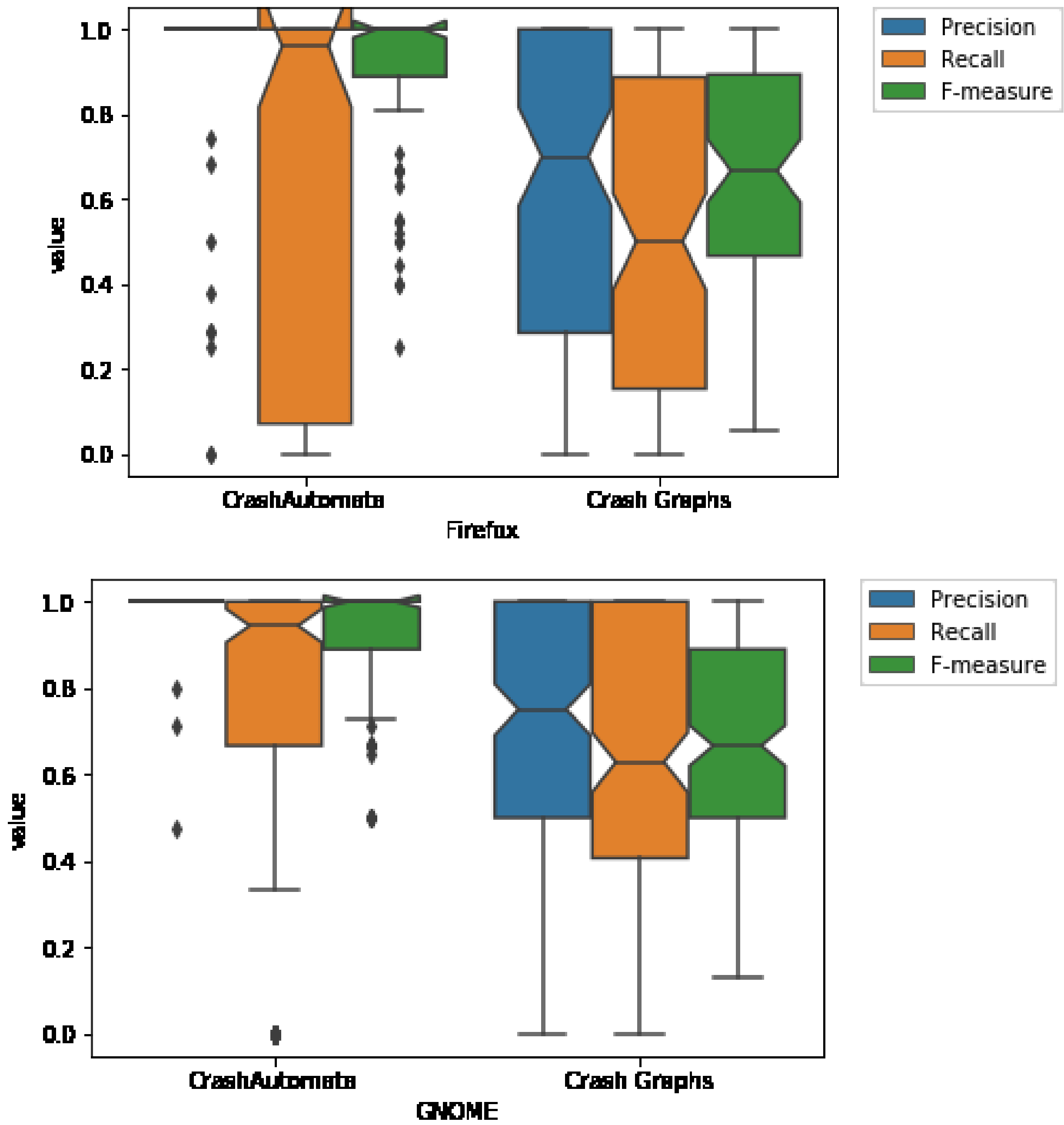


Figure 4.9. Boxplot of Precision, Recall and F-measure for all groups in Firefox and GNOME CrashAutomata (α=0.5) and Crash Graphs.

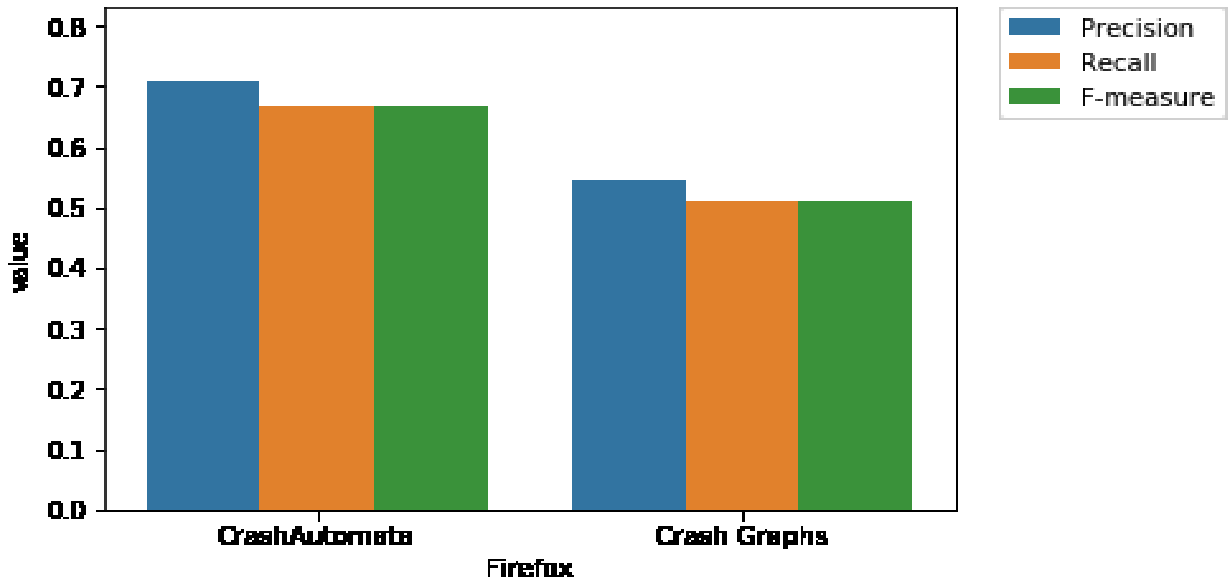


Figure 4.10. Average Precision, Recall and F-measure for CrashAutomata and Crash Graphs when applied to the Firefox dataset

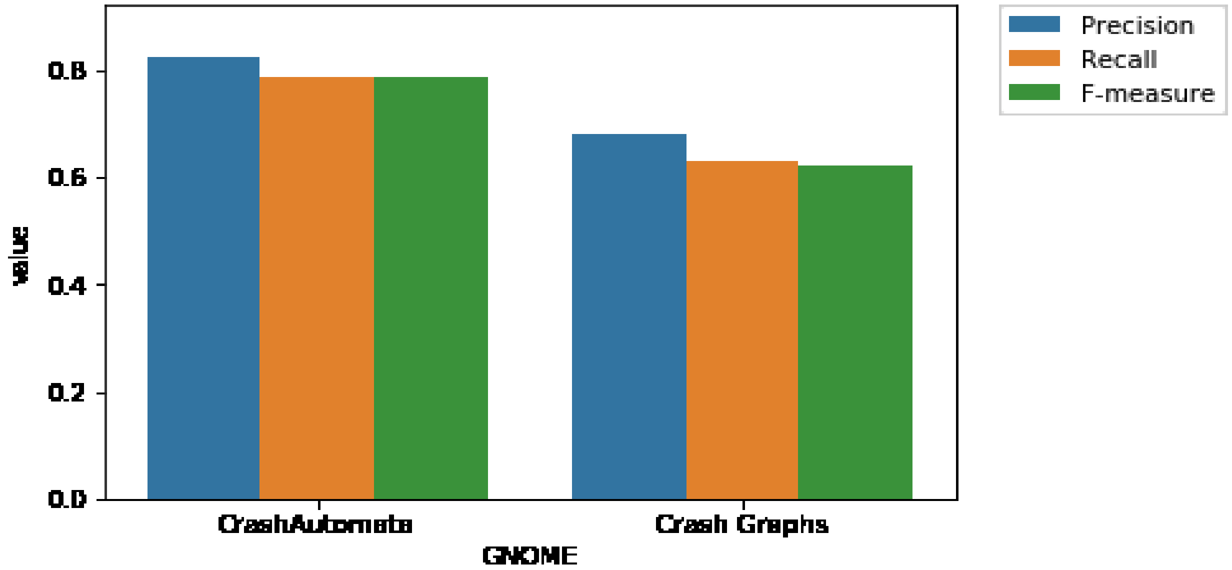


Figure 4.11. Average Precision, Recall and F-measure for CrashAutomata and Crash Graphs when applied to the GNOME dataset

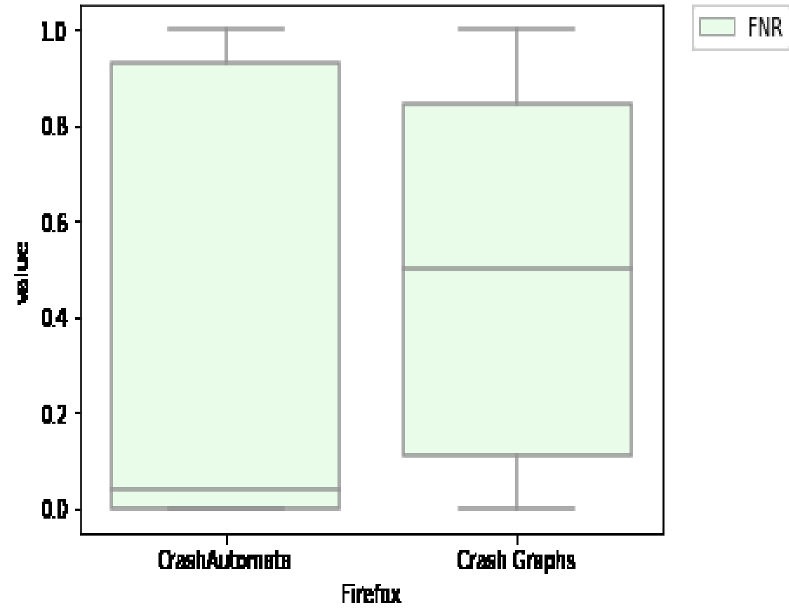


Figure 4.12. Comparison of variation of False Negative Rates for CrashAutomata and Crash Graphs for Firefox dataset

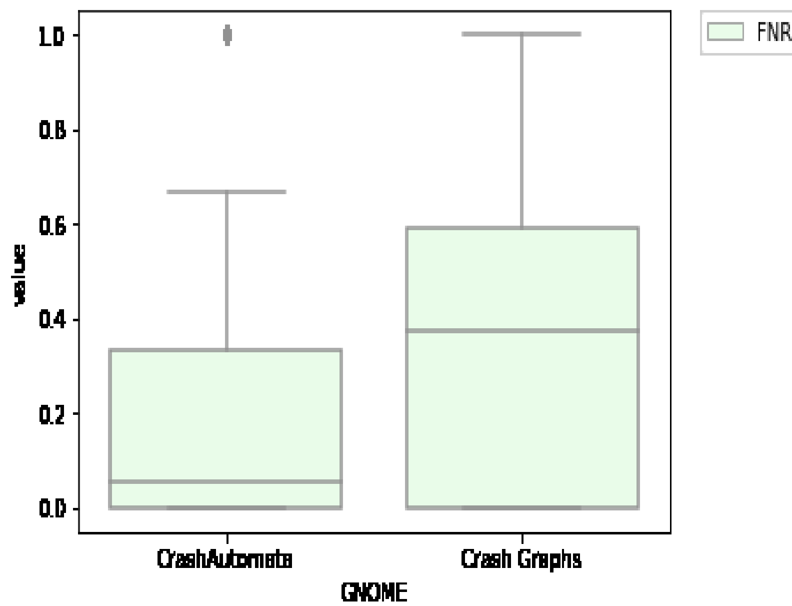


Figure 4.13. Comparison of variation of False Negative Rates for CrashAutomata and Crash Graphs for GNOME dataset

4.5. Discussion

The current case study shows promising results. CrashAutomata achieves 72% precision and 68% recall in Firefox, and a very impression result of 82% precision and 79% recall in GNOME. This means CrashAutomata has a better recall than Crash Graphs, which does not support any generalization of the trained model. In the following subsections, we discuss two aspects of CrashAutomata that need further study.

Generalization of the automata: The goal of CrashAutomata is to detect duplicate reports early in the crash-handling process to save software developers time and effort. Unlike existing techniques, CrashAutomata is built with generalization in mind by modeling stack traces in a way that unseen traces can be easily classified. Since not every normal trace is seen and collected in the training data, a certain capacity of generalization is desirable to reduce false positives and false negatives in detection. The question is, how many generalizations should the automata have in order to obtain good detection accuracy. Here, we discuss the findings of our experimentation with various values of α to find the most suitable one. We expect that α changes from one dataset to another. The danger with generalization is that it may lead to automata that are too loose, which may affect the true positives, meaning that true duplicates may end up misclassified as non-duplicates. It is therefore highly recommended to keep a tight representation of the automata to guarantee an adequate true positive rate.

Misclassified stack traces: There is a possibility to consider a threshold for the similarity between an upcoming stack trace and existing CrashAutomata groups. In this case, if the calculated similarity between the stack trace and CrashAutomata does not satisfy the similarity requirements, the stack trace will be labelled 'Unspecified'. Unspecified traces may be an indication that new

bug report groups are needed. Using CrashAutomata and the ability to create new groups can help design a new bug report grouping system that relies on the automata representation to classify incoming crashes. The new system starts with a reliable set of groups (just like the ones we constructed for Firefox and GNOME) and classifies incoming bug reports by measuring the similarity between the stack traces and the automata representation. Traces that show a high degree of dissimilarity with all existing groups should lead to the creation of new ones.

Differences between Firefox and GNOME: We found that our approach (as well as the Crash Graphs approach) performs better for GNOME than for Firefox. This may be due to the fact that GNOME has more traces than Firefox (4,600 compared to 2,883). We also have more duplicate groups in GNOME than in Firefox. Besides, GNOME has more duplicates in each DG than Firefox as shown in Chapter 3. More traces in a duplicate group would mean a better characterization of bug report of the same group, which helps with the classification process achieved by CrashAutomata (and Crash Graphs).

4.6. Chapter Summary

In this chapter, we presented CrashAutomata, which is a technique for classifying duplicate bug reports using stack traces. Unlike other techniques, CrashAutomata is built with generalizations in mind. Stack traces are first processed to extract varied-length n-grams, that are used to form automata. The extract algorithm relies on a variable α that controls the level of generalization of the automaton. The idea is to have a model that can be general enough to classify similar traces that were unseen during training. Once the automata are built, every time a new stack trace (bug report) arrives, we determine its duplicate group by comparing it to traces of existing duplicate bug report groups using their corresponding generalizable automata. To make our

approach practical, new groups can be created if the similarity is below a certain threshold, meaning that the new bug report does not have already established duplicates. This threshold needs to be determined in practice. Note that, in our experiments we are using labelled data, meaning that all bug reports considered in the testing phase have duplicates.

We experimented with CrashAutomata on crash reports from the Firefox and GNOME systems. The F-measure of our approach on average is 68% for Firefox and 75% for GNOME. We showed that CrashAutomata outperforms Crash Graphs, resulting in better precision and recall than Crash Graphs. We attributed this improved precision and recall to the generalization power of CrashAutomata.

Chapter 5. An HMM-Based Approach for Automatic Detection and Classification of Duplicate Bug Reports

In this chapter, we further improve CrashAutomata, presented in the previous chapter, by leveraging the use of Hidden Markov Models (HMM). HMM is a machine learning technique that is used in speech recognition, DNA sequence analysis [83], and language processing [37], [84]. HMM is especially designed for sequential data analysis [85], [86]. We use HMM to model function calls of stack traces of historical bug reports to build a training model that is later used to classify incoming bug reports.

5.1. Background on HMMs

A Markov process typically assumes that the states are directly visible to the observation data produced in the system. While in HMM, the states are hidden, but the output of each hidden state (i.e., the state transition probability) is dependent on the observation data. Moreover, based on the data types, an HMM can be further classified into discrete (typically the data is a discrete sequence produced from a finite number of tokens or symbols over time) or continuous (typically the data is generated from a Gaussian distribution such as speech, music, etc.). Since the observation data in our system is a discrete sequence of function calls, we have used the discrete form for the output distributions to model the function calls forming stack traces. A typical topology of an HMM is shown in Figure 5.1.

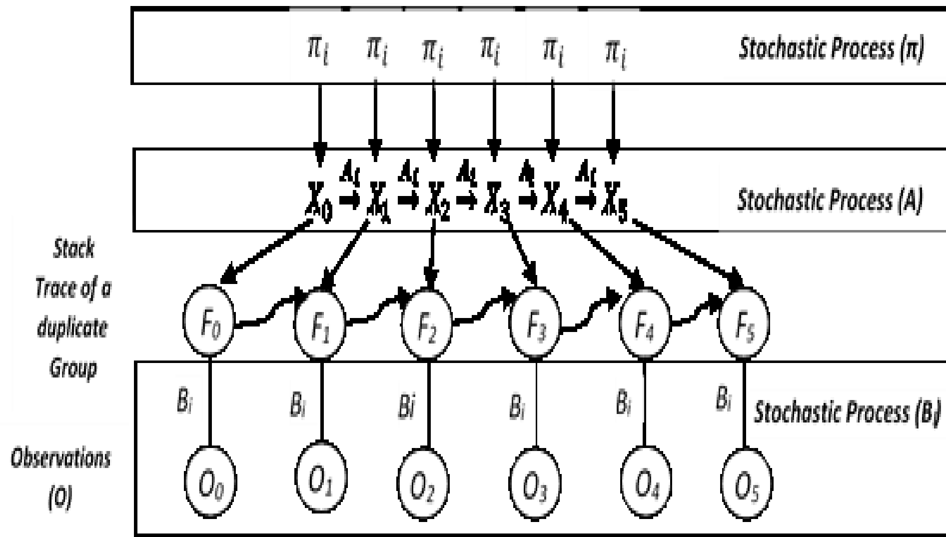


Figure 5.1. Typical topology of an HMM

Training an HMM model requires defining the parameters, described in what follows. The number of hidden states: To train an HMM model, we need to set the number of hidden states (N) in the Markov process. Let the distinct states in a Markov process be S_i, $i = \{0,1,\dots,N-1\}$ and the notation $X_t = S_i$ represents the hidden states sequence S_i at time t.

The number of observation symbols: To train an HMM model, we need to set the number of observation symbols (M). Let the distinct observation symbols be R_k, $k = \{0,1,\dots,M-1\}$ and the notation $O_t = R_k$ represents the observed symbol R_k at time t for the given sequence of observations $(\mathcal{O}_0, \mathcal{O}_1, \dots, \mathcal{O}_{T-1})$, where T is the length of that sequence.

State transition probability distribution: The first-row stochastic process is the hidden state transition probability distribution matrix $A = \{a_{ij}\}$. A is an $N \times N$ square matrix and the probability of each element $\{a_{ij}\}$ is calculated by the following equation:

$$a_{ij} = P\left(state\ S_j\ at\ t+1 \middle| state\ S_i\ at\ t\right),\ i,j = \{0,1,\dots,N-1\}$$

In Equation (1), the transition from one state to the next is a Markov process of order one [37]. This means that the next state depends only on the current state and its probability value. Since the original states are "hidden", we cannot directly compute the probability values in the past. However, we can observe the observation symbols for the current state S_i at time t from the given observations sequence O to train an HMM model.

Observation symbol probability distribution: The second-row stochastic process is the observations symbol probability distribution matrix $B = \{b_j(R_k)\}$. B is an $N \times M$ matrix which is computed based on the observation sequences (i.e., the temporal order of stack traces). The probability of each element $b_j(R_k)$ is given by the following equation:

$$\mathrm{b_j(R_k) = P(observation\ symbol\ R_k\ at\ t | state\ S_j\ at\ t)}$$

Initial state probability distribution: The third-row stochastic process is the initial state probability distribution $\pi = \{\pi_i\}$. π is a $1\times N$ row matrix and the probability of each element $\{\pi_j\}$ is given by Equation (3)

$$\pi_i = \mathrm{P(state\ S_i\ at\ t = 0)}$$

Training an HMM model aims to maximize the likelihood function $P(O|\lambda)$ over the above three-parameter space. The Baum-Welch (BW) algorithm is the most commonly employed expectation-maximization (EM) algorithm to estimate HMM parameters. It uses a forward-backward (FB) algorithm at each iteration to efficiently evaluate the likelihood function $P(O|\lambda)$. It updates the model parameters until a maximum number of iterations is reached or the likelihood function achieves no more improvement. In this chapter, we also use BW to estimate HMM parameters.

5.2. Overall Approach

Our approach for using HMM to detect duplicate bug report is similar to the one of CrashAutomata, except that for each duplicate bug report group, DG_i, we train an HMM using 60% of the contained traces, validate the HMM using 10% of the traces, and test the model using 30% of the traces of this DGi and every other DG. The validation step is needed to set the HMM parameters. In addition, because it is difficult to know in advance how many hidden states are needed, we train with multiple number of hidden states with the dual objective of (1) determining the most suitable number of hidden states, and (2) determining whether the number of hidden states has at all an impact on the overall approach.

5.2.1 Training Phase

We build an HMM for each duplicate bug report group, DG_i. We vary the number of hidden states N from 15 to 50 with a leap out of 5. To our knowledge, no study specifies how to set the number of hidden states. Most studies that use HMMs set this parameter through experimentation. In our case, we found that for Firefox and GNOME, the best accuracy is obtained when N=20 and N=40, respectively. We experimented with higher N > 50 values and observed no improvement.

5.2.2 Validation Phase

Validation is used to better estimate the best fit for the HMM parameters A, B, and π. In our study, we used 10% of traces in each DG_i to validate the HMM constructed through training. We performed 10 iterations[6] to estimate the HMM parameters A, B, and π. Initial parameter values are

[6]Our experiments have shown that after 10 iterations, the parameter values do not vary.

passed to the Baum-Welch algorithm to compute the log-likelihood as scores for all traces inside the validation set. The best-recorded parameters A_i , B_i and π_i that are obtained from the minimum mean value among the 10 iterations are used to construct the HMM models.

5.2.3 Testing Phase

We used 30% of traces from each duplicate bug report group as a testing dataset. Let a new stack trace ST_i be mapped to sequences of observations $O_i = \{O_1, O_2, \dots, O_{T-1}\}$. The latter ones are presented to HMM models, $\lambda_l = \{\lambda_1, \lambda_2, \dots, \lambda_L\}$, of all duplicate bug report groups. Then, the log-likelihood of the sequence of observations $P(O_i|\lambda_l), \forall\, l = \{1, 2, 3, \dots, L\}$ for every trained HMM model is calculated. A set of ordered scores for all HMM models, $S = \{S_1, S_2, \dots, S_L\}$ is subsequently generated and reported to specify possible labels within the ranked list.

5.2.4 Evaluation Metrics

Since HMM outputs similarity scores for each stack trace, it is easier to provide a list of duplicate candidates to the developer for further investigation. Therefore, we used the Recall Rate@k and the Mean Average Precision (MAP) to assess the effectiveness of our approach. These metrics are used extensively in the literature [11], [15], [27], [44], [77] so as to evaluate the performance of a ranked list. The Recall Rate@k is defined as:

$$\text{Recall Rate@k} = \frac{\text{N}_{\text{detected@k}}}{\text{N}_{\text{total}}}$$

where $N_{detected@k}$ is the number of correctly retrieved k stack traces. Recall Rate@k is defined as the percentage of duplicates for which the master is found for a given top list size k.

The Mean Average Precision (MAP) indicates how accurately duplicate candidates are

ranked. It is measured as follows:

$$\mathrm{MAP} = \frac{1}{\mathrm{Q}}\sum_{\mathrm{n}=1}^{\mathrm{Q}} \frac{1}{\mathrm{rank(n)}}$$

where Q is the number of correctly retrieved duplicate candidates and $rank(n)$ is the position in which the right stack trace is retrieved.

MAP ranges from 0 to 100%. An approach that returns MAP=100% means that for all bug reports in the testing set, the approach was able to classify them accurately at the top rank. It is sufficient for a triager to look at one duplicate bug report group to find the corresponding duplicate group. A MAP close to zero means that the approach would return many possible duplicate bug report groups for each bug report in the testing set because of a poor classification.

5.3. Evaluation

5.3.1 Firefox and GNOME Datasets

The results of applying our approach to Firefox dataset are shown in Table 5.1. The recall rates with ranks with k ranging from 1 to 20 show that our approach achieves promising results across all HMMs with different states. The average recall for Rank k =1 is 59%, for Rank k=2 is 75.55%. We start reaching the 90% recall from k=10.

In the case of MAP (see Figure 5.2), we obtained MAP values between 75.77% and 76.44% with different numbers of hidden states, an average of 76.24%. In other words, a given incoming bug report can be identified by our approach in the first duplicate bug report group that the approach suggests with 76% of chances. We pass to the 85% MAP bar with 5 duplicate bug report groups, which we believe it is considered a good result.

Table 5.1. Median of Recall Rate@k with different HMM state numbers using Firefox dataset.

Rank	Number of Hidden States							
	15	20	25	30	35	40	45	50
1	59.08%	59.70%	59.26%	59.81%	57.91%	58.21%	58.86%	58.50%
2	74.28%	76.36%	76.01%	75.76%	74.32%	75.23%	76.79%	75.67%
3	80.09%	81.20%	81.90%	81.84%	80.80%	80.55%	82.10%	82.95%
4	84.64%	85.44%	86.19%	85.08%	85.29%	85.24%	85.36%	85.19%
5	86.92%	86.55%	87.46%	87.33%	86.41%	87.05%	86.58%	87.11%
6	87.95%	88.53%	88.14%	88.17%	88.16%	88.10%	87.44%	88.52%
7	88.62%	89.03%	88.69%	88.64%	88.67%	88.71%	87.84%	88.65%
8	88.91%	89.48%	88.91%	88.80%	88.87%	88.88%	88.62%	88.82%
9	89.15%	89.57%	89.31%	89.20%	89.08%	89.19%	88.83%	89.03%
10	89.88%	89.90%	89.76%	89.52%	89.53%	89.54%	89.22%	89.47%
11	90.10%	90.00%	89.95%	89.84%	89.73%	89.76%	89.37%	89.57%
12	90.25%	90.12%	89.99%	89.96%	89.87%	89.85%	89.46%	89.86%
13	90.25%	90.12%	90.15%	90.04%	89.87%	89.94%	89.79%	90.04%
14	91.29%	90.81%	90.64%	90.58%	90.57%	90.72%	90.28%	90.55%
15	91.37%	91.36%	91.19%	90.65%	90.75%	91.28%	90.88%	91.22%
16	91.52%	91.48%	91.28%	90.81%	91.05%	91.36%	91.04%	91.31%
17	91.98%	91.80%	91.74%	91.66%	91.37%	91.92%	91.42%	91.77%
18	92.02%	92.26%	91.77%	91.77%	91.41%	91.98%	91.46%	91.77%
19	92.02%	92.26%	91.77%	91.84%	91.90%	91.98%	91.79%	91.77%
20	92.08%	92.33%	92.08%	91.84%	92.04%	92.02%	91.79%	91.77%
MAP	75.77%	76.44%	76.29%	76.40%	76.15%	76.33%	76.29%	76.26%

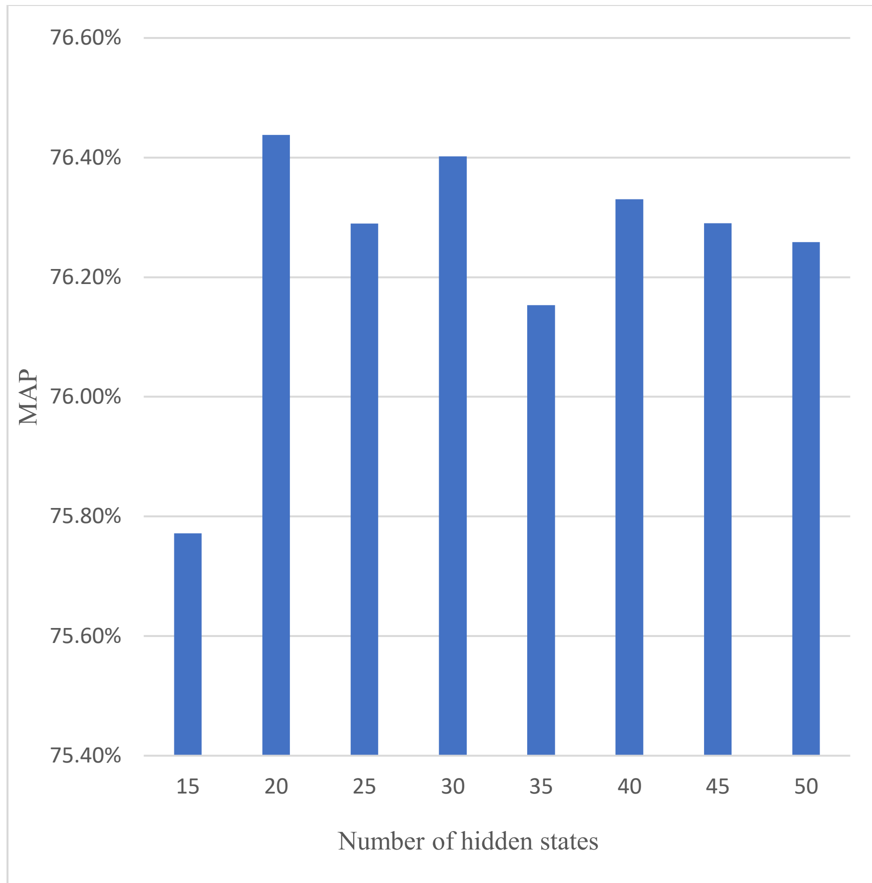


Figure 5.2. MAP obtained with different HMM state numbers for Firefox

Table 5.2 shows the results of the recall at rank k with k ranging from 1 to 20 with HMM models with different state numbers for GNOME dataset. The recall at k=1 is almost 63% for all state numbers, while this value increases by about 10% for k=2. It can also be observed that having a recommended list of 2 duplicate groups, the detection accuracy of about 73% can be achieved. This detection accuracy increases to 97% for k=11. In terms of MAP rate, values of about 72% and 73% were achieved using state numbers of 40 and 45, respectively (see Figure 5.5). Similar to Firefox, we did not see a significant impact in changing the number of hidden states.

Table 5.2. Median of Recall Rate@k with different HMM state numbers using GNOME dataset.

Rank	Number of Hidden States							
	15	20	25	30	35	40	45	50
1	62.99%	62.74%	62.66%	63.39%	63.15%	63.72%	62.18%	62.50%
2	73.21%	73.54%	73.54%	73.30%	73.13%	72.81%	73.38%	73.21%
3	76.95%	77.27%	76.54%	76.87%	76.70%	76.79%	76.54%	77.03%
4	78.57%	78.73%	78.17%	78.08%	78.49%	78.41%	78.33%	78.65%
5	79.79%	79.79%	79.38%	79.79%	79.71%	79.38%	79.30%	80.19%
6	80.76%	81.25%	80.60%	81.09%	81.33%	80.68%	80.52%	81.09%
7	82.06%	82.47%	81.74%	82.39%	81.98%	81.98%	81.57%	81.98%
8	83.36%	83.36%	82.47%	83.20%	82.79%	82.95%	82.71%	82.87%
9	83.85%	83.93%	83.36%	84.17%	83.44%	83.77%	83.52%	83.60%
10	84.50%	84.50%	84.09%	84.74%	84.25%	84.17%	84.58%	84.82%
11	85.23%	85.23%	84.90%	85.39%	85.15%	84.98%	85.06%	85.39%
12	85.63%	86.36%	85.63%	85.88%	85.88%	85.96%	85.80%	85.96%
13	86.53%	87.09%	86.28%	86.61%	87.18%	86.61%	86.53%	86.85%
14	87.18%	87.58%	87.01%	87.34%	87.74%	87.01%	87.26%	87.26%
15	87.91%	87.91%	88.07%	88.07%	88.23%	87.99%	88.31%	87.91%
16	88.56%	88.56%	88.88%	88.56%	88.72%	88.47%	88.80%	88.72%
17	89.04%	88.96%	89.69%	88.96%	89.12%	89.29%	89.20%	89.37%
18	89.53%	89.53%	90.26%	89.37%	89.53%	89.61%	89.77%	89.69%
19	90.02%	89.85%	90.58%	90.02%	90.34%	90.26%	90.26%	89.94%
20	90.26%	90.42%	90.99%	90.58%	90.91%	90.99%	90.58%	90.34%
MAP	71.32%	71.33%	71.13%	71.53%	71.37%	71.57%	70.86%	71.11%

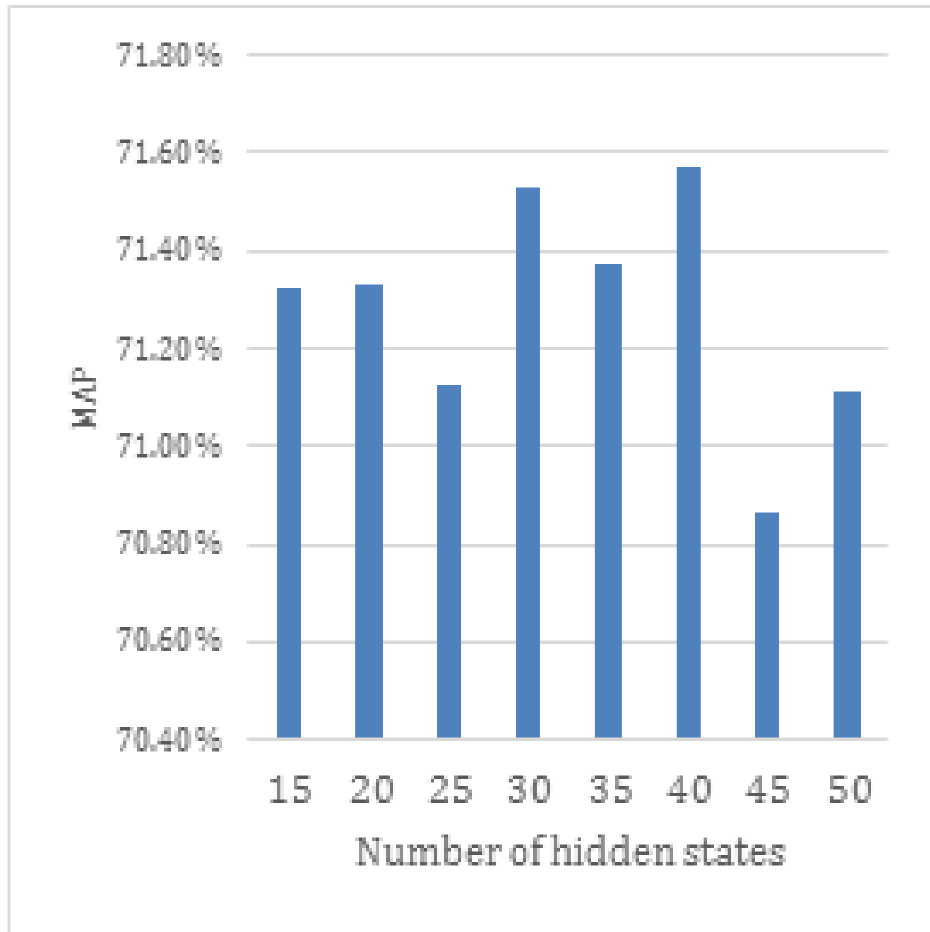


Figure 5.3. MAP obtained with different HMM state numbers for GNOME

5.3.2 Comparison

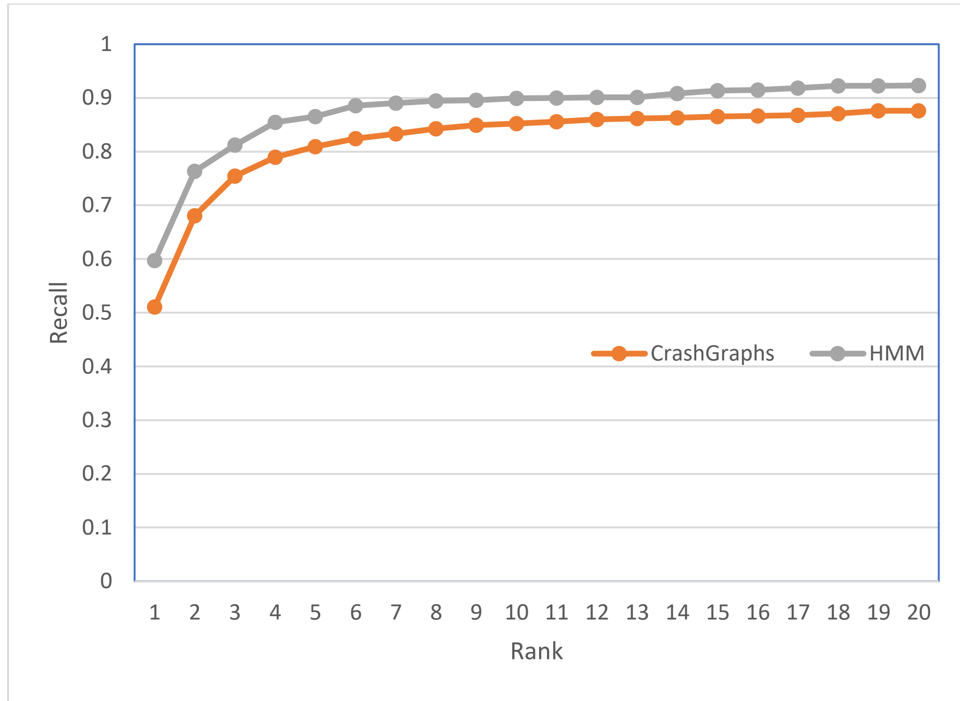
We compared our approach to Crash Graphs the way as for CrashAutomata. We applied Crash Graphs to duplicate bug report groups of Firefox and GNOME to predict the duplicate bug report group of an incoming stack trace. We used the same setting as for HMM. More precisely, for each DGi (whether it is for Firefox of GNOME), we used 70% of traces to construct a CrashCraph and 30% of traces to test it. Note that we did not use the validation set to validate Crash Graphs since Crash Graphs does not use any particular heuristics.

We compared Crash Graphs with HMM N=20 for Firefox, and HMM N=40 for GNOME

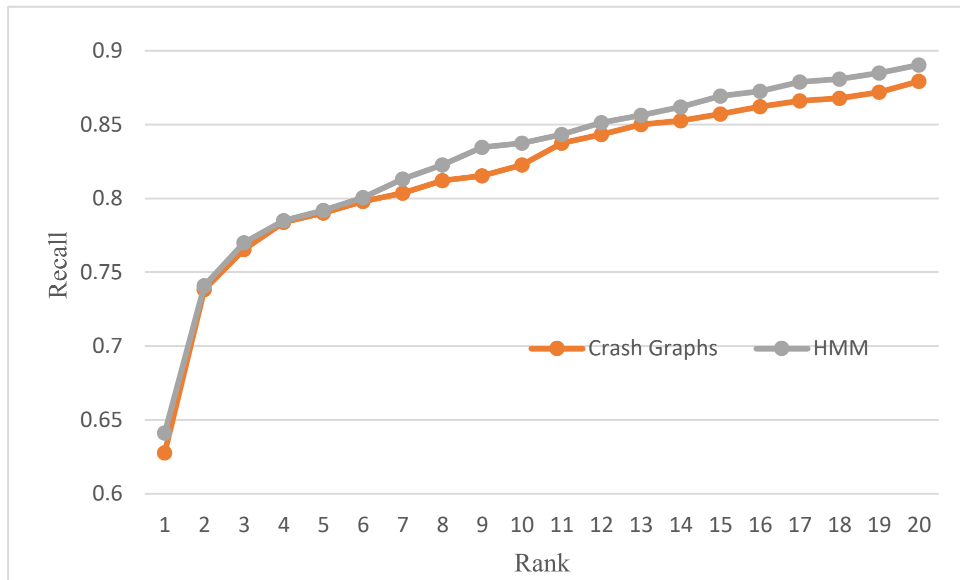
since these are the HMM models that provide the best accuracy. The results are shown in Figure 5.4.

The results show that HMM performs better than Crash Graphs when applied to the Firefox dataset. At Rank 1, our approach achieves a recall of 59.7% whereas Crash Graphs achieves a recall of 51%. This gap is maintained as k increases as shown in Figure 5.4 MAP of HMM is also better than the one obtained with Crash Graphs.

For GNOME, our HMM-based approach performs almost the same as Crash Graphs as shown in Figure 5.4 and Figure 5.5. This may be due to the fact that there are many more traces in GNOME than Firefox. Crash Graphs was able to build a representative graph that characterizes the traces of a duplicate bug report group.

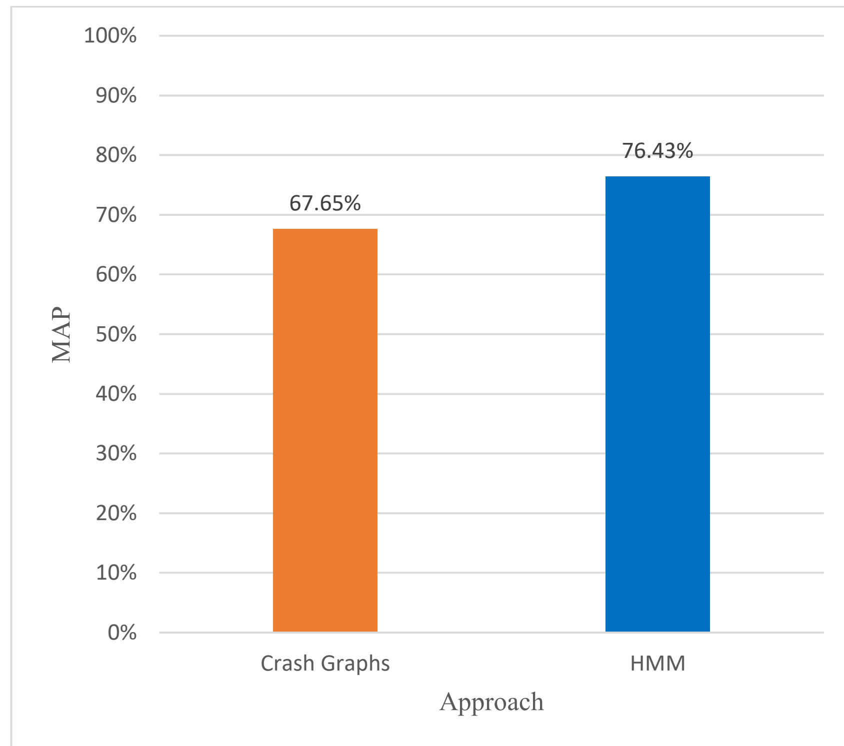


Firefox

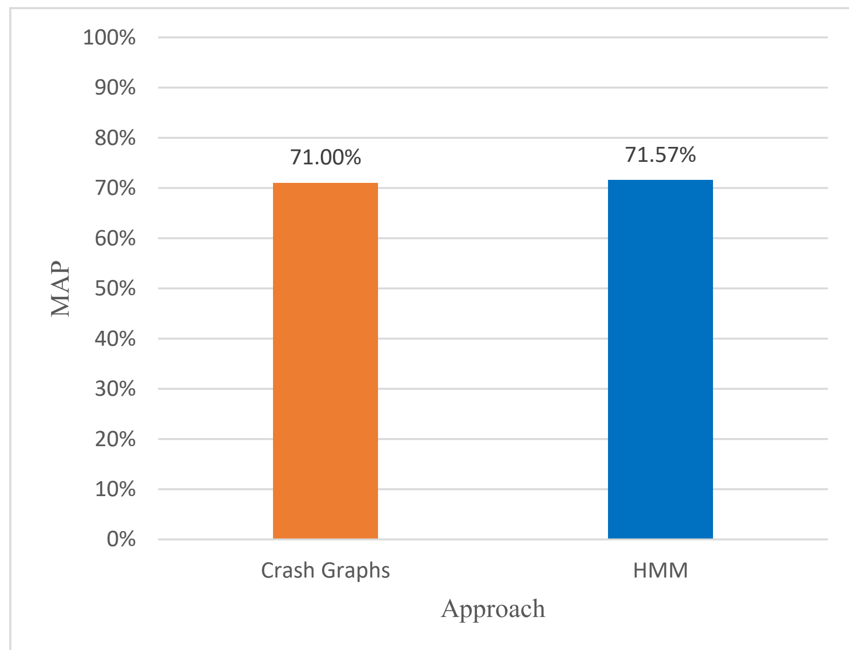


GNOME

Figure 5.4. Comparison of Recall Rate@k between Crash Graphs and HMM



Firefox



GNOME

Figure 5.5. Comparison of MAP between Crash Graphs and HMM

5.3.3 Discussion

Varying the number of hidden states: To train the HMM models, we varied the number of hidden states from 15 to 50 with bounds of 5. A different setting may lead to different results. However, our results suggest that the number of hidden states does not have a major impact on the overall approach. As we can see from the boxplots in Table 5.1 and Table 5.2, the recall changes slightly by varying the number of hidden states. Take for example the results obtained for Rank 1 for Firefox, the recall ranges from 57.91% to 59.81%, i.e., a 1.9% difference.

Impact on triaging effort: The MAP is an indication of how well a given approach ranks incoming bug reports (in our case a bug report is characterized by its stack trace). The average MAP across all HMMs is 76% for Firefox and 71% for GNOME. This means that, in general, our approach ranks well the incoming bug reports, which should reduce the time spent by triagers to find the right duplicate bug report group for an incoming bug report. The better the recall and MAP, the less effort is needed. Note that for both datasets, the gap between recall at Rank 1 and Rank 2 is significantly reduced when comparing recalls between the subsequent pairs of ranks (Rank 2 with Rank 3, Rank 3 with Rank 4, *etc.*). This suggests that MAP should be even higher if we ignore Rank 1, meaning that a triager would accept to examine at least two duplicate bug report groups to determine the right duplicate bug report group.

5.5. Chapter Summary

In this chapter, we presented a novel approach aimed at automatically detecting duplicate bug reports using stack traces and Hidden Markov Models. Based on this study and the one presented in the previous chapter we recognize the benefits we derive from using stack trace's information solely that we believe improves the detection accuracy of duplicate bug reports.

Our experiments highlight that with a list of rank-1 bug reports, recall values of 80% and 63% have been achieved on Firefox and GNOME datasets, respectively. With the same list of bug reports, our approach detects the duplication of a given report with an average MAP value of 87% and 71.5% on Firefox and GNOME datasets, respectively. It has also been observed that the higher the rank level, the higher the recall rate. For instance, the recall rate with a list of rank-2 has been about 12% higher than that with a list of rank-1.

Chapter 6. Towards a Deep learning Approach for Detecting Duplicate Bug Reports

A natural extension to our work is to explore the use of deep learning methods, combined with stack traces, for the detection of duplicate bug reports. This is also motivated by the work of Deshmukh et al. [21], who proposed a deep-learning-based approach for automatic duplicate bug detection, using bug report descriptions as features. In this chapter, we use stack traces. More particularly, the information inside stack traces are fed into two deep recurrent neural network models to automatically detect and assign each duplicate bug report to its corresponding bug report group. The models are the Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU), widely used in deep learning. To the best of our knowledge, this is the first time that deep learning techniques have been used to encode stack trace's information solely to automatically detect duplicates. By using bug reports collected from Firefox and GNOME repository, our approach can achieve a Recall Rate@k ranging from 86% to 98% for Firefox and 63% to 93% for GNOME.

6.1. Background

This section provides an overview of the deep recurrent neural network models used in this study.

6.1.1 Recurrent Neural Networks

Artificial Neural Networks (ANNs) are formed by layers to simulate the brain's processing. They are extensively used in a variety of applications such as image processing and forecasting.

Neural layers consist of smaller numerical interconnected components named nodes. The hidden nodes compute a weighted sum of inputs that are then passed through an activation function to produce the node's output value. In conventional neural networks, the inputs and outputs are independent of each other, but this may not work very well when the network processing involves the temporal sequencing of inputs such as predicting the next word in a sentence. In this case, it would be better to know which words came before. Recurrent Neural Networks (RNN) differ from conventional neural networks in that the hidden layers have recurrent links back to themselves. Moreover, an RNN model typically builds an internal memory (state) to process arbitrary sequences of inputs to achieve better prediction accuracy.

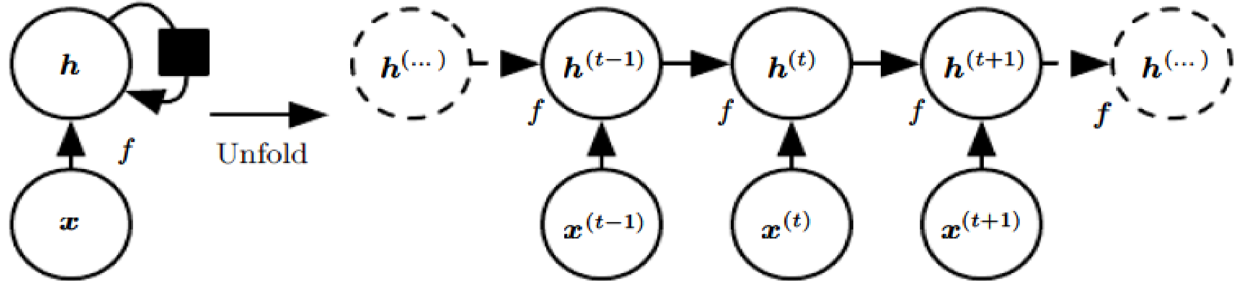


Figure 6.1. Unrolled recurrent neural network (taken from [87])

An example of an unrolled RNN, taken from [87], is shown in Figure 6.1. A typical RNN model has one input layer, one output layer, and at least one hidden layer. The input nodes are fed into a hidden layer through an activation function. The output of the hidden layer state at time t-1 is then fed back into the same hidden state for use at a later time t. Given an input sequence $x = (x_1, x_2, \dots, x_t)$, the RNN computes its recurrent hidden state h_t according to the following relationship:

$$h_t = f(Wx_t + Uh_{t-1})$$

where f is an activation function (the most common choices are the logistic sigmoid and the hyperbolic tangent), x_t is the current input, h_{t-1} is the previous calculated hidden state, W and U are the model parameters to be learned (weight matrices). It is worth noting that the dot product Wx_t extracts information from the current input; while the dot product Uh_{t-1} extracts information from the entire history of the inputs, and then combines them with the information collected from the current input to produce the output.

An RNN based model is typically trained using an appropriate objective function to ensure that the network input-output mapping satisfies the desired probabilistic interpretation. Depending on the considered objective function, the RNN model can either output a sequence $(y_1, y_2, \dots, y_k)$ or a single value y_k, where the output at each time step is calculated as follows [34]:

$$y_k = g(h_t)$$

In the above equation, the function $g(.)$ is a non-linear function (e.g., softmax) that can provide estimates of the class probabilities. Note that Softmax is an appropriate non-linearity for estimating posterior probabilities by ensuring that the values are non-negative and have a sum of one.

Recurrent networks are widely used in processing input sequences such as language modeling, handwriting recognition and generation, machine translation, speech recognition, video analysis and image captioning. In practice, however, they are challenging to train in capturing long-term dependencies due to the well-known gradient explosion or gradient vanishing problems. The former arises when gradients explode as the weights become more substantial and the norm of the

gradient increases substantially during training. Conversely, the latter refers to the exponential shrinking of gradients magnitude as they are propagated back through time. To overcome these problems, long short-term memory recurrent neural network models were proposed in the literature [88], [89]. Two RNN variants, namely the Long-Short-Term Memory (LSTM) and Gate Recurrent Unit (GRU) are used in this chapter.

6.1.2 LSTM

To address the issue of gradient vanishing in training RNNs, LSTM was first introduced in [90]. A central addition to the initial LSTM was the introduction of self-recurrent connections having a constant weight of 1.0 and serving as paths where the gradient can flow for long durations. An improvement was made to make the weights on these self-recurrent connections controlled by another hidden unit (gated) so that the time scale of integration can be changed dynamically.

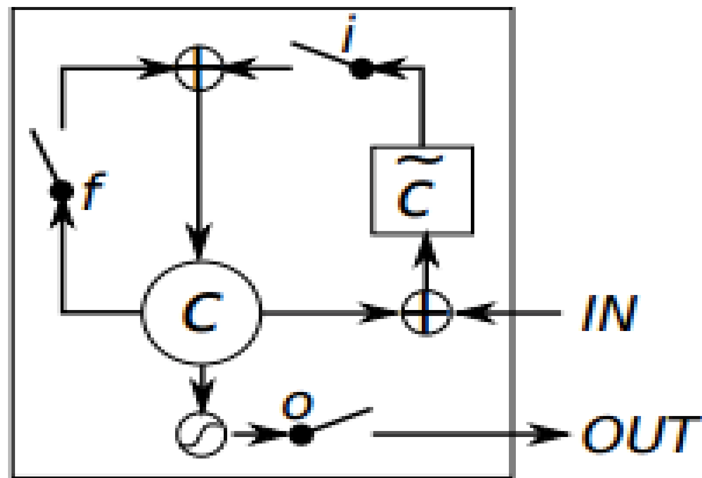


Figure 6.2. Long Short-Term Memory [91]

A common LSTM architecture comprises an input gate, an output gate and a forget gate which all regulate the flow of information inside the LSTM unit. The input gate is responsible for adding

information to the cell, the forget gate decides which information to keep and which ones are of less importance and no longer required and finally, the output gate selects the important information from the current cell and produces the output [88]. A typical LSTM unit is shown in Figure 6.2. At the time t, the input is x_t, the hidden output is h_t, and h_{t-1} is the previous time step output. Given the input sequence as $x = (x_1, x_2, \dots, x_t)$, hidden state of a memory cells as $h = (h_1, h_2, \dots, h_t)$ and output time series as $y = (y_1, y_2, \dots, y_t)$, each j-th LSTM unit performs following calculations for a memory c_t^j at time t [91]:

$$h_t^j = \mathrm{O}_t^j \tanh(c_t^j)$$

$$O_t^j = \sigma(W_o x_t + U_o h_{t-1} + V_o c_t)^j$$

$$c_t^j = f_t^j * c_{t-1}^j + i_t^j * \tilde{C}_t^j)$$

$$\tilde{c}_t^j = \tanh(W_c x_t + U_c h_{t-1})^j$$

$$f_t^j = \sigma(W_f x_t + U_f h_{t-1} + V_f c_{t-1})^j$$

$$i_t^j = \sigma(W_o x_t + U_o h_{t-1} + V_o c_t)^j$$

where i_t^j, f_t^j, and o_t^j are input, forget and output gates respectively. V_o is a diagonal matrix and $W$and U are parameters defining weight matrices. h_{t-1} is the hidden state at the previous time step and σ denotes a logistic sigmoid function. All three gates have similar equations by a different set of parameters. c_t^j and $\tilde{c}_t^j$ are existing memory cell and new memory content respectively. The output unit is computed by h_t.

6.1.3 GRU

The Gated Recurrent Network (GRU) is a variation of LSTM recurrent neural networks introduced by Kyunghyun Cho et al. (2014) [89]. The GRU network was designed to have more persistent memory, making it very suitable to capture long-term dependencies between elements of a sequence. Similar to LSTM, GRU adaptively updates or resets its memory content by using reset and update gates. The former has the ability to mitigate the past hidden state if it is irrelevant to the computation of the new state, whilst the latter conditionally determines how much of the past state should be passed forward to the next state. Note that the main difference with the LSTM is that the forget and input gates are combined into a single update gate, making its cells computationally more efficient than the standard LSTM.

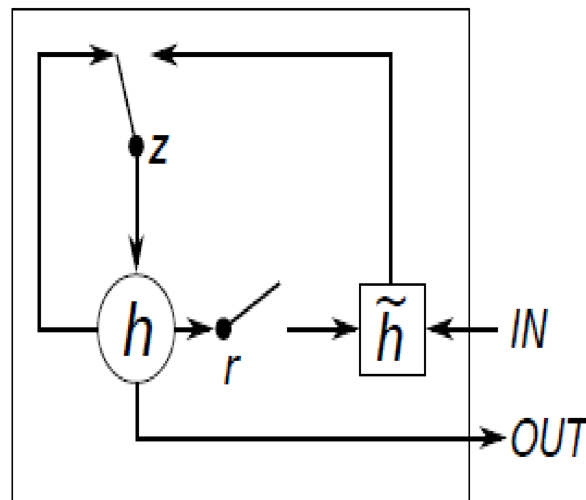


Figure 6.3. A Gated Recurrent Unit [91]

Figure 6.3 illustrates the components of a GRU cell. The GRU computes the following values:

$$z_t^j = \sigma(W_z x_t + U_z h_{t-1})^j$$

$$r_t^j = \sigma(W_r x_t + U_r h_{t-1})^j$$

$$\widetilde{h_t^j} = tanh(Wx_t + \mathrm{U}(r_t \odot h_{t-1}))^j$$

$$h_t^j = \left(1 - z_t^j\right) * h_t^j + z_t^j * \widetilde{h_t}$$

where z_t^j denote the update gate, that decides the number of activation updates by unit. r_t^j and $\widetilde{h_t^j}$ are the reset gate and new memory content respectively. Subsequently, the memory unit at a time step is denoted by h_t^j [91].

6.2. Approach

6.2.1 Building the Networks

We performed hyperparameter tuning and examined random models by combining LSTM and GRU layers to achieve the best model. Moreover, the layers in each model vary in different parameters such as the number of input and output layers, dropout and number of epochs. Our deep neural network models are composed of Embedding, GRU, LSTM and Fully Connected (Dense in Keras) layers.

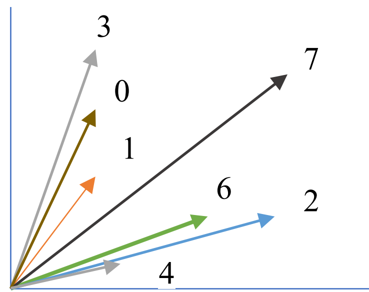


Figure 6.4. Example of the embedding converting IDs to vectors

Embedding maps the words (function IDs) to low dimensions and learns a dense vector for each word [92]. By learning a dense vector for each word, word embeddings are able to capture

semantic and syntactic properties. For example, assume we have two stack traces: T1: 1 2 4 5 3 3 and T2: 1 2 4 7 3 2 2 7. Embedding converts each ID to an N dimensional vectors in the space. In this example, we have 7 different IDs and the converted vectors are shown in Figure 6.4.

The vectors resulting from the embedding layer are passed to the next hidden layer and hidden layer pass the information to the neurons of the output layer in order to output the results as shown in Figure 6.5.

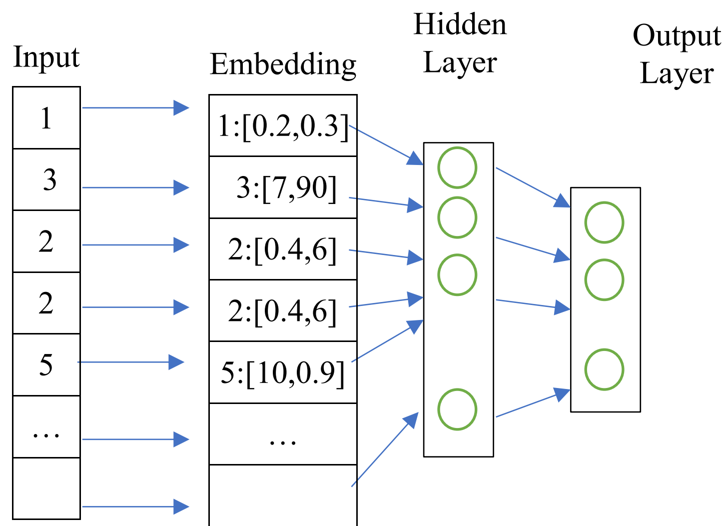


Figure 6.5. Process of passing information through the layers

Using this notion, we compute the similarity between the stack traces in each DG and stack traces of a new bug report. Dropout is a regularization technique for avoiding overfitting in neural networks [93]. In a dropout layer, all neurons with probability p and their related edges dropping out, so keeping neurons with probability $q = 1 - p$. This technique reduces the co-adaptation of neurons [94]. Dropout can add as a single layer and/or as a parameter applied on layers. Our proposed models involve dropout as well. The architecture of two samples for our generated networks for Firefox and GNOME are shown in Table 6.1 and Figure 6.6.

Table 6.1. The attributes of the combined model

Dataset	Embedding	GRU/LSTM	Dropout	Dense (FC)	fit_epochs
Firefox	256	2 GRU Layers	0.5	256	30
Gnome	128	1 GRU Layer	0.5	64	30

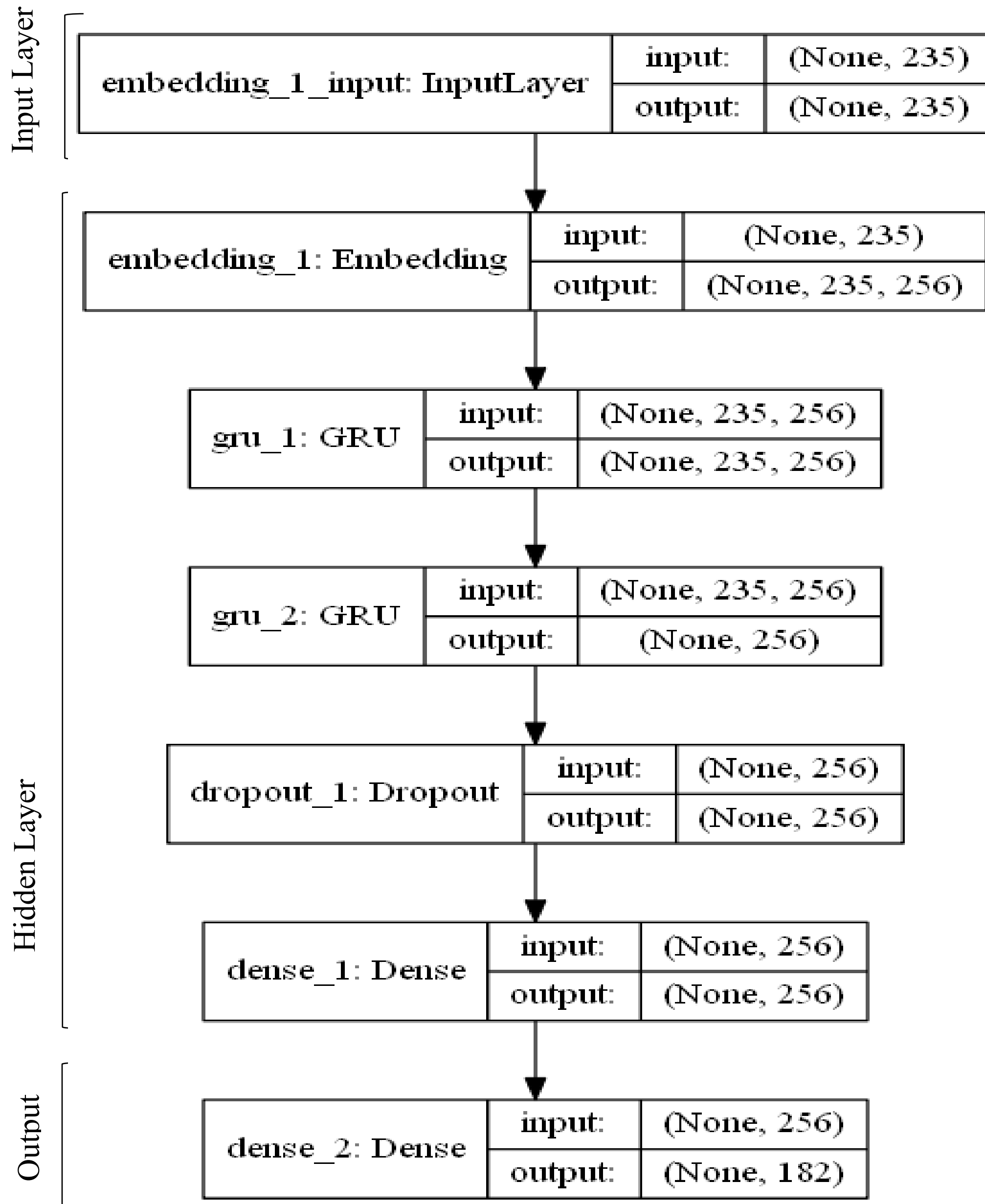


Figure 6.6. The selected network model for Firefox

The stack traces of function calls are assumed to be sequences of words in neural networks. Therefore, an Embedding layer converts word sequences (function sequences in our case) to dense vectors. The output of the Embedding layer is given to the next LSTM/GRU layer. The output of LSTM/GRU layers is then passed to the Dense or Fully Connected layers. Eventually, the Fully Connected layer outputs the classification results.

We used Keras[7] a TensorFlow based library to set up and train our neural networks. The number of neurons in the input layer corresponds to the number of distinct functions (7355 and 4019 distinct function names for Firefox and GNOME respectively) and the number of output neurons correspond to the number of classes (e.g., number of duplicate groups). The output is the ranked list of probabilities of the testing instances to the most relevant labels.

Training a neural network requires initializing many parameters known as hyperparameters. These parameters extensively affect the accuracy and cost function[8] of the network model. The cost function measures how good the neural network model can expect the outputs to be, regarding the input values [94]. In practice, the best way to find the most appropriate parameters is to perform some trials that tune the hyperparameters [87]. The goal is to train a model with a minimized cost function. In this case, a best model is one with highly efficient parameters in which the cost function is the closest, if not equal, to zero [94].

To achieve this hypothesized best model, we examined the proposed models by varying the number of layers as well as increasing the range of its hyperparameters. The trial models were composed of Embedding, LSTM, GRU and Dense layers. The Embedding layer converts positive

---

[1] https://keras.io/

[2] Sometimes refer to as loss function or objective function

integers to vectors of fixed size. We selected ReLU [95] and softmax [96] as the activation functions for Dense layers. The Softmax activation function is used to calculate the probability distribution over multiple classes at the output layer. A well-trained model should be learned in such a way that it assigns high similarity to similar duplicate groups and very low similarity to non-related duplicate groups; all at minimum cost. To configure the training process, the compile method defines 'categorical cross entropy' [97] as a cost function and 'Adam' [98] as an optimizer. Cross-entropy measures the probability of classification by a probability between 0 and 1. Models with a cost close to 0 are considered to be best models.

The number of neurons in each layer tests between 64 and 512. Indeed, for each of the LSTM, GRU, and Dropout layers, the dropout values were examined with assigned probabilities between 0.1 and 0.7. We chose this range because we found all our most successful results within it. Finally, all the chosen parameters were tested for epochs between 20 and 50.

For the last step, all results were evaluated according to the achieved accuracy on the validation set. **Table 6.1** describes the final observed models' parameters for the two datasets. Figure 6.6 illustrates the plotted models for the final select networks.

6.2.2 Testing Phase

When a new bug is found in the system, its stack trace is compared with all the duplicate bug groups' stack traces to detect if the stacks are duplicates or not. The Softmax function (mainly in the last fully-connected layer) outputs the duplicate predication values to rank the most relevant duplicate groups to the incoming stack trace.

We have evaluated our model by feeding the network with stack traces of both Firefox and

GNOME test data. Stack traces are a sequential ordering of all running functions at the crash time. We only considered stack traces as they reveal the behavior of the software at the time of the crash, as they are automatically generated and cannot be edited manually. The findings were then ranked to measure the accuracy and Recall Rate@k.

6.2.3 Evaluation Metrics

For both datasets, we applied our models to explore the most appropriate parameters for best accuracy. We used Recall Rate@k (described in the previous chapter) and accuracy as the evaluation metrics. Accuracy is a metric that the models calculate automatically as:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

6.3. Experimental Results and Comparison

We ran the models on Firefox and GNOME datasets and evaluated our models by ranking achieved Recall Rate@k results, from the top 1 to top k (see Table 6.2). As the accuracy level did not show a significant fluctuation after rank 20, we stopped our experiments at top k=20. Our findings in terms of accuracy, show that our models can achieve accuracy between 61.9% and 92.4% for the GNOME dataset and between 86% and 99.7% for the Firefox dataset.

Table 6.2. Results for Recall Rate@k for Firefox and GNOME datasets

Recall Rate		
Rank	**GNOME**	**Firefox**
1	63.47%	80.99%
2	75.00%	90.05%

3	77.44%	95.41%
4	79.55%	96.30%
5	81.25%	96.94%
6	82.31%	97.19%
7	83.77%	97.32%
8	84.90%	97.32%
9	86.04%	97.30%
10	86.77%	97.70%
11	87.58%	98.09%
12	88.31%	98.09%
13	88.96%	98.09%
14	89.77%	98.21%
15	90.42%	98.21%
16	90.99%	98.47%
17	91.64%	98.47%
18	92.13%	98.47%
19	92.78%	98.47%
20	92.94%	98.47%

Figure 6.7 and Figure 6.8 depict how accuracy @top-1 changes by increasing the number of epochs until reaches to an almost a fixed value. The stable epoch depends highly on the models and datasets. For example, on the model applied to Firefox the stable epoch is 17 while the stable epoch value for the model on GNOME dataset is 30. Indeed, Firefox obtained higher accuracy in all ranks compared to GNOME. One reason for this difference could be a result of the natures of their respective stack traces in duplicate groups. Whilst stack traces in Firefox have identical or similar signatures, which keep more similar stack traces together, the stack traces of duplicate groups in GNOME do not share signatures. The other reason for having higher accuracy in Firefox compared to GNOME could be the diversity of the duplicates in each duplicate group, causing more dissimilar stack traces.

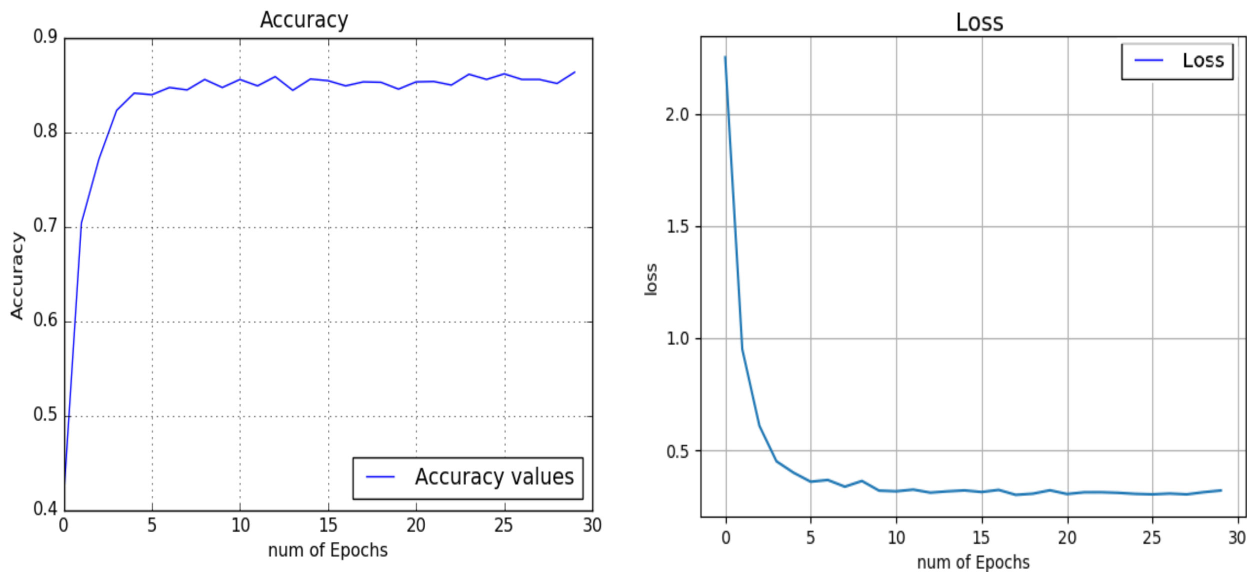


Figure 6.7. Accuracy and cost in Firefox model by number of epochs

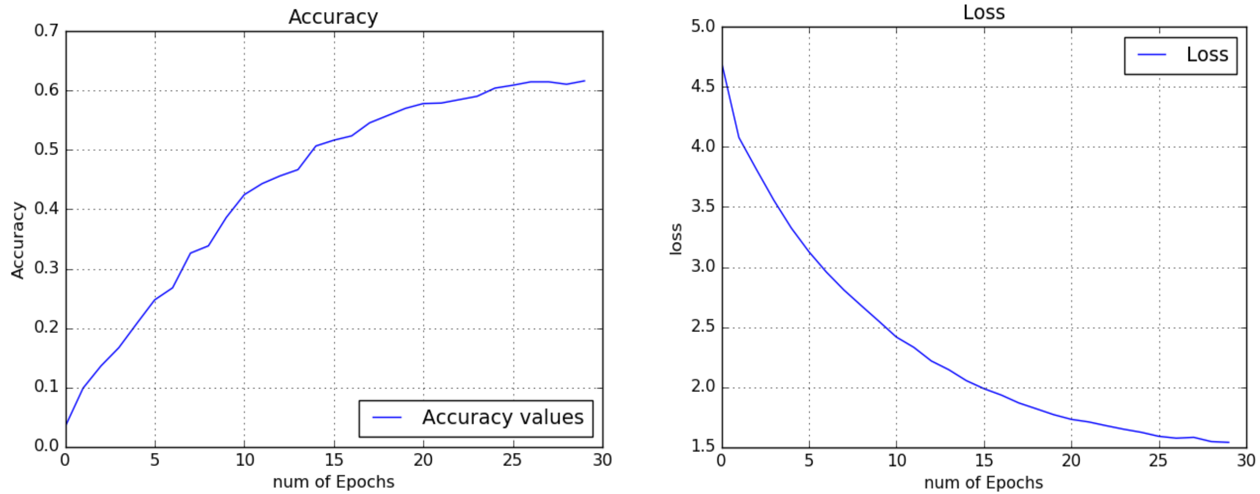


Figure 6.8. Train vs Test accuracy and cost in GNOME model by number of epochs

We compare our results with our HMM-based approach (Chapter 6) which implemented HMM on the same datasets. Figure 6.9 and Figure 6.10 illustrate the comparison of Recall for HMM and our deep neural network model with various ranks on Firefox and GNOME respectively. As we can see, from Rank 1 Firefox reaches the highest Recall of 86%, which improved the HMM result at this rank by almost 5%. The Recall level jumps to 97.32% at Rank 2, with another 4% HMM improvement. From Rank 3, the model's Recall jumps as high as 99.19%. This trend is comparable to GNOME dataset, in that the HMM and neural network model hit similar accuracy at Rank 1. However, at Rank 4 and higher, the NN model's Recall stays around 2% above the accuracy by HMM.

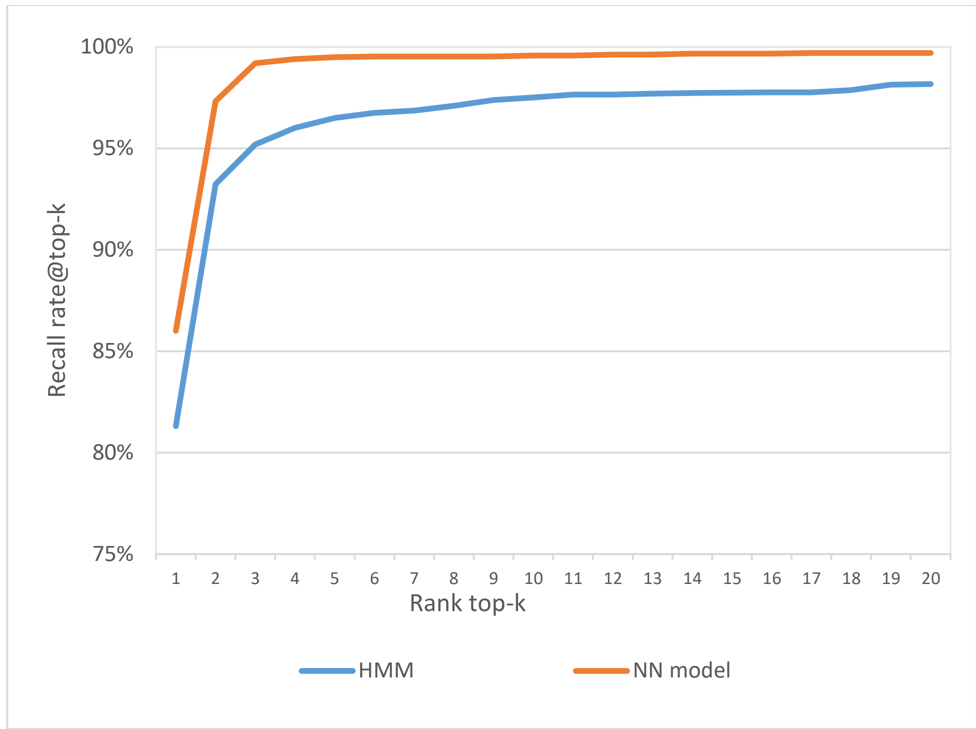


Figure 6.9. Firefox Recall Rate@k comparison of HMM and our deep NN model

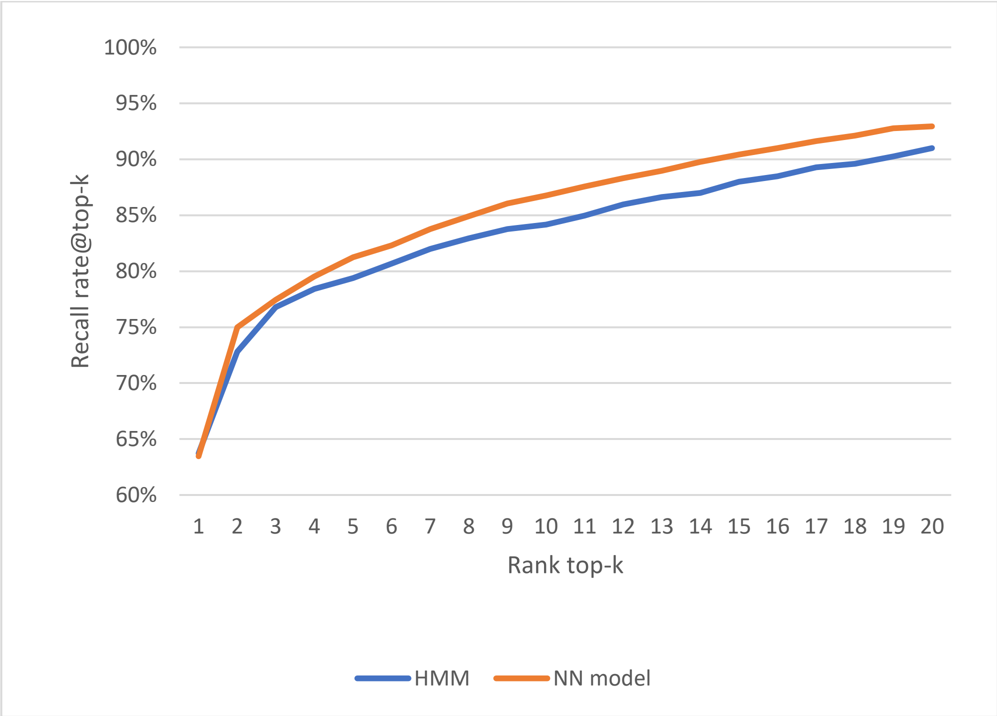


Figure 6.10. GNOME Recall Rate@k comparison of HMM and deep NN model

6.4. Discussion

In this chapter, we showed that using deep learning methods, namely a combination of LSTM and GRU, improves the results over the use of HMMs. The datasets were divided into training and testing sets. The number of available stack traces varies for each bug report; meaning that for each generated duplicate group we have different number of stack traces. To determine the best-fit parameters, multiple models were examined and evaluated with accuracy and Recall Rate@k metrics. Among all generated and tested models, we chose the one with the highest obtained accuracy.

Moreover, in neural network models, the amount of available training data plays an essential role in obtaining results. The results of the proposed neural network model show superior accuracy in almost all ranks for both underlying datasets comparing to the results from HMM. We theorize that this could be because of two main reasons:

- HMM's state transition depends on the current state only since no memory exists to keep track of the past.
- The number of hidden states in an HMM is an effective parameter, which must be defined by the user. Therefore, in the case of having long stack traces, there might exist unseen relations between stack traces which cannot be caught by HMM since only limited number of functions are taken for the HMM construction and the rest are eliminated.

We found that the required number of epochs to train the model is as low as 30. Compared to the work of Deshmukh et al. [21], who proposed a deep-learning-based approach for automatic

duplicate bug detection, using bug report descriptions as features, our proposed deep learning model reveals a higher accuracy even at the first ranks.

6.5. Chapter Summary

In this chapter, we proposed an approach that leverages stack traces and deep learning techniques (LSTM and GRU) to detect duplicate bug reports. When applied to the Firefox and GNOME bug report repository, our approach that combines LSTM and GRU, trained on stack traces, provides a very high accuracy.

In the future, we intend to build on this work by experimenting with more repositories. We also plan to extend our models to other bug report features such as a combination of stack traces and bug report descriptions and comments.

Chapter 7. Conclusions and Future Directions

In this chapter, we present the conclusions for this thesis and suggest potential avenues for future study.

7.1. Summary of Contributions

The main contributions of this thesis consist of a set of techniques for detecting duplicate bug reports with the objective is to help triaging teams and software developers in the provision of fixes. These techniques are based on machine learning and stack traces.

We start by generating new datasets of duplicate bug report groups with their associated stack traces from two large open source projects, Firefox and GNOME. These datasets will be made available online to other researchers and practitioners.

Then, we proposed an approach, called CrashAutomata, based on the concept of generalizable automata for detecting duplicate bug reports. CrashAutomata creates training models from stack traces of duplicate bug report groups by extracting varied-length n-grams and by varying a threshold α that controls the level of generalization of the automaton. The idea is to have a model that can be general enough to classify similar traces that were unseen during training.

The next contribution consists of a novel technique that is based on HMM, which achieved a very good accuracy. With a list of rank-1 bug reports, recall values of 60% and 63% have been achieved on Firefox and GNOME datasets, respectively. With the same list of bug reports, our HMM-based approach detects the duplication of a given report with an average MAP value of

76% and 71% on Firefox and GNOME datasets, respectively. It has also been observed that the higher the rank level, the higher the recall rate. For instance, the recall rate with a list of rank-2 has been about 12% higher than that with a list of rank-1.

The last contribution of this thesis explores the use of deep learning methods, combined with stack traces, for the problem of duplicate bug report detection. We showed that a combination of LSTM and GRU provides promising results.

Finally, we believe that these approaches are easily applicable to industrial datasets as long as the bug reports contain stack traces. The traces can be in any format such as JSON. Our algorithms take numerical IDs which correspond to functions in stack traces. Based on the industry type of data, needs, resource availability, training time and performance expectations, one can decide which approach (i.e., CrashAutomata, HMM, and deep learning method) to apply. There is also possibility to perform a test on a small set with all approaches to find out the algorithm that best fit the datasets.

7.2. Threats to Validity and Limitations

Our proposed approach and the conducted experiments are subject to threats to validity, namely external, internal, and construct validity.

7.2.1 Threats to External Validity

Our approach is evaluated against two open source datasets and tested on duplicate bug reports with stack traces. We need to apply our technique to more datasets (including those from industry). We also need to evaluate if it outperforms existing work and approaches that use other bug report features such bug report descriptions and comments.

7.2.2 Threats to Internal Validity

For CrashAutomata, a threat to internal validity exists in the way we chose α. A different value may lead to different results. Determining this threshold in advance is not an easy problem and even if we succeed to do so for a give system, the value may be different when using another system. We anticipate that a tool that support CrashAutomata should provide enough flexibility to users to experiment with different values of α to find out the one that fits best the datasets.

In the HMM-based approach, the way we set the parameters A and B, and the conditional probability matrices to construct HMMs could be a threat to internal validity. We used the validation set to set the bounds to optimize A and B. A different validation set could result in a different initialization, resulting in a different model. However, to our knowledge, there is no clear solution to this problem and most studies that use HMM follow random initialization of A and B and repeat this process several times until a satisfactory model is obtained.

The way we set the number of hidden states is another threat to validity. We followed the common practice of setting this number to a small number and then increase it with bounds of 5. Although different state numbers do not seem to bring much improvement in accuracy, there is always a possibility that a different number may lead to other models.

In all experiments, for Firefox, we used traces of Thread0 only. Considering all threads will require an extensive amount of time to train the models. However, for crashes due to hangs, the top few methods are more or less the same (with functions such as "wait") between crash reports even if the root causes of the crashes are different. This may cause the associated bug reports to end up in the same bug report groups. To address this issue, we should (a) examine in depth what the impact would be, and (b) consider including other threads. This said, considering all threads

may cause scalability problems. Therefore, a trade-off between precision and scalability should be investigated.

The Firefox results depend on the quality of the initial manual triage and the quality of the signatures. Errors in manual triage and incorrect signatures may affect our results.

Finally, another threat to internal validity is related to the way we collected our datasets including web crawling and parsing tools implemented to collect bug reports and extract stack traces. Datasets vary from time to time and need to be upgraded frequently in order to generate the appropriate machine learning models. Also, if the structure of the web pages change, the tools, crawlers and parser have to change as well.. To mitigate this threat, we verified our data multiple times.

7.2.3 Threats to Construct Validity

The construct validity shows how the used evaluation measures could reflect the performance of our predictive model. In this thesis, we used precision, recall, and Mean Average Precision, which are widely used in other studies to assess the accuracy of machine learning models with applications to the problem of duplicate bug report detection.

7.2.4 Limitations and Opportunities for Future Research

The main limitation of the studies presented in this thesis is that they rely on stack traces, which may not always be available. Many bug tracking systems do not automatically collect stack traces. It is up to the report submitters to copy and paste the trace in the bug report description field. As an example, only 10% of Eclipse bug reports described by Lerch et al. [30] contain stack traces. Mozilla keeps stack traces for only one year because of the cost of saving a large number

of stack traces. Nevertheless, we believe that an approach that uses stack traces remains very useful, especially because stack traces are needed for other tasks such as bug reproduction [31], [56], [63], [64], [70], [99], fault localization[60], [76], [99]–[104], and bug prioritization [2], [10], [51], [59], [105]. An opportunity for future work is to design better bug tracking systems that collect automatically traces to allow for post-mortem analysis. The traces should contain useful information that characterize the crash.

Currently, our approach does not deal with new incoming bug reports that do not have prior duplicates in the database. We need to improve the approach to consider the creation of new groups on the fly. One possibility to do this is to determine a threshold below which an incoming bug report is deemed dissimilar enough to all existing bug reports and hence should be put in a new group that needs to be created. This threshold can be determined during the validation step of our approach using a validation set that contains a mix of duplicate bug reports and new bug reports (bug reports without prior duplicates).

In addition, techniques that work on stack traces are known to suffer from scalability issues. Building automata, HMMs, and deep learning models using stack traces may turn to be ineffective when applied to very large systems with a large number of duplicates. We need to investigate more scalable techniques by reducing the bug report space using heuristics and by developing parallel and distributed versions of the proposed algorithms. We also need to examine how traces can be reduced in size while keeping their main information. For this, studies in the field of trace abstraction and simplification (e.g., [106], [107]) may be useful.

Finally, we need to conduct user studies with developers in order to gather their feedback on the quality of the duplicate bug report detection techniques proposed in this thesis. The feedback

obtained could help to improve the parameter tuning of the proposed approaches.

7.4. Closing Remarks

Handling bug reports is a challenging task in software organization with large client base. In this thesis, we proposed many approaches to help bug report triaging teams detect duplicate bug reports upon their arrival to the system, hence reducing the required efforts to analyze bug reports. We chose to focus on techniques that operate on stack traces to overcome the imprecision of bug report descriptions. Our experiments demonstrate that the proposed techniques achieve good accuracy when applied to bug reports of two open source projects. By doing so, this thesis contributes to literature on software debugging and maintenance systems. We sincerely hope that practitioners and researchers can benefit from the work presented in this thesis.

Bibliography

[1] S. S. Yau, J. S. Collofello, and T. MacGregor, "Ripple effect analysis of software maintenance," in *The IEEE Computer Society's Second International Computer Software and Applications Conference, 1978. COMPSAC'78.*, 1978, pp. 60–65.

[2] F. Khomh, B. Chan, Y. Zou, and A. E. Hassan, "An entropy evaluation approach for triaging field crashes: A case study of Mozilla Firefox," in *Proceedings - Working Conference on Reverse Engineering, WCRE*, 2011, pp. 261–270.

[3] S. Wang, F. Khomh, and Y. Zou, "Improving bug localization using correlations in crash reports," in *IEEE International Working Conference on Mining Software Repositories*, 2013, pp. 247–256.

[4] J. Anvik, L. Hiew, and G. C. Murphy, "Coping with an open bug repository," *2005 OOPSLA Workshop on Eclipse Technology eXchange, eclipse'05, October 16, 2005 - October 17, 2005*, pp. 35–39, 2005.

[5] Z. Gu, E. T. Barr, D. J. Hamilton, and Z. Su, "Has the bug really been fixed?," in *2010 ACM/IEEE 32nd International Conference on Software Engineering*, 2010, vol. 1, pp. 55–64.

[6] K. Aggarwal, T. Rutgers, F. Timbers, A. Hindle, R. Greiner, and E. Stroulia, "Detecting Duplicate Bug Reports with Software Engineering Domain Knowledge," *Journal of Software: Evolution and Process*, vol. 29, no. 3, 2017.

[7] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim, "Duplicate bug reports considered

harmful... Really?," in *IEEE International Conference on Software Maintenance, ICSM*, 2008, pp. 337–345.

[8] E. Shihab *et al.*, "Predicting re-opened bugs: A case study on the Eclipse project," in *Proceedings - Working Conference on Reverse Engineering, WCRE*, 2010, pp. 249–258.

[9] A. Tsuruda, Y. Manabe, and M. Aritsugi, "Can We Detect Bug Report Duplication with Unfinished Bug Reports ?," in *2015 Asia-Pacific Software Engineering Conference Can*, 2015, pp. 151–158.

[10] S. Kim, T. Zimmermann, and N. Nagappan, "Crash graphs: An aggregated view of multiple crashes to improve crash triage," in *Proceedings of the International Conference on Dependable Systems and Networks*, 2011, pp. 486–493.

[11] A. Alipour, A. Hindle, and E. Stroulia, "A contextual approach towards more accurate duplicate bug report detection," in *IEEE International Working Conference on Mining Software Repositories*, 2013, pp. 183–192.

[12] M. J. Lin, C. Z. Yang, C. Y. Lee, and C. C. Chen, "Enhancements for duplication detection in bug reports with manifold correlation features," *Journal of Systems and Software*, pp. 1–11, 2014.

[13] Y. C. Cavalcanti, P. A. da Mota Silveira Neto, D. Lucrédio, T. Vale, E. S. de Almeida, and S. R. de Lemos Meira, "The bug report duplication problem: An exploratory study," *Software Quality Journal*, vol. 21, no. 1, pp. 39–66, 2013.

[14] A. Lazar, S. Ritchey, and B. Sharif, "Improving the accuracy of duplicate bug report

detection using textual similarity measures," *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*, pp. 308–311, 2014.

[15] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo, "A discriminative model approach for accurate duplicate bug report retrieval," in *2010 ACM/IEEE 32nd International Conference on Software Engineering*, 2010, vol. 1, pp. 45–54.

[16] B. S. Neysiani and S. M. Babamir, "Improving Performance of Automatic Duplicate Bug Reports Detection using Longest Common Sequence," *2019 5th Conference on Knowledge Based Engineering and Innovation (KBEI)*, pp. 378–383, 2019.

[17] L. Poddar, L. Neves, W. Brende, L. Marujo, S. Tulyakov, and P. Karuturi, "Train One Get One Free: Partially Supervised Neural Network for Bug Report Duplicate Detection and Clustering," *arXiv preprint arXiv:1903.12431*, 2019.

[18] O. Chaparro, J. M. Florez, U. Singh, and A. Marcus, "Reformulating Queries for Duplicate Bug Report Detection," *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 218–229, 2019.

[19] A. Hindle and C. Onuczko, *Preventing duplicate bug reports by continuously querying bug reports*. Empirical Software Engineering, 2019.

[20] C. Olah, "Understanding LSTM Networks -- colah's blog." [Online]. Available: https://colah.github.io/posts/2015-08-Understanding-LSTMs/. [Accessed: 26-Sep-2018].

[21] J. Deshmukh, K. M. Annervaz, S. Podder, S. Sengupta, and N. Dubash, "Towards Accurate Duplicate Bug Retrieval using Deep Learning Techniques," in *2017 IEEE International*

Conference on Software Maintenance and Evolution, 2017, pp. 115–124.

[22] A. Budhiraja, “Towards Word Embeddings for Improved Duplicate Bug Report Retrieval in Software Repositories,” in *Proceedings of the 2018 ACM SIGIR International Conference on Theory of Information Retrieval*, 2018, pp. 167–170.

[23] N. Jalbert and W. Weimer, “Automated duplicate detection for bug tracking systems,” in *Proceedings of the International Conference on Dependable Systems and Networks*, 2008, pp. 52–61.

[24] M. S. Rakha, W. Shang, and A. E. Hassan, “Studying the needed effort for identifying duplicate issues,” *Empirical Software Engineering*, vol. 21, no. 5, pp. 1960–1989, 2016.

[25] P. Runeson, M. Alexandersson, and O. Nyholm, “Detection of duplicate defect reports using natural language processing,” in *Proceedings - International Conference on Software Engineering*, 2007, pp. 499–508.

[26] A. Sureka and P. Jalote, “Detecting duplicate bug report using character N-gram-based features,” in *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*, 2010, pp. 366–374.

[27] A. T. Nguyen, T. T. T. N. Nguyen, D. Lo, and C. Sun, “Duplicate bug report detection with a combination of information retrieval and topic modeling,” in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering - ASE 2012*, 2012, p. 70.

[28] C. C. Yguarata, E. Santana De Almeida, C. E. Santana De Almeida, D. Lucredio, and S.

Romero de Lemos Meira, “An Initial Study on the Bug Report Duplication Problem,” 2010, pp. 2–5.

[29] K. Koochekian Sabor, A. Hamou-lhadj, and A. Larsson, “DURFEX : A Feature Extraction Technique for Efficient Detection of Duplicate Bug Reports,” in *2017 IEEE International Conference on Software Quality, Reliability and Security*, 2017, pp. 240–250.

[30] J. Lerch and M. Mezini, “Finding Duplicates of Your Yet Unwritten Bug Report,” in *2013 17th European Conference on Software Maintenance and Reengineering*, 2013, pp. 69–78.

[31] N. Chen and S. Kim, “Star: Stack trace based automatic crash reproduction via symbolic execution,” *PhD Thesis, Honk Kong University of Science and Technology*, vol. 41, no. 2, pp. 198–220, 2015.

[32] I. Burnstein, *Practical Software Testing: A Process-Oriented Approach*. Springer Science & Business Media, 2006.

[33] International Requirements Engineering Board (IREB) and M. Glinz, “A Glossary of Requirements Engineering Terminology,” *Standard Glossary for the Certified Professional for Requirements Engineering (CPRE) Studies and Exam*, no. May, p. 116, 2014.

[34] J. Radatz, “IEEE Standard Glossary of Software Engineering Terminology,” *IEEE Std 610.12-1990*, pp. 1–84, Dec. 1990.

[35] “Wikipedia.” [Online]. Available: https://en.wikipedia.org/wiki/Main_Page.

[36] O. Chaparro *et al.*, “Detecting missing information in bug descriptions,” in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 396–407.

[37] L. R. Rabiner, *A tutorial on hidden Markov models and selected applications in speech recognition*, vol. 77. 1989.

[38] N. Ebrahimi, A. Trabelsi, M. S. Islam, A. Hamou-Lhadj, and K. Khanmohammadi, "An HMM-based approach for automatic detection and classification of duplicate bug reports," *Information and Software Technology*, 2019.

[39] N. Ebrahimi Koopaei, M. S. Islam, A. Hamou-Lhadj, and M. Hamdaqa, "An Effective Method for Detecting Duplicate Crash Reports Using Crash Traces and Hidden Markov Models," *Proceedings of the 26th Annual International Conference on Computer Science and Software Engineering*, pp. 75–84, 2016.

[40] N. Ebrahimi Koopaei and A. Hamou-Lhadj, "CrashAutomata: An Approach for the Detection of Duplicate Crash Reports Based on Generalizable Automata," *Proceedings of the 25th Annual International Conference on Computer Science and Software Engineering*, pp. 201–210, 2015.

[41] K. Khanmohammadi, N. Ebrahimi, A. Hamou-Lhadj, and R. Khoury, "Empirical study of android repackaged applications," *Empirical Software Engineering*, pp. 1–43, 2019.

[42] H. Kagdi, M. L. Collard, and J. I. Maletic, "A survey and taxonomy of approaches for mining software repositories in the context of software evolution," *Journal of software maintenance and evolution: Research and practice*, vol. 19, no. 2, pp. 77–131, 2007.

[43] T. Xie, T. Zimmermann, and A. van Deursen, "Introduction to the special issue on mining software repositories." Springer, 2013.

[44] C. Sun, D. Lo, S. C. Khoo, and J. Jiang, "Towards more accurate retrieval of duplicate bug reports," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering, ASE 2011, Proceedings*, 2011, pp. 253–262.

[45] G. Jeong, S. Kim, and T. Zimmermann, "Improving Bug Triage with Bug Tossing Graphs," in *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 2009, pp. 111–120.

[46] "Life Cycle of a Bug." [Online]. Available: https://www.bugzilla.org/docs/4.4/en/html/lifecycle.html. [Accessed: 23-Jul-2018].

[47] T. Koponen, "Life cycle of defects in open source software projects," in *IFIP International Conference on Open Source Systems*, 2006, pp. 195–200.

[48] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?," in *Proceeding of the 28th international conference on Software engineering - ICSE '06*, 2006, vol. 2006, p. 361.

[49] G. Cuevas, "Managing the software process with a software process improvement tool in a small enterprise," in *Journal of Software-Evolution and Process*, 2012, vol. 24(5), no. July 2010, pp. 481–491.

[50] J. Zhou, H. Zhang, and D. Lo, "Where Should the Bugs Be Fixed ?," in *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*, 2012, pp. 14–24.

[51] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals, "Predicting the severity of a reported bug," in *Proceedings - International Conference on Software Engineering*, 2010, pp. 1–10.

[52] "Bugzilla bug life cycle." [Online]. Available: https://www.researchgate.net/figure/Bugzilla-bug-life-cycle-work-flow-taken-from-Bugzilla-manual_fig1_221500961.

[53] K. Koochekian Saboor, M. Hamdaqa, and A. Hamou-Lhadj, "Automatic Prediction of the Severity of Bugs Using Stack Traces and Categorical Features," *Elsevier Journal of Information and Software Technology (IST)*, 2019.

[54] K. Koochekian Saboor, M. Hamdaqa, and A. Hamou-Lhadj, "Automatic Prediction of the Severity of Bugs Using Stack Traces," in *IBM 26th Annual International Conference on Computer Science and Software Engineering (CASCON)*, 2016, pp. 96–105.

[55] K. Koochekian Sabor, M. Hamdaqa, and A. Hamou-lhadj, "Predicting Bug Report Fields Using Stack Traces and Categorical Attributes," in *IBM 29th Annual International Conference on Computer Science and Software Engineering (CASCON)*, 2019.

[56] M. Nayrolles, A. Hamou-Lhadj, S. Tahar, and A. Larsson, "JCHARMING: A bug reproduction approach using crash traces and directed model checking," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2015, pp. 101–110.

[57] A. Schröter, N. Bettenburg, and R. Premraj, "Do stack traces help developers fix bugs?," in *Proceedings - International Conference on Software Engineering*, 2010, pp. 118–121.

[58] S. Just, R. Premraj, and T. Zimmermann, "Towards the next generation of bug tracking systems," in *Proceedings - 2008 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2008*, 2008, pp. 82–85.

[59] D. Kim, X. Wang, S. Kim, A. Zeller, S. C. Cheung, and S. Park, “Which crashes should i fix first?: Predicting top crashes at an early stage to prioritize debugging efforts,” *IEEE Transactions on Software Engineering*, vol. 37, no. 3, pp. 430–447, 2011.

[60] Y. Gu *et al.*, “Does the Fault Reside in a Stack Trace? Assisting Crash Localization by Predicting Crashing Fault Residence,” *Journal of Systems and Software*, vol. 148, pp. 88–104, 2018.

[61] J. Steven, P. Chandra, B. Fleck, and A. Podgurski, *jRapture: A capture/replay tool for observation-based testing*, vol. 25, no. 5. ACM, 2000.

[62] S. Narayanasamy, G. Pokam, and B. Calder, “Bugnet: Continuously recording program execution for deterministic replay debugging,” in *ACM SIGARCH Computer Architecture News*, 2005, vol. 33, no. 2, pp. 284–295.

[63] S. Artzi, S. Kim, and M. D. Ernst, “Recrash: Making software failures reproducible by preserving object states,” in *European conference on object-oriented programming*, 2008, pp. 542–565.

[64] W. Jin and A. Orso, “BugRedux: Reproducing field failures for in-house debugging,” in *Proceedings - International Conference on Software Engineering*, 2012, pp. 474–484.

[65] J. Rößler, A. Zeller, G. Fraser, C. Zamfir, and G. Candea, “Reconstructing core dumps,” in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, 2013, pp. 114–123.

[66] H. Jaygarl, S. Kim, T. Xie, and C. K. Chang, “OCAT: object capture-based automated

testing," in *Proceedings of the 19th international symposium on Software testing and analysis*, 2010, pp. 159–170.

[67] N. Kaushik and L. Tahvildari, "A comparative study of the performance of IR models on duplicate bug detection," in *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR*, 2012, pp. 159–168.

[68] Y. Tian, C. Sun, and D. Lo, "Improved duplicate bug report identification," in *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR*, 2012, pp. 385–390.

[69] X. Wang, D. Lo, J. Jiang, L. Zhang, and H. Mei, "Extracting Paraphrases of Technical Terms from Noisy Parallel Software Corpora," in *Proceedings of the ACL-IJCNLP 2009 Conference Short Papers*, 2009, no. August, pp. 197–200.

[70] M. Nayrolles, A. Hamou-Lhadj, S. Tahar, and A. Larsson, "A bug reproduction approach based on directed model checking and crash traces," *Journal of Software: Evolution and Process*, vol. 29, no. 3, p. e1789, 2017.

[71] X. W. X. Wang, L. Z. L. Zhang, T. X. T. Xie, J. Anvik, and J. S. J. Sun, "An approach to detecting duplicate bug reports using natural language and execution information," in *2008 ACM/IEEE 30th International Conference on Software Engineering*, 2008.

[72] A. Lazar, S. Ritchey, and B. Sharif, "Generating duplicate bug datasets," in *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*, 2014, pp. 392–395.

[73] Y. Dang, R. Wu, H. Zhang, D. Zhang, and P. Nobel, “ReBucket: A method for clustering duplicate crash reports based on call stack similarity,” in *Proceedings - International Conference on Software Engineering*, 2012, pp. 1084–1093.

[74] M. Castelluccio, C. Sansone, L. Verdoliva, and G. Poggi, “Automatically analyzing groups of crashes for finding correlations,” in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2017*, 2017, pp. 717–726.

[75] P. K. Novak, N. Lavrač, and G. I. Webb, “Supervised descriptive rule discovery: A unifying survey of contrast set, emerging pattern and subgroup mining,” *Journal of Machine Learning Research*, vol. 10, no. Feb, pp. 377–403, 2009.

[76] T. Dhaliwal, F. Khomh, and Y. Zou, “Classifying field crash reports for fixing bugs: A case study of Mozilla Firefox,” in *IEEE International Conference on Software Maintenance, ICSM*, 2011, no. November 2009, pp. 333–342.

[77] M. S. Rakha, C. P. Bezemer, and A. E. Hassan, “Revisiting the Performance Evaluation of Automated Approaches for the Retrieval of Duplicate Issue Reports,” *IEEE Transactions on Software Engineering*, vol. 5589, no. c, pp. 1–27, 2017.

[78] J. Anvik, “Automating bug report assignment,” in *Proceedings of the 28th international conference on Software engineering*, 2006, vol. Shanghai, pp. 937–940.

[79] J. Xie, M. Zhou, and A. Mockus, “Impact of triage: a study of mozilla and gnome,” in *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2013, pp. 247–250.

[80] G. Jiang, H. Chen, C. Ungureanu, and K. Yoshihira, *Trace analysis for fault detection for application server*. CRC Press, 2007.

[81] D. M. . Powers, “Evaluation: From Precision, Recall and F-Measure to ROC, Informedness, Markedness & Correlation,” *Journal of Machine Learning Technologies*, vol. 2, no. 1, pp. 37–63, 2011.

[82] I. H. Witten, F. Eibe, and M. A. Hall, *Data Mining: Practical Machine Learning Tools and Techniques*, 3rd ed. Morgan Kaufmann, 2011.

[83] S. R. Eddy, “Hidden Markov models,” *Current Opinion in Structural Biology*, vol. 6, no. 3, pp. 361–365, 1996.

[84] E. Onegin, “Hidden Markov Models,” no. Chapter 18, pp. 1–21, 2014.

[85] Y. Bengio, “Markovian Models for Sequential Data,” *Neural Computing Surveys*, vol. 2, pp. 129--162, 1996.

[86] M. Bicego, V. Murino, and M. A. T. Figueiredo, “Similarity-Based Clustering of Sequences Using Hidden Markov Models,” *Machine Learning and Data Mining in Pattern Recognition*, pp. 86–95, 2003.

[87] I. Goodfellow, Y. Bengio, and A. Courville, “The Deep Learning Book,” *MIT Press*, vol. 521, no. 7553, p. 785, 2017.

[88] S. Hochreiter and J. Schmidhuber, “Long Short-Term Memory,” *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[89] K. Cho *et al.*, “Learning phrase representations using RNN encoder-decoder for statistical

machine translation," *arXiv preprint arXiv:1406.1078*, 2014.

[90] J. Gray, "Why Do Computers Stop and What Can Be Done About It?," in *Symposium on reliability in distributed software and database systems*, 1986, pp. 3–12.

[91] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling," in *NIPS 2014 Deep Learning and Representation Learning Workshop*, 2016, vol. 2015-Augus, no. September, pp. 4–8.

[92] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[93] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A Simple Way to Prevent Neural Networks from Overfitting," *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 2014.

[94] M. Nielsen, *Neural Networks and Deep Learning*, 25th ed. San Francisco, CA, USA: Determination press, 2015.

[95] V. Nair and G. E. Hinton, "Rectified Linear Units Improve Restricted Boltzmann Machines," in *Proceedings of the 27th International Conference on Machine Learning*, 2010, no. 3, pp. 807–814.

[96] C. M. Bishop, "Pattern recognition and machine learning (information science and statistics) springer-verlag new york," *Inc. Secaucus, NJ, USA*, 2006.

[97] P.-T. De Boer, D. P. Kroese, S. Mannor, and R. Y. Rubinstein, "A tutorial on the cross-entropy method," *Annals of operations research*, vol. 134, no. 1, pp. 19–67, 2005.

[98] D. P. Kingma and J. Ba, “Adam: A Method for Stochastic Optimization,” *arXiv preprint arXiv:1412.6980*, Dec. 2014.

[99] W. Jin and A. Orso, “F3: Fault Localization for Field Failures,” in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, 2013, pp. 213–223.

[100] L. Gong, H. Zhang, H. Seo, and S. Kim, “Locating Crashing Faults based on Crash Stack Traces,” *arXiv preprint arXiv:1404.4100*, p. 10, 2011.

[101] R. Wu, H. Zhang, S.-C. Cheung, and S. Kim, “CrashLocator: locating crashing faults based on crash stacks,” in *Proceedings of the 2014 International Symposium on Software Testing and Analysis - ISSTA 2014*, 2014, pp. 204–214.

[102] L. Moreno, J. J. Treadway, A. Marcus, and W. Shen, “On the use of stack traces to improve text retrieval-based bug localization,” in *Proceedings - 30th International Conference on Software Maintenance and Evolution, ICSME 2014*, 2014, pp. 151–160.

[103] a. J. Jones and M. J. Harrold, “Empirical evaluation of the tarantula automatic fault-localization technique,” *Automated Software Engineering*, pp. 282–292, 2005.

[104] C. Liu and J. Han, “Failure proximity: a fault localization-based approach,” in *Proceedings of the 14th ACM SIGSOFT international …*, 2006, pp. 46–56.

[105] A. Podgurski *et al.*, “Automated support for classifying software failure reports,” in *25th International Conference on Software Engineering, 2003. Proceedings.*, 2003, vol. 6, pp. 465–475.

[106] A. Hamou-Lhadj and T. C. Lethbridge, “Compression techniques to simplify the analysis

of large execution traces," in *Proceedings 10th International Workshop on Program Comprehension*, 2002, pp. 159–168.

[107] A. Hamou-lhadj, "Techniques to Simplify the Analysis of Execution Traces for Program Comprehension," University of Ottawa, 2006.