POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

ALGORITHMS AND LEARNING MODELS FOR BUG REPORT DEDUPLICATION

IRVING MULLER RODRIGUES
Département de génie informatique et génie logiciel

Thèse présentée en vue de l'obtention du diplôme de *Philosophiæ Doctor*
Génie informatique

Mai 2022

ALGORITHMS AND LEARNING MODELS FOR BUG REPORT DEDUPLICATION

DEDICATION

To my wife, Karoline,
my parents, Mario and Gerda,
and my sister, Ingrid,
I love you...

ACKNOWLEDGEMENTS

My most sincere appreciation goes to Professors Daniel Aloise and Michel Dagenais. I am truly grateful for the opportunity to work under your supervision. Thank you for all the support, teaching, and advice you have provided over the last few years. It has been a great honor to be mentored by you.

I want to express my deepest thanks to Eraldo Rezende Fernandes. Thank you for all the valuable teaching and guidance since my graduation. I fell deeply honored to have worked with you during these years.

A special thanks to my wife Karoline, my parents Mario and Gerda, and my sister Ingrid. It's hard to put into words how eternally grateful I am for all the support, encouragement and sacrifice you made for me. Thank you so much for everything. You are the real MVP!

To all my friends, thank you for always supporting me and for all the great laughter and memories shared. Your company always made my life brighter and lighter. To my labmates and colleagues, thank you for all your help, shared ideas, and advice. It was an immense pleasure to work with you.

I am honored to have the chance to work and collaborate with: Dmitrij Koznov, George Chernishev, Aleksandr Khvorov, Roman Vasiliev, Dmitry Luciv, and Nikita Povarov. Moreover, I extend my gratitude to Professors Michel Gagnon, Foutse Khomh, and Bram Adams for accepting the invitation to compose my thesis jury.

Also, my thanks go to Compute Canada and regional partner organizations for providing the computer resources that were crucial for my articles presented in this thesis.

RÉSUMÉ

Dans les projets logiciels, une pratique courante consiste à utiliser des système de suivi des bugs (BTSs) afin de gérer et suivre les enregistrements de bogues. Une tâche cruciale pour les BTS consiste à identifier si un nouveau rapport décrit un bogue qui a déjà été signalé, c'est-à-dire s'il s'agit d'un rapport double. La déduplication est également particulièrement pertinente pour les projets dans lesquels les applications sont équipées de systèmes automatisés de signalement des plantages. Ces systèmes sont capables de collecter automatiquement les informations sur un platage et ils regroupent ces informations dans un document, appelé *rapport de plantage*, qui est soumis dans des les référentiels des plantages. Une partie importante des rapports soumis est en double et, par conséquent, leur détection est importante pour un processus de maintenance logicielle efficace. En raison du volume considérable de soumissions, en particulier dans les applications avec une large base d'utilisateurs, la déduplication manuelle des nouveaux rapports dans les BTS et dans les les référentiels de plantages peut être longue et laborieuse. Par conséquent, en pratique, une telle tâche nécessite le soutien de méthodes automatiques.

Dans cette thèse, nous avons étudié et proposé des nouvelles méthodes afin de traiter la déduplication des bogues dans les BTS et la déduplication des rapports de plantage dans les référentiels.

Notre première contribution porte sur les limites des méthodes d'apprentissage profond précédament porposées pour la déduplication des bogues. Ces méthodes sont basées sur des architectures avec une interaction limitée entre les données textuelles de deux rapports de bogues. Ainsi, nous proposons un nouveau modèle d'apprentissage profond basé sur l'alignement d'attention souple (soft alignment attention) qui peut extraire dynamiquement les caractéristiques d'un rapport qui est lié à l'autre rapport. Dans une évaluation empirique, nous démontrons que l'architecture proposée est plus efficace que les précédentes et notre méthode surpasse significativement les techniques bien connues et de pointe.

Dans la deuxième contribution de cette thèse, nous proposons une méthode de déduplication de rapport de plantage qui combine TF-IDF, l'alignement global optimal et l'apprentissage automatique d'une manière originale. Suivant la majorité de la littérature, la déduplication est effectuée en calculant la similarité entre des paires de trace d'appels. Contrairement aux techniques précédentes, notre méthode calcule le score d'alignement entre deux trace d'appels en se basant sur les positions et les fréquences globales de toutes les fonctions dans ces traces. Dans les expériences rapportées, nous comparons notre méthode avec des techniques bien connues et de pointe au moyen d'une nouvelle méthodologie d'évaluation. Notre méthode est la seule à obtenir de bonnes performances de manière constante dans tous les ensembles de données distincts. De plus, elle surpasse de manière significative les méthodes concurrentes dans la majorité des configurations expérimentales. Enfin, une étude approfondie de l'ablation est réalisée pour démontrer l'importance des éléments de la méthode.

Finalement, la dernière contribution de cette thèse aborde la question du calcul efficace des alignements de séquences optimaux. Particulièrement dans les applications populaires, l'efficacité joue un rôle crucial sur le déploiement des systèmes de déduplication en raison de la grande quantité de rapports de plantage soumis chaque jour. Puisque les performances de tels systèmes peuvent simplement être améliorée en accélérant la comparaison des traces d'appel, nous proposons une nouvelle méthode d'alignement de séquence pour la déduplication des rapports de platage qui, contrairement au temps de complexité quadratique des techniques de pointe, fonctionne en temps linéaire par rapport à la longueur des traces d'appel. Inspirée par des éléments particuliers de la déduplication des rapports de plantage, notre méthode mesure efficacement la similarité des traces d'appel en se basant sur : (i) l'alignement indépendant des même functions; et (ii) une heuristique qui fait correspondre les fonctions identiques en se basant uniquement sur leurs positions absolues. Malgré sa simplicité, nous montrons que la méthode proposée atteint des performances de pointe dans tous les scénarios d'évaluation et qu'elle est nettement plus efficace que les autres méthodes basées sur l'alignement optimal des séquences.

ABSTRACT

In software projects, a popular practice is to employ Bug Tracking Systems (BTSs) to manage and track records of bugs. A crucial task for BTSs consists in identifying whether a new report describes a bug that was previously reported or not, i.e., if it is a *duplicate* report. Deduplication is also particularly relevant for projects where applications are equipped with automated crash reporting systems. These systems are able to automatically collect information about a crash, then grouping it in a so-called crash report. Given the current industrial practice, repositories of crash reports contain a significant amount of duplicate crash reports and, thus, their detection is important for an effective software maintenance process. Due to the considerable submission volume, specially in applications with a wide user base, the manual deduplication of new reports in both BTSs and crash repositories can be time-consuming and laborious. Hence, in practice, such task requires the support of automatic methods.

In this thesis, we studied and proposed novel methods to address *bug deduplication* in BTSs and *crash report deduplication* in crash repositories.

Our first contribution addresses the limitation of previous deep learning methods for bug deduplication. Such methods are based on architectures with a limited feature interaction between textual data from two bug reports. Thus, we propose a novel deep learning model based on soft-attention alignment that can dynamically extract features from a report that is related to a specific part of the other report. Through a series of experiments, we demonstrate that the proposed architecture is more powerful than previous ones and our method significantly outperforms strong baselines and state-of-the-art (SOTA) techniques.

In the second contribution of this thesis, we propose a method for crash report deduplication which combines TF-IDF, optimum global alignment, and machine learning (ML) in a novel way. Following the majority of the literature, the deduplication is performed by computing the similarity between pairs of stack traces. In contrast to previous techniques, our method computes the alignment score between two stack traces based on the positions and global frequencies of all frames in the stack traces. We extensively compare our method with strong baselines and SOTA techniques by means of a new evaluation methodology for this task. Our method is the only one that consistently performs well in all distinct datasets. Additionally, it significantly surpasses competitive methods in the majority of experimental setups. Finally, an extensive ablation study is performed to demonstrate the importance of method elements.

Finally, the last contribution of this thesis addresses the issue of efficiently computing optimal sequence alignments for crash report deduplication. Due to the massive amount of crash reports submitted every day, deduplication systems must be efficient, thus requiring fast stack trace comparison. We propose a novel sequence alignment method for crash report deduplication that, in contrast to the quadratic complexity time of SOTA techniques, runs in linear time with respect to the length of the stack traces. Inspired by particular aspects found in crash report deduplication, our method efficiently measures the stack trace similarity based on: (i) the independent alignment of frames with the same subroutine; and (ii) an heuristic that matches identical frames based only on their absolute positions. Despite its simplicity, we show that the proposed method achieves SOTA performances in all evaluation scenarios and it is substantially more efficient than other methods based on optimal sequence alignment.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ACRONYMS

AUC	Area Under the ROC Curve
BERT	Bidirectional Encoder Representations from Transformers
Bi-LSTM	Bidirectional Long Short-Term Memory
BPTT	BackPropagation Through Time
BRNN	Bidirectional Recurrent Neural Networks
BTS	Bug Tracking System
CFC	Class-Feature-Centroid
CNN	Convolutional Neural Network
FC	Fully Connected
IDF	Inverse Document Frequency
IR	Information Retrieval
LCS	Longest Common Subsequence
LDA	Latent Dirichlet Allocation
LIFO	Last In, First Out
LSTM	Long Short-Term Memory
KDE	Kernel Density Estimation
MAP	Mean Average Precision
ML	Machine Learning
MLP	MultiLayer Perceptron
NLP	Natural Language Processing
NW	Needleman-Wunsch
QA	Quality Assurance
RNN	Recurrent Neural Networks
ROC	Receiving Operating Characteristic
RR@k	Recall Rate@k
Seq2seq	Sequence-to-sequence
SOTA	State-Of-The-Art
SVM	Support Vectors Machine
TF	Term Frequency
TF-IDF	Term Frequency-Inverse Document Frequency
TPE	Tree-structured Parzen Estimator
VSM	Vector Space Model

LIST OF APPENDICES

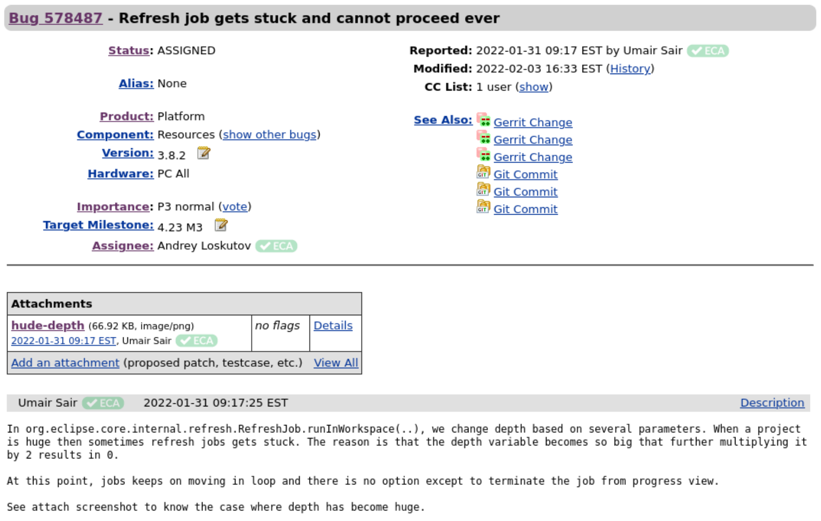
CHAPTER 1 INTRODUCTION

Despite developer efforts, bugs are likely to arise during a program lifetime, as a consequence, bug fixing is a prevalent activity in software development [1]. In order to support such activity, companies often employ *Bug Tracking Systems* (*BTSs*), to manage and track records of bugs. In an BTS, each bug is represented by a *bug report* – an example of such document is depicted in Figure 1.1. As shown in this example, a report might contain distinct categorical information related to a bug, e.g., the names of product and component affected by a error, severity, priority, and so on. Moreover, the person who submits a report must provide a *summary* and a *full description* of the bug. In our example, these two fields are represented by the bold line at the top of the figure and the text block at the bottom, respectively. Additionally, reporters can attach diverse types of files (e.g., screenshots, log files, and source codes) to a report.

A common strategy adopted by software projects, especially open-source ones, is to allow end-users, besides testers, to report new bugs in BTSs [2, 3, 4]. Such strategy improves software quality and provides system feedback to developers. However, it has a drawback: bug reports may be of poor quality or even describe non-existent bugs. Projects are aware of this disadvantage and, thus, implement a process for reviewing such reports, called *bug triage process* [5]. Overall, in the bug triage process, a person, called the *triager*, assesses the quality of a new bug report, verifies whether an equivalent error has not already been previously submitted, and certifies that the reported bug is reproducible. After that, the triager assigns a report to a developer or a team that will investigate the bug and fixed it. In Figure 1.1, the person in charge of resolving a bug is indicated in the field *assignee*. In cases where the bug cannot be reproduced, the triager can request further information for the reporter. This information exchange is recorded in the form of comments in the bug report.

Additionally, BTSs allow to control and visualize the current status of bugs in the maintenance process. The possible status values might vary among different BTSs, e.g., in Figure 1.1, status *ASSIGNED* means that an expert is fixing the bug. In the same BTS, for instance, *WORKSFORME* is attributed to non-reproducible bugs, whereas *RESOLVED* is associated with fixed bugs waiting for quality assurance. For further details regarding bug status and its cycle, we reference the reader to Koponen [6].

In additional to BTSs, many projects equip their products with automated crash reporting



Bug 578487 - Refresh job gets stuck and cannot proceed ever

Status:	ASSIGNED
Alias:	None
Product:	Platform
Component:	Resources (show other bugs)
Version:	3.8.2
Hardware:	PC All
Importance:	P3 normal (vote)
Target Milestone:	4.23 M3
Assignee:	Andrey Loskutov ✓ ECA
Reported:	2022-01-31 09:17 EST by Umair Sair ✓ ECA
Modified:	2022-02-03 16:33 EST (History)
CC List:	1 user (show)
See Also:	Gerrit Change Gerrit Change Gerrit Change Git Commit Git Commit Git Commit

Attachments		
hude-depth (66.92 KB, image/png) 2022-01-31 09:17 EST, Umair Sair ✓ ECA	*no flags*	Details
Add an attachment (proposed patch, testcase, etc.)		View All

Umair Sair ✓ ECA 2022-01-31 09:17:25 EST Description

```
In org.eclipse.core.internal.refresh.RefreshJob.runInWorkspace(..), we change depth based on several parameters. When a project
is huge then sometimes refresh jobs gets stuck. The reason is that the depth variable becomes so big that further multiplying it
by 2 results in 0.

At this point, jobs keeps on moving in loop and there is no option except to terminate the job from progress view.

See attach screenshot to know the case where depth has become huge.
```

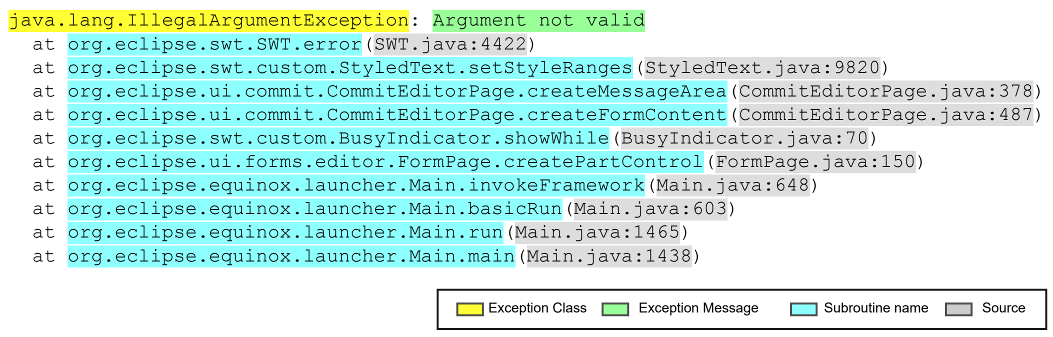
Figure 1.1 An example of a bug report.

systems (e.g., Apport[1] and Mozilla Socorro[2]). Such systems reduce the dependence of end-users that may not wish to notify failures or, due to the lack of technical knowledge, might not provide appropriate information for bug fixing. In a nutshell, these systems automatically detect software crashes, collect data related to user environment, system state and execution information, and group this data into a so-called *crash report* [7]. Each crash report is sent to a repository for reviewing and, after assessment, they can originate new bug reports within BTSs.

Besides the additional data related to the system and execution environment, crash reports are composed of stack traces which provide valuable information for bug investigation and fixing [8]. In Figure 1.2, we depict an example of stack trace in Java. The first line contains the exception type and the error message. The subsequent lines represents the state of an application call stack right before a failure. A call stack consists of special data structures, called *frames*, that are stored following the LIFO (last in, first out) principle. The first frame (topmost) represents the subroutine that was running at the moment of the crash. The remaining ones are associated to subroutines paused until the execution of the adjacent frames closer to the top. Specifically, the last frame is the application entry point, i.e., the bottom of the call stack. The displayed information in each frame depends on the environment and programming language. As shown in Figure 1.2, Java's stack traces contains the subroutine names and sources (filename and line number) of each frame.

Due to the asynchronous nature of bug submission in both crash report repositories and

[1]https://wiki.ubuntu.com/Apport
[2]https://crash-stats.mozilla.com/



```
java.lang.IllegalArgumentException: Argument not valid
  at org.eclipse.swt.SWT.error(SWT.java:4422)
  at org.eclipse.swt.custom.StyledText.setStyleRanges(StyledText.java:9820)
  at org.eclipse.ui.commit.CommitEditorPage.createMessageArea(CommitEditorPage.java:378)
  at org.eclipse.ui.commit.CommitEditorPage.createFormContent(CommitEditorPage.java:487)
  at org.eclipse.swt.custom.BusyIndicator.showWhile(BusyIndicator.java:70)
  at org.eclipse.ui.forms.editor.FormPage.createPartControl(FormPage.java:150)
  at org.eclipse.equinox.launcher.Main.invokeFramework(Main.java:648)
  at org.eclipse.equinox.launcher.Main.basicRun(Main.java:603)
  at org.eclipse.equinox.launcher.Main.run(Main.java:1465)
  at org.eclipse.equinox.launcher.Main.main(Main.java:1438)
```

Figure 1.2 Stack trace example in Java.

BTSs, a significant portion of the reports are duplicate, i.e., they are associated with the same error [9, 10]. Hence, in such environments, an important task is *report deduplication*, that consists in grouping duplicate reports into a single cluster, called *bucket*. This task avoids developers work on the same bug [9], provides complementary information about errors [11, 12], and helps to prioritize bug investigation [13].

Especially in systems with large user bases, it is impractical to manually deduplicate bug reports in BTSs and crash reports within repositories due to the high submission volume that is beyond the triage team capability [2, 10, 14, 15]. For instance, considering Mozilla's environment[3], over 350 new reports needed to be triaged every day in its BTS [15], while its crash repository could receive more than 300.000 crash reports in a day [10]. Therefore, automatic methods are needed to efficiently address such issue. In this work, we refer to the duplicate bug reports detection in BTSs as *bug report deduplication* or simply *bug deduplication*. Regarding crash reports, the analogous task is called *crash report deduplication*.

1.1 Research Objectives

Bug deduplication and crash report deduplication have been extensively studied in the last two decades. Nonetheless, such tasks are still considered practical open problems, i.e., method performances in term of effectiveness are far insufficient for deployment of automatic deduplication systems without human interference. Regarding efficiency, especially in crash report repositories, the deployment of state-of-the-art (SOTA) techniques in real environments is still a relevant research issue due to the high volume of report comparisons associated to the deduplication of a new report [10].

Furthermore, despite the vast literature, there is no consensus regarding the evaluation

[3]`https://mozilla.org/`

methodology and previous proposed methods were not extensively compared among themselves due to the lack of source code and data availability. Given the above, within the scope of this thesis, the objective is to investigate bug deduplication and crash report deduplication by: (i) proposing new methods; (ii) discussing evaluation methodologies; and (iii) extensively comparing and analysing our proposed methods with previous ones. More specifically, this is conceived around three main contributions:

1. **A novel deep learning for bug deduplication.** The majority of the studies have addressed bug deduplication by mainly comparing textual data between a pair of bug reports. Many of them proposed deep learning models to perform such comparison based on siamese neural networks [16], that is, architectures that separately generates fixed-length representations of textual data from each report. In siamese neural networks, text representations are invariable, i.e., the model cannot extract different features from textual data that are more relevant to specific comparison [17, 18]. In order to mitigate such shortcoming, an attentive model was proposed in [19] which generates fixed-length text representations by means of an *attention mechanism* [20]. In a nutshell, this mechanism compares word representations from a report with a unique set of textual features from the other report. Nonetheless, the textual comparison is still limited since the compression of textual data into fixed set of features can lose relevant information for a specific deduplication.

   Motivated by this limited information interaction of reports during the feature extraction, we propose a method that, better leveraging the attention mechanism, can dynamically focus on distinct segments of a report regarding the textual content of the other one. We demonstrated that the proposed method outperforms the competitive methods on all datasets and that the more dynamic feature interaction is crucial for bug deduplication. Moreover, as other contribution, we performed an extensive comparison of deep learning models and strong baselines employing a more realistic evaluation methodology.

2. **A method for crash report deduplication which combines TF-IDF, optimum global alignment, and machine learning.** Crash report deduplication is frequently addressed by mainly comparing the stack traces contained in crash reports. Since sequence order is relevant for the task, methods based on optimal global alignment [21, 22] have been proposed for measuring stack trace similarity. In a nutshell, optimal global alignment applied to crash report deduplication consists in finding the best end-to-end alignment between two stack traces. To compare distinct viable solutions, a scoring scheme is required to calculate the score of a sequence alignment. Brodie et al. [21] proposed a scheme in which an alignment of two identical subroutines (called *match*)

affects the final score based on two subroutine features: their absolute position in the stack traces and their global frequency within the repository. Such features are relevant for crash report deduplication because (i) subroutines near to the topmost positions are more likely to contain the bug that caused a crash [8], and (ii) frequent subroutines represent ordinary functionality that are usually not related to the crash. Exclusively employing positional information, Dang et al. [22] introduced parameters into the scoring scheme that control the impact of such information feature on the sequence alignment. Such parameters are learned by a machine learning (ML) algorithm and provide more flexibility to the method. However, both studies ignore the position and global frequency information regarding *unmatched* subroutines, the ones that are not shared between two stack traces. We argue that such features are also important in these cases since rare subroutines near to the top of the stack traces should have higher impact on the score than the frequent ones close to the bottom.

Inspired by the previous works, we propose *TraceSim*, a novel optimal global alignment method for crash report depuplication. TraceSim assigns a weight for each subroutine that depends on its rarity in a repository (similar to TF-IDF) and its position in a stack trace. Each weight captures the subroutine importance for the deduplication. Our method finds the optimal global alignment based on the weights of matched and unmatched subroutines in the sequence alignment. To provide a certain method flexibility on different datasets, ML algorithms are employed to learn parameters that control the impact of subroutine features on the final alignment. As an additional contribution, we propose an evaluation methodology that can measure the method capacity on: (i) separating duplicate and non-duplicate reports, (ii) correctly assigning a report to its correct bucket; and (iii) ranking similar stack traces. We extensively compare TraceSim to SOTAs techniques and strong baselines on five different datasets. Our experimental results demonstrated that TraceSim consistently achieve the best performance in the majority of the evaluation scenarios.

3. **A linear time stack trace alignment heuristic for crash report deduplication** Especially in industrial environments, deduplication systems usually receive a high volume of submissions, e.g., approximately two million reports are weekly submitted into Mozilla's repository [10]. Given such scenario, these systems must be carefully developed to achieve adequate throughput (processed reports per second). A possible approach to accelerate a deduplication system is to directly reduce the computational cost for calculating stack trace similarity. However, in the literature, the fastest methods that run in linear time are significantly less effective than SOTA techniques whose time

complexity is quadratic. Hence, we study how to efficiently compare stack traces based on sequence alignment without effectiveness degradation.

Since the primary goal of crash report deduplication is to capture the stack trace similarity and not find the best sequence alignment, we argue that some constraints of the optimal global alignment can be relaxed to improve efficiency without affecting the method effectiveness. Based on that, we propose an alignment heuristic that compares two stack traces in linear time. Such method, called *FaST*, groups subroutines based on the their names and, for each group, it iteratively matches subroutines by lining up the two available ones closer to the top. Due to the frequency differences, it may not be possible to match all subroutines within the stack traces. In such cases, we consider that subroutines are aligned to special structures, called *gaps*. The similarity score is then computed based on a similar scoring scheme proposed by TraceSim. In our experiments, we show that our alignment heuristic consistently achieves SOTA performances on all datasets and it is substantially faster than TraceSim and other methods based on optimal sequence alignment.

1.2 Thesis outline

The remaining of this document is organized as follows. Chapter 2 presents some background information and provides a critical literature review of studies in bug deduplication and crash report deduplication. Chapter 3 describes an overview of this thesis contributions. In the subsequent chapters, three research articles are presented. Chapter 4 proposes a soft alignment deep learning model for bug deduplication. Chapter 5 introduces TraceSim and a new methodology for crash report deduplication. Chapter 6 investigates the performance issue associated to sequence alignment methods in crash report deduplication and proposes *FaST*, a linear time method for stack trace comparison. A general discussion is presented in Chapter 7. Finally, Chapter 8 summarizes this thesis contributions and discusses limitations and promising research avenues.

1.3 Publications

The chapters 4, 5, and 6 of this thesis include, respectively, the following published articles:

1. Irving Muller Rodrigues, Daniel Aloise, Eraldo Rezende Fernandes, and Michel Dagenais, “A soft alignment model for bug deduplication,” in the 17th International Conference on Mining Software Repositories (MSR). New York, NY, USA: Association for Computing Machinery, 2020, p. 43–53.

2. Irving Muller Rodrigues, Aleksandr Khvorov, Daniel Aloise, Roman Vasiliev, Dmitrij Koznov, Eraldo Rezende Fernandes, George Chernishev, Dmitry Luciv, and Nikita Povarov, "Tracesim: An alignment method for computing stack trace similarity," Empirical Software Engineering, vol. 27, no. 2, p. 53, Mar 2022.

3. Irving Muller Rodrigues, Daniel Aloise, and Eraldo Rezende Fernandes, "FaST: A linear time stack trace alignment heuristic for crash report deduplication," in Mining Software Repositories (MSR2022), in press

Furthermore, during this Ph.D research, one of our collaborations also originated the following article:

- Aleksandr Khvorov,Roman Vasiliev, George Chernishev, Irving Muller Rodrigues, Dmitrij Koznov, and Nikita Povarov, "S3M: Siamese stack (trace) similarity measure," in 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), 2021, pp. 266–270.

CHAPTER 2 LITERATURE REVIEW

Sequential data comparison plays a crucial role in bug deduplication and crash report deduplication since texts and stack traces are one of the most valuable information sources for these two tasks, respectively. Hence, in this chapter, we present techniques for comparing similarity of two sequences. Most of the examples are related to natural language, but they can be generalized to any sequential data. Additionally, we describe the methodology evaluations and methods proposed for such tasks. Even though a comprehensive literature review is provided here, it is worthy to mention that this manuscript is composed of self-contained chapters which include their particular and independent literature reviews.

2.1 Information retrieval techniques

A classic problem in Information Retrieval (IR) consisting in retrieving documents that are relevant for a given query [23]. To compare the textual data between each document and a query, two classic techniques can be employed: *Term Frequency – Inverse Document Frequency* (TF-IDF) and *BM25*.

2.1.1 TF-IDF

A common approach to compare textual data is to represent texts as fixed-length vectors whose each dimensions is associated with a specific term. Such vector representation is called *Vector Space Model* (VSM) [23, 24] and, for simplicity, we consider that each term is a word in a pre-defined set V, called *vocabulary*. In Table 2.1, we depict an example of VSM representations of two sentences: Brass is a heavy game while Wingspan is a light one and Brass is not that heavy game. These sentences are respectively named as Sentence 1 and 2. Each column of Table 2.1 represents one of 13 vocabulary words, whereas the rows contains the dimension weights in each sentence. In our example, the weights are simply the word frequencies. For instance, the cell in row 1 and column 2 has the value 2 because the word is

	Brass	is	not	that	heavy	game	a	while	Wingspan	light	one
Sentence 1	1	2	0	0	1	1	2	1	1	1	1
Sentence 2	1	1	1	1	1	1	0	0	0	0	0

Table 2.1 Example of vector representation of two sentences: Brass is a heavy game while Wingspan is a light one and Brass is not that heavy game.

occurs twice in Sentence 1. As one can notice, vector each sentence is represented by vector with length equal to the vocabulary size.

In exampled presented in Table 2.1, vector weights are the raw frequency of words. However, such approach can be problematic since word frequency distribution is usually skewed – only a few words have high frequencies in a document [24]. Consequently, the weight of some words will be much higher than the remaining ones, i.e., similarity comparison will be highly affected by a small subset of terms. Moreover, a set of words (including those with high frequency) appears in most of the documents and, therefore, they do not help to discriminate documents [23]. For instance, considering a board game web forum, the word game is expected to appear in a significant part of the posts. Thus, having the word game does not make documents more or less similar. However, the word storytelling is rarer and documents that share such word are more likely to be more similar among them. A technique to overcome such shortcomings is *TF-IDF*.

TF-IDF weights vectors by means of *inverse document frequency* (IDF) and word frequency, called *term frequency* (TF). TF is the local information about how important is a word to a specific document [23]. IDF measures the discrimination level of the words in a dataset [24]. Common words in a domain are not helpful to discriminate documents since most of them contain these terms. So, to reduce their relevance in the documents, these words should have low weights in the vectors. The following equation calculates IDF:

$$\mathrm{IDF}(t) = \log\left(\frac{|S|}{\mathrm{df}(t)}\right), \tag{2.1}$$

where $|S|$ is the number of documents in the dataset S, $\mathrm{df}(\cdot)$ is the number of documents that contain a word t. The new weight w_{td} of a word t in a document d using TF-IDF is computed as:

$$w_{td} = \mathrm{TF}(t,d) \times \mathrm{IDF}(t), \tag{2.2}$$

where $\mathrm{TF}(t,d)$ returns the term frequency of word t in the document d.

2.1.2 BM25

Despite its simplicity, TF-IDF ignores an import factor that affects term importance: document length. Long documents tend to repeat the same terms through their content which can increase their impact on the similarity measurement [25]. Moreover, as long documents have a more extensive variety of words, they have a higher probability of being more similar to queries than the shorter ones.

Okapi weighting or BM25 weighting scheme is a well-known technique in information retrieval that normalizes the term impact by the document length. BM25 is based on probability theory and measures the document relevance to a query [23]. The following equation calculates the BM25 score of a document d given the query q:

$$BM25(d,q) = \sum_{t \in d \cap q} \mathrm{IDF}(t) \times \frac{(k_1+1)\,\mathrm{TF}(t,d)}{k_1((1-b)+b \times (\frac{L_d}{Lv})) + \mathrm{TF}(t,d)}, \tag{2.3}$$

where $d \cap q$ is the disjunction of the terms in the document d and query q, k_1 is a positive parameter that regulates the influence of the term frequency, L_d is the document length, Lv is the average length of the documents in the dataset, and b is a parameter with values between 0 and 1 which controls the effect of the document length.

BM25 was designed to score documents with a single content. However, in many domains, documents can have multiple fields. For example, a website about video games have reviews that are structured into title, content, and summary - which is a compilation of the reviewer's opinion. Merging these fields into a single text is a possible option to compare the reviews using Okapi weighting. However, this strategy gives the same weight to the three fields. The title and summary might be more relevant to the document comparison and, therefore, equally weighting all fields in the final score can be misleading. A more flexible option is to use the BM25F [26, 27] which is an extension of BM25 and applies different weights to each field. Before computing the final score, the local relevance of a term t in document d is calculated using $\mathrm{TF}_D(d,t)$:

$$\mathrm{TF}_D(d,t) = \sum_{f \in d} w_f \frac{\mathrm{TF}(t,f)}{1 - b_f + \frac{b_f * L_f}{Lv_f}}, \tag{2.4}$$

where f is a field in the document d, $\mathrm{TF}(t,f)$ is the frequency of term t in a field f, w_f and b_f are scalar parameters related to f, L_f is the length of a field f, and Lv_f is the average length of a field f. Given a positive variable k_1, the following equation is employed to compute BM25F score:

$$BM25F(d,q) = \sum_{t \in d \cap q} \mathrm{IDF}(t) \times \frac{\mathrm{TF}_D(d,t)}{k_1 + \mathrm{TF}_D(d,t)}. \tag{2.5}$$

In BM25F, the parameters w_f and b_f of each field and k_1 have to be tuned. Robertson and Zaragoza [28] describes many strategies to find the optimum values for those variables.

2.2 Optimal Global Alignment

A shortcoming of the previous IR techniques is that they disregard sequence order, which might be a crucial information for some problems, e.g., the order of letters in words are crucial for finding misspells in a text. In Figure 2.1, we depict an example of how to compare a misspelled word acknoledgemint with its correct form acknowledgment. Such example illustrates a *global alignment* between these words that consists in lining up each sequence element to either: an element from the other sequence or a *gap*, a representation of an insertion/deletion operation depicted by a hyphen symbol in Figure 2.1. An alignment between two identical elements (e.g., the two letters a) is called a *match*, whereas the opposite scenario (e.g., letters i and e) represents a *mismatch*. A constraint in global alignment is that the initial sequences can be restored by removing gaps from the found alignment, i.e., element order must be preserved.

```
a c k n o - l e d g e m i n t
| | | | | | | | | | | | | | |
a c k n o w l e d g - m e n t
```

Figure 2.1 Global alignment between acknoledgemint and acknowledgment.

Since there is usually a large number of valid global alignments between two sequences, an important problem consists in finding the best one which is referred to as *optimal global alignment problem*. In order to compare different solutions, a scoring scheme is defined to compute the values of each match, mismatch, and gap alignment. In this manuscript, such values are computed by means of the functions match(·), mismatch(·), and gap(·), respectively. Thus, the score of a global alignment is the sum of match values penalized by the sum of mismatch and gap values. Given two sequences s and s' with lengths n and m, the highest alignment score between such sequences is found in $O(nm)$ time complexity by means of the Needleman-Wunsch (NW) algorithm.

NW is a dynamic programming algorithm that fills a Matrix $M \in \mathbb{R}^{n \times m}$ from the smallest row and column indexes until the highest ones. Considering that s_i and s'_j represents the i-th and j-th elements of the sequences s and s', respectively, $M_{i,j}$ is computed as follows:

$$M_{i,j} = \max \begin{cases} M_{i-1,j} + \text{gap}(s_i) \\ M_{i,j-1} + \text{gap}(s'_j) \\ M_{i-1,j-1} + F(s_i, s'_j) \end{cases}, \tag{2.6}$$

where $F(s_i, s'_j)$ is defined as follows:

$$F(s_i, s'_j) = \begin{cases} \text{mismatch}(s_i, s'_j), & \text{if } s_i \neq s'_j \\ \text{match}(s_i, s'_j), & \text{otherwise} \end{cases}. \tag{2.7}$$

After running the algorithm, the alignment score is stored in $M_{n,m}$.

2.3 Deep learning

In the recent past, deep learning has achieved the state-of-art in many different fields such as NLP, vision and speech recognition [29]. Neural networks, that lost their popularity after the '80s, have emerged as one of the most important techniques in machine learning due to many factors: data availability, the increase of computational power, regularization techniques, understanding of how to make training more stable and so on. Deep learning techniques are based on a hierarchical structure of representations in which the highest-levels depends on the lowest ones [30]. Each representation level abstracts information from the previous one which can generate representations that are more invariants and focus on the most relevant information of a problem [31]. This section describes the neural networks - how they work and methods to train them - and also provides high-level explanations of recurrent neural networks (*RNN*) and the attention mechanism.

2.3.1 Neural networks

Inspired by our brain function, researchers developed a mathematical model of the neuron, depicted in Figure 2.2, with three main components: input links, linear function and activation function. A neuron receives inputs through the input links. Each link has a weight associated with it. After that, the linear function sums the multiplication product between the weight and input values for every input link. The activation function is a non-linear function that receives such weighted sum as input and applies an often non-linear transformation on it.

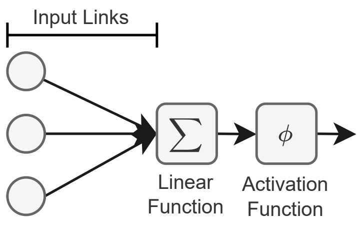


Figure 2.2 Mathematical neuron model.

A set of neurons can connect each other creating a structure called *neural network*. This section is focused on *feed-forward networks*, a type of neural network that groups neurons in a stack of layers [32]. In this manuscript, following Goodfellow et al. [33], we also referred a feed-forward network as multilayer perceptron (MLP). In such network, each layer only receives the outputs of the previous adjacent layer in the stack. This arrangement creates a directed graph in which information run through a point to another one without any cycle. Consequently, the output of the feed-forward neural network depends only on its link weights and initial input.

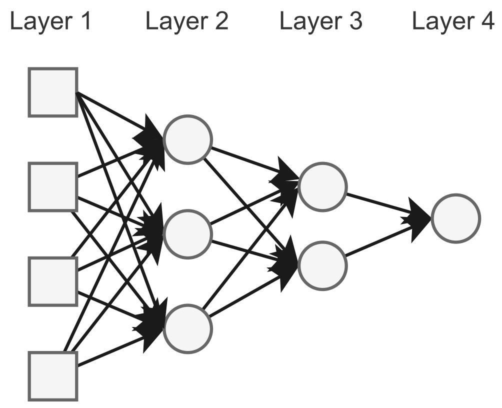


Figure 2.3 Example of feed-forward neural network.

Figure 2.3 depicts an example of a feed-forward neural network with four layers where the circles represent the neurons, the directed lines symbolize the input links (weights), and the squares depict the inputs. The first layer, named input layer, receives the input and does not have any weight. The last layer, called output layer, generates the final output of the neural network. The hidden layers are all the other ones that are between the input and output layers. Neural networks have a *depth* architecture when it has more than one hidden layer, otherwise its architecture is named as *shallow* [34].

The L layers in a feed-forward neural networks are indexed by a unique positive number. The matrix $W^l \in \mathbb{R}^{|l| \times |l-1|}$ denotes all link weights between layer l and $l-1$, where $|l|$ and $|l-1|$ are the number of neurons in $|l|$ and $|l-1|$,respectively. The linear function of a layer l is represented by:

$$z^l = W^l h^{l-1} + b^l, \tag{2.8}$$

where $h^{l-1} \in \mathbb{R}^{|l-1|}$ is the output of the neurons in the previous layer $l-1$, $z^l \in \mathbb{R}^{|l|}$, and $b \in \mathbb{R}^{|l|}$ is called *bias* and translates the function from the origin. Except in the input layer, the neuron outputs are usually produced by the activation function - often a non-linear function represented by ϕ. Table 2.2 shows some activation functions used in literature. Given an activation function ϕ, the neuron outputs in a layer l is denoted by:

$$h^l = \phi(z^l), \tag{2.9}$$

Function Name	**Equation**
Sigmoid	$\frac{1}{1+e^{-x}}$
Tanh	$\frac{2}{1+e^{-2x}} - 1$
Relu	$\max(0, x)$

Table 2.2 Activation functions ϕ.

The vector $h^l \in \mathbb{R}^{|l|}$ represents the output of all neurons in the layer l and θ symbolizes all parameters - weights and bias - of the neural network.

Many machine learning problems require that the neural network outputs a probability distribution [35]. In binary classification problems, a viable option is to use a neural network with a single neuron in the last layer that has a sigmoid function as the activation function. Since this function returns values between 0 and 1, the output of this neuron can be interpreted as $P(y^+|x;\theta)$ - the probability of the positive class y^+ given the input x and the parameters θ.

In multiclass classification, neural network has an architecture in which each neuron of the last layer is linked to a specific class. For instance, in a neural network with three neurons in the last layer, the first, second and third neurons can represent the score of an image being, respectively, a cat, dog or person. The neuron outputs usually does not have the properties required to be a probability distribution since their outputs range in an interval $(-\infty, +\infty)$ and their output sum is not guaranteed to be 1. An alternative to assure such properties is to normalize the outputs using a function, called *softmax* function. After applying the softmax function, a neuron output that represents the label y can be interpreted as the conditional probability of y given an input x and the parameters θ of the neural network. The following equation calculates the conditional probability $P(y|x;\theta)$:

$$P(y|x;\theta) = h_y^L = \frac{e^{z_y^L}}{\sum_{y'}^{\mathcal{Y}} e^{z_{y'}^L}}, \tag{2.10}$$

where $\mathcal{Y}$ is the set of all labels and z_y^L is the linear output of the neuron that represents y. The layers with the softmax function as their activation functions are called *softmax layers* in this manuscript. Two techniques, called *backpropagation algorithm* and *gradient descent algorithm*, are used to find θ values that best estimate $P(y|x;\theta)$.

Generally, in the machine learning problems, there is an objective function and set of a training examples. In this project, despite the small differences [33], the objective function is also referenced as *cost function* or *loss function*. The goal is to find the model parameters that minimize or maximize the loss function in the training examples. Given a fixed set of training instances, the derivative of a parameter estimates how much small changes in a parameter value increase or decrease the loss function. Thus, based on the derivative values, small parameter value updates can be made to the direction that minimizes or maximizes the loss function. This process is called descend gradient algorithm. In the neural network, the backpropagation algorithm is the technique used to efficiently calculates the parameter derivatives. This algorithm is based on calculus chain rules and propagates the error back from the output layer until the input layer [33].

2.3.2 Recurrent neural network

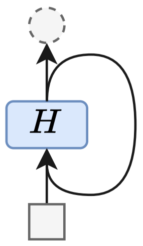


Figure 2.4 RNN with one self-connected hidden layer H.

Different from feed-forward neural networks, *Recurrent Neural Networks* (RNNs) is a type of network whose neurons have connections to themselves [32]. Such connections generate cycles which allows RNNs to have a state and, consequently, a memory capability. Figure 2.4 depicts a simple RNN with one self-connected hidden layer, represented by H. The input and output of H are illustrated by the square and dashed circle, respectively. This section focus on RNN due to its simplicity.

Figure 2.5 illustrates an example which the RNN predicts an output for each word in the sentence `this store is the worst`. In RNNs, a *step* or *timestep* denotes each time that the network receives an input. Steps are indexed by positive numbers. As shown in Figure 2.5, for each step, the same hidden layer calculates a new output, also called *state*, using the hidden output

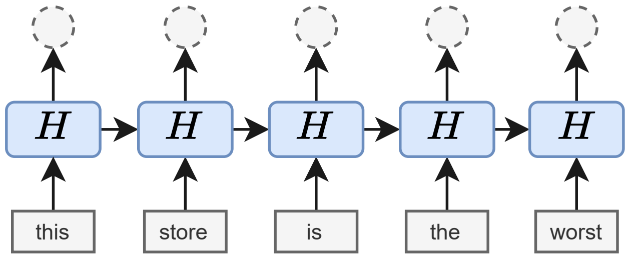


Figure 2.5 Unfolded RNN.

of the previous step and a input. Thus, in a step t, the linear output z_t of the hidden layer H with $|H|$ neurons is:

$$z_t = b^H + Ux_t + Wh_{t-1}, \tag{2.11}$$

where $b^H \in \mathbb{R}^{|H|}$ is the bias, $x_t \in \mathbb{R}^I$ is the input, $h_{t-1} \in \mathbb{R}^{|H|}$ is the hidden output in a step $t-1$, $U \in \mathbb{R}^{|H|\times I}$ are the weights applied in the input, and $W \in \mathbb{R}^{|H|\times|H|}$ are the weights of the previous hidden state. The final hidden layer output in t is:

$$h_t = \phi(z_t). \tag{2.12}$$

After passing all sequence elements to the RNN, this generates a directed acyclic graph, called *unrolled neural network*, that represents the same RNN for a specific input [33]. This process of removing the RNN cycle is known as *unrolling* or *unfolding*. Figure 2.5 depicts the unfolded RNN after passed it through the sentence this store is the worst.

In RNNs, an output of hidden layer in a step t can impact on all the other outputs after t. Thus, calculating the weight derivative requires measuring the output influence in the loss function values of the subsequent outputs [36]. Hence, the RNN are first unrolled and, then, the algorithm, called *backpropagation through time* (BPTT), is applied to backpropagate the error and compute the weight derivatives [36]. BPTT is very similar to the backpropagation and sequentially flows the error back from the last step T down to the first one.

In the traditional RNNs, information always flows from the past to the present. However, in many problems, also receiving information from the future can be determinant to label a sequence correctly in a step t [36]. For instance, the previous and next characters can help to disambiguate the correct letter in a specific part of a handwriting word. For those problems, it is more suitable to use a *bidirectional recurrent neural networks* (BRNN) [37]. BRNNs have two hidden layers. The first layer is unrolled from step 1 until the last step T while the latter is reversely unfolded from step T until the first step [36]. Figure 2.6 depicts the moment that a BRNN generates an output in a step t. As illustrated in Figure 2.6, BRNN has two hidden layers H and H' that receive their generated outputs in the steps $t-1$ and $t+1$, respectively. The final output of BRNN in t is the concatenation of the two hidden layer outputs. For training, BPTT is also used to calculate the BRNN weights derivatives.

The learning process of RNN is known to be unstable. During training, RNN can suffer from exploding and/or vanishing gradient problems [36]. The first problem occurs when the gradient blows up to higher values when the error is backpropagated, while the second one arises in the opposite scenario: gradients decay to insignificant values. Exploding gradient problem can be easily managed by clipping gradients [38]. However, the vanishing problem

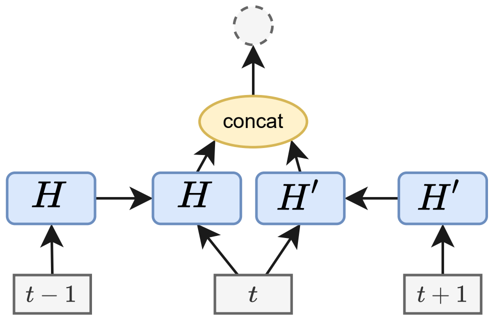


Figure 2.6 Bidirectional RNN.

can be more challenging to deal with.

Although RNNs can theoretically learn long term dependencies, in practice, due to the vanishing problem, RNNs are not able to handle these dependencies [39]. Aiming to overcome this problem and, therefore, making the RNN training more stable, Hochreiter and Schmidhuber [40] have proposed a new internal architecture to RNN, called *Long Short-Term Memory* (LSTM). In this section, we focus on the LSTM architecture introduced by Gers et al. [41] because it is the default LSTM implementation in two popular machine learning libraries: Tensorflow [42] and Pytorch [43].

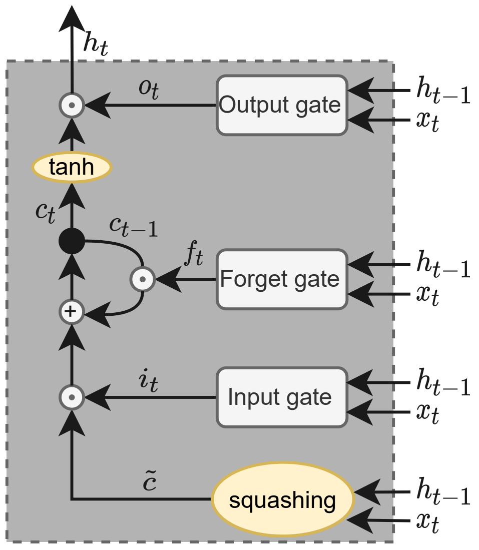


Figure 2.7 LSTM memory block.

The fundamental element of the LSTM is the *memory block* [36]. The memory block is composed of self-connected memory cells and three multiplicative gates (*input gate*, *forget gate* and *output gate*). Figure 2.7 depicts a memory block with one memory cell. The content of the memory cell is controlled by the gates that select which portion of the memory is updated and released as an output of the block. This gating mechanism can create paths in which the gradient of long-dependencies can flow without vanishing [33].

As shown in Figure 2.7, a squashing function combines information from the input x_t at the current step t and the memory block output h_{t-1} of the last step $t-1$ into a single vector $\tilde{c}$. In the LSTM, $\tilde{c}$ is called *squashed input* and is computed as follows:

$$\tilde{c} = \tanh(U^{\tilde{c}}x_t + W^{\tilde{c}}h_{t-1} + b^{\tilde{c}}), \tag{2.13}$$

where $U^{\tilde{c}}$ and $W^{\tilde{c}}$ are weight matrices and $b^{\tilde{c}}$ is the vector of bias weights.

After generating the squashed input, the input gate is responsible for selecting which part of this input will be written in the memory cell. For that, the memory block performs the element-wise multiplication (represented as $\odot$) of the squashed input $\tilde{c}$ and the vector i_t produced by the input gate. The input gate can fully or partially block the input by selecting values between 0 and 1 in i_t. For instance, the input gate can block all the information from the input by setting all the values of i_t to zero or it can move forward the full information of a specific dimension of $\tilde{c}$ by inserting the value 1 in the same dimension of i_t. The input gate generates the vector i^t as follows:

$$i_t = \text{sigmoid}(U^i x_t + W^i h_{t-1} + b^i), \tag{2.14}$$

where U^i and W^i are weight matrices and b^i is a vector of bias weights.

The forget gate chooses the part of the memory cell that will be erased using a similar mechanism employed in the input gate. The forget gate generates a vector f^t:

$$f_t = \text{sigmoid}(U^f x_t + W^f h_{t-1} + b^f), \tag{2.15}$$

where U^f and W^f are weight matrices and b^f is the vector of bias weights. Using the element-wise operation, the forget gate can keep the partial or complete information of the previous state of the memory cell by selecting values between 0 and 1 in f_t. The new memory cell state c_t in the step t is the combination of the piece of information that the input gate chooses to block in the squashed input and that the forget gate preserves in the previous memory cell:

$$c_t = i_t \odot \tilde{c} + f_t \odot c_{t-1}. \tag{2.16}$$

Following the same mechanism of the two gates, the output gate generates a vector, denoted

o_t, that selects a portion of the current memory cell state c_t to be the memory block output h_t at the step t. The following equations computes o_t and h_t:

$$o_t = \text{sigmoid}(U^o x_t + W^o h_{t-1} + b^o), \tag{2.17}$$

$$h_t = \tanh(c_t) \odot o_t, \tag{2.18}$$

where U^o and W^o are weight matrices and b^o is the vector of bias weights. All gates employ the sigmoid function to make all values to be between 0 and 1, and their decisions are only based on the input of the step t and the last output of the memory block.

LSTMs can represent texts as fixed-length representations using three approaches. The first one uses the RNN output from the last step to represent a text [18]. The second generates the fixed-length vector by applying a *max pooling* or *mean pooling* to all the RNN outputs [44]. The last one, called *intra-attention*, uses the *attention mechanism* to learn how to represent texts as vectors automatically [44].

2.3.3 Attention

The sequence-to-sequence network (*seq2seq network*) is a neural network that yields a sequence, named *target sequence*, by receiving another sequence, named *source sequence*, as input. Sutskever et al. [45] have originally proposed this network to the machine translation field, although other studies (e.g., [46], [47], [48]) have applied the sequence-to-sequence in many different problems.

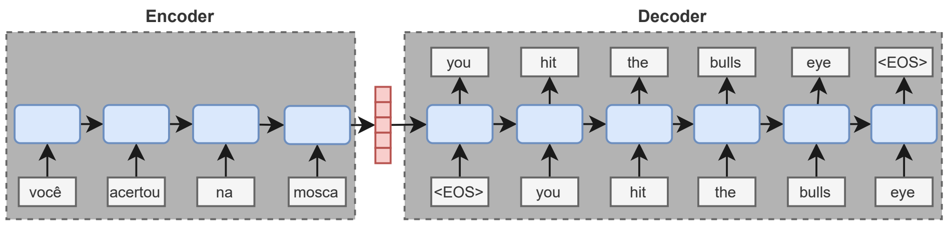


Figure 2.8 Seq2seq network.

Figure 2.8 depicts an example of seq2seq network that translates the sentence você acertou na mosca in Portuguese to English. As shown in this figure, the seq2seq network is composed of two distinct RNNs: the *encoder* and the *decoder*. The encoder receives the source sequence inputs and generates a fixed-length vector in the end. Using this vector, the decoder outputs the target sequence which, in the example of Figure 2.8, is the English translation you hit the bull's eye of the sentence você acertou na mosca. In the first step, the decoder always receives the symbol <EOS> to inform it that a new sequence have to be produced. In the next steps, the decoder inputs are the previous elements of the target sequence.

In the seq2seq network, encoders have to extract information from the source sequence that is helpful to the decoder and compress this information into a fixed-length vector. This compression can be the network bottleneck, especially for long sequences. Bahdanau et al. [20] have proposed a new method, called *attention mechanism*, to help seq2seq networks to tackle long sequences. In the literature, the attention models can be categorized in soft attention and hard attention [49]. This subsection focus on the former one.

The attention mechanism allows a neural network to focus on specific parts of the input in a particular moment. For instance, Figure 2.9 shows the moment when a decoder predicts the word hit after predicting the word you. Besides the word representation of you and the last decoder hidden output h_1^d, the decoder selects the word hit using a vector $\bar{c}_2$, called *context vector*, that is yielded from the attention mechanism based on encoder hidden outputs. Basically, such vector contains the most relevant information from the encoder to the current decoding step.

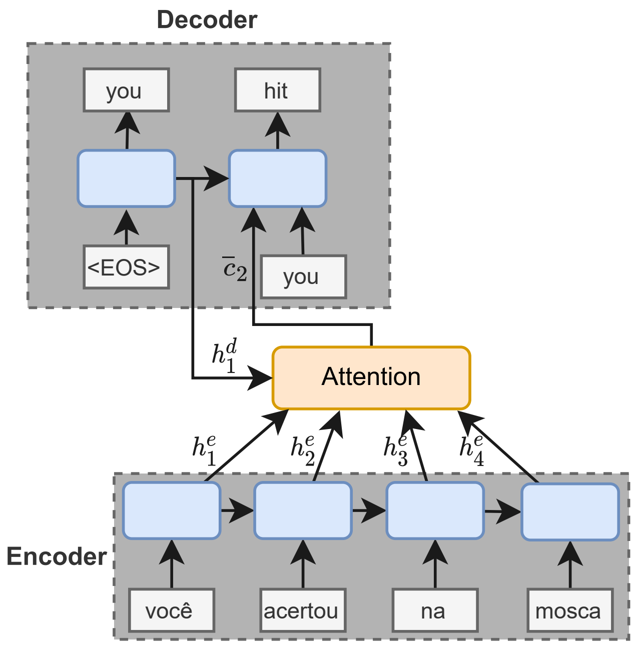


Figure 2.9 Seq2seq with attention.

In order to generate a context vector $\bar{c}_t$ within a step t, first, the attention mechanism computes a score between the last decoder hidden output h_d^{t-1} and each encoder hidden output by means of a function $F(\cdot)$. Being W_α a matrix of parameters, u a vector of parameters and $[;]$ a concatenation operation, Table 2.3 shows possible options of $F(\cdot)$ [49]. Then, such scores

Function Name	**Equation**
dot	$(h_d^{t-1})^\top h_e^i$
general	$(h_d^{t-1})^\top W_\alpha h_e^i$
concat	$u^\top tanh(W_\alpha[h_d^{t-1}; h_e^i])$

Table 2.3 Attention function scores.

are normalized using a softmax function. Thus, a normalized score α_i, called *attention score*, of the i-th encoder hidden output is:

$$\alpha_i = \frac{F(h_i^e, h_{t-1}^d)}{\sum_{j=1}^{T} F(h_j^e, h_{t-1}^d)}, \tag{2.19}$$

where T is source sequence length and h_e^i is the encoder hidden ouput in a step i. Finally, a context vector $\overline{c}_t$ is computed as the weighted average of the encoder hidden outputs:

$$\overline{c}_t = \sum_{j=1}^{T} \alpha_j h_e^j. \tag{2.20}$$

In summary, the higher is the attention score of a encoder hidden output, the largest is its impact on the context vector $\overline{c}_t$.

2.4 Representation learning

Traditionally, feature engineering has been a crucial task in machine learning. This task investigates how to create a data representation that positively impacts the performance of machine learning methods [31]. In feature engineering, humans use their knowledge about a domain to select manually discriminative and relevant features from the data. However, the use of human capabilities has a price, feature engineering is known to be a time-consuming and a labor costly task [31]. Due to these high costs, a new field, called representation learning or feature learning, has focused on proposing methods that can automatically learn features without human intervention.

Recently, deep learning techniques have emerged as featuring learning technique. Neural networks can learn automatically data representations that make them achieve state-of-art performance without the use of hand-craft features. In NLP, a significant breakthrough was the use of a neural network to learn efficiently word representations, called *word embeddings*.

2.4.1 Word embedding

Articles, books, tweets, texts are ways to communicate among ourselves. These simple collections of words are powerful enough to express our feelings or explain complexity theories. When humans read a word, they can understand its meaning and what it represents. Unfortunately, standard machines are not able to understand or extract information from these words. For them, words are just arbitrary arrays of bytes. Modeling words in meaningful representations is essential to create machines that can understand texts.

In the past, NLP systems represented words as sparse binary vectors using the one-hot representation [50]. In this representation, each vector dimension is linked to a word in the vocabulary (the vector dimensionality depends on the vocabulary size). If the i-th dimension is related to a word w, so the vector of w have only the i-th dimension with value 1 and the other ones with 0. Given a vocabulary with five words (home, canada, brazil, computer, machine), Table 2.4 shows an example of how to represent those words with a one-hot representation.

Word	**Vector**
home	[1,0,0,0,0]
canada	[0,1,0,0,0]
brazil	[0,0,1,0,0]
computer	[0,0,0,1,0]
machine	[0,0,0,0,1]

Table 2.4 Example of one-hot representation.

In real scenarios, vocabulary can easily have more than 30,000 words yielding the vectors with thousands of dimensions. This high dimensional representation can be affected by the curse of dimensionality. In this case, the curse of dimensionality is related to the large number of examples required to train a model [51]. A good discriminator needs examples that cover a relevant part of the feature space to be able to well discriminate the problem. The increase of the input size (feature space) grows exponentially the number of relevant areas to be covered and, consequently, more examples are required to train a good discriminator.

Besides the curse of dimensionality, in the one-hot representation, the weights linked to a dimension are only trained when a specific word appears in the training dataset. Therefore, rare words have their parameters poorly tuned [52]. Moreover, the parameters of words that do not exist in the training dataset are never estimated which can degrade the model performance when those words appear in one test example. Finally, it is not possible to compare the similarity between words using the one-hot representation because the vector distances are always the same [50].

Recently, a new type of representation, called *word embeddings* or *distributed word representation*, has been shown very useful for NLP [53]. In word embeddings, words are represented as real, low dimensional and dense vectors. Those vectors describe word positions in a new feature space that retain syntactic and semantic information [53, 54]. In contrast to one-hot representation, word embeddings suffer less with the curse of dimensionality and improve the model capability to handle unknown and rare words in the training [55]. Furthermore, using distributed word representations, it is possible to perform arithmetical operations and calculate the distance between words. Mikolov et al. [56] have found compelling results when linear transformations are applied to word embeddings, for instance, the vector of Berlin is close to the sum of vectors of Germany and capital.

Besides words, dense and real vectors can be used to represent characters, sentences, documents, images or any object [57]. The *distributed representations* (also called *embeddings*) of those objects capture relevant features that characterize them [55]. In this manuscript, encoders are defined as any model that generates a distributed representation of objects. In the literature, a specific neural network, called *siamese neural network*, uses a shared encoder to compare similarity between objects.

2.4.2 Siamese neural networks

Siamese neural networks were proposed by Bromley et al. [58] to address signature verification. Due to its simplicity, many other studies have applied them to a great variety of tasks: person re-identification [59], bug deduplication, sentence similarity, question answering and so on. Figure 2.10 depicts an example of a siamese neural network applied to the question answering problem, a task that consists of finding the answer to a question. In this figure, there is one question what is a country?, two possible candidate answers and an encoder which can be any neural network. The encoder generates the distributed representation of $Q1$, $A1$, and $A2$ (depicted by red) for, respectively, the Question1, Answer1 and Answer2.

Siamese neural networks have a shared component that encodes objects to vectors. It is expected that such shared component learns discriminative features since its output is used by the neural network to compare both objects. During feature extraction, a siamese network has only access to the information of a specific object, i.e., the encoding of an object is independent of the other objects [17]. Consequently, siamese neural networks always return the same distributed representation for equal inputs.

In Figure 2.10, the siamese neural network calculates the similarity between the distributed representations of a question and an answer using a cosine similarity function. This function is traditionally used to compute the similarity between vectors in NLP, although, other

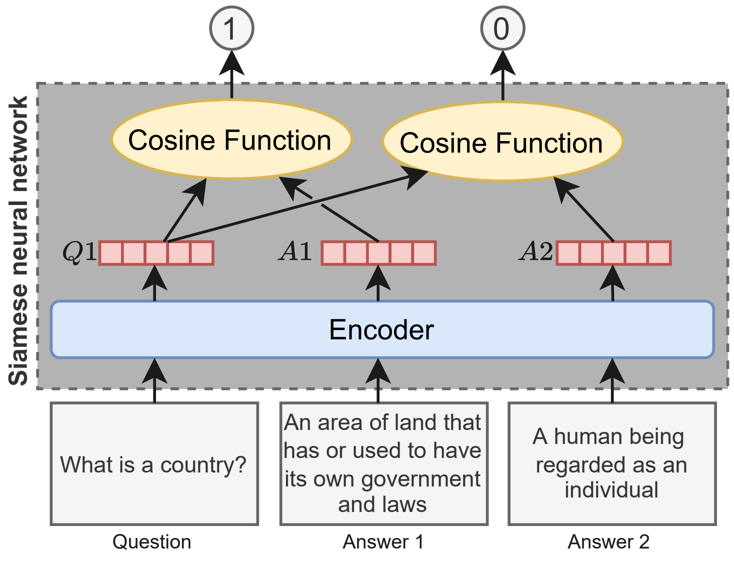


Figure 2.10 Siamese neural network for question answering.

dissimilarity metrics, similarity functions or even neural networks (a complex function) can be employed to compare the distributed representations.

Figure 2.10 displays a siamese network that yields the highest similarity to the question and its correct answer (Answer1) and much smaller score to the pair formed by the wrong answer (Answer2) and the question. This figure illustrates a desirable scenario where siamese network is able to draw the question and its correct answer into a feature space where they have almost the same direction. In general, we want to train siamese networks to produce higher scores to similar objects than to the different ones. One option to achieve this is to use labeled pairs of objects and the *contrastive loss* to train the model [60]. The pairs labeled with 1, called *positive pairs*, are composed of two similar objects, while *negative pairs* - identified with 0 - are formed with different objects. The siamese neural network can be trained to minimize the contrastive loss:

$$\mathcal{L}_c = y(1 - S(q,o)) + (1-y)(S(q,o) - M), \tag{2.21}$$

where q and o are the query and object of a pair, y is the pair label, S is the output of siamese neural network and M is the margin.

Instead of pairs, the siamese network can be trained using triplets [18]. Each triplet is a tuple composed of the following sequence of objects: a query (*anchor*), a similar instance to the query (*positive*) and a different object to the anchor (*negative*). For instance, the example of Figure 2.10, the tuple (Question1, Answer1, Answer2) is a valid triplet while (Question1, Answer2, Answer1) is an invalid because the positions of the positive and negative are inverted.

When the triplets are employed, the training objective can be defined as a *hinge loss* (called *triplet loss* or *max margin loss*):

$$\mathcal{L}_t = \max\{0, M - S(q, o^+) + S(q, o^-)\}, \tag{2.22}$$

where q, o^+ and o^- are the query, positive object and negative object of a triplet.

Another approach for training a siamese neural network is to consider the problem as a binary classification [61]. For instance, in the question answering problem, the siamese network output can be interpreted as the probability of an answer being correct given a question. For this end, it is necessary to change a little bit the siamese network architecture. A classifier (e.g., a neural network) is added after the shared encoder and it returns the probability of an example being positive. Given the positive and negative pairs in the training, the siamese network is trained to minimize the *binary cross-entropy loss* which is described in the following expression:

$$\mathcal{L}_b = y \log(P(y^+|p)) + (1 - y) \log(1 - P(y^+|p)) \tag{2.23}$$

where y is the pair label and $P(y^+|p)$ is the probability of being y^+ given the pair p. $P(y^+|p)$ is estimated by the siamese neural network.

2.5 Studies on Bug deduplication

In the literature, several works have proposed methods to address bug deduplication. In Appendix A, we summarize the main aspects (e.g, evaluation methodology, textual data feature extraction, novelty) of such works. According to Lin et al. [62], studies on bug deduplication can be grouped into three main categories based on the evaluation methodology:

1. **Decision-making approach.** Methods are evaluated regarding their capability to correctly predict pairs of bug reports as positive or negative. A pair is labeled as positive when it is composed of duplicate reports; otherwise, it is considered as a negative example. Classic binary metrics (e.g., accuracy, precision, recall, F1, and AUC) are used to measure the performance of a model in labeling pairs. Works grouped in this approach have achieved excellent results, e.g., Lazar et al. [63] reported achieving 99% in all metrics with their method. However, such performances are highly overestimated since they employed a number of negative pairs for evaluation that is much smaller than what observed in real environments. For instance, Lazar et al. [63] employed a ratio of four non-duplicate pairs for each duplicate one. Based on such ratio, we can

only generate 15,848 negative pairs[1] regarding the validation set in the Eclipse dataset employed in Chapter 4. This number is less than 0.01% of the 286 million possible negative pairs that can be generated by simply combining each duplicate report in a validation set with a report in the training set.

2. **Binary classification approach.** Instead of using pair of reports, binary classification approach consists of a group of methods that predict whether *bug reports are duplicate or not*. For each new report, methods extract features from it and use such features to calculate its similarity to previously submitted reports. After report comparisons, they summarize the results using aggregation functions (e.g., mean and max) or any other summarization process. A classifier uses the information from summarization and the features extracted from the new report to predict whether such report is duplicate or not. This an ideal scenario since a model can automatically detect and label duplicate reports without any human assistance. However, this compelling scenario is hard, and the current techniques do not perform well. For instance, Banerjee et al. [15] method incorrectly classifies 36% of all the duplicate reports in the Eclipse, which means that around 14,000 duplicate reports could still be assigned to the developers. Furthermore, this same method wrongly classifies 26% of the non-duplicate bug reports in the same dataset.

3. **Ranking approach.** The ranking approach acknowledges that the recent techniques are not mature enough to automatically detect the duplicate reports without harming the maintenance software process. Given that, methods of this approach generate a list of the k most likely duplicate reports of a specific given report. The triager receives this list and, then, identifies if a report is duplicate using only reports from the ranking list. Thus, instead of searching and examining hundreds of reports in a BTS, the triager only focus on the K recommended reports. Therefore, in such approach, works focus on evaluating the quality of ranking lists produced by their methods. It is worthy to mention that the majority of the methods in the literature are encompassed by the ranking approach.

In order to address bug deduplication, methods extract features that are related to a single report, or derived by comparing data from two or more reports. After such extraction, a similarity comparison or classification is performed by passing these features as input to simple functions (e.g, weighted average, linear combination and cosine similarity) or even complex models as Support Vectors Machines (SVMs), decision trees, and MLPs.

[1]We generate all possible combinations of positive pairs from reports in the same buckets. For such pair generation, we only consider buckets whose at least one report is within the validation set.

In bug deduplication, different information sources were employed to extract valuable features for this task. Sun et al. [14] demonstrated that features derived from categorical data (e.g., product and component) can improve method effectiveness. Following Sun et al. [14], several works [1, 63, 64, 65, 66] extracted a Boolean feature from each categorical field in which its value is one when two reports contain the same categorical values. On the other hand, some studies based on deep learning methods represented categorical values as embeddings and directly compared such representations using MLP or cosine similarity [67, 68, 69].

Besides categorical data, Wang et al. [9] proposed to employ the sequence of subroutines executed during an execution for addressing bug deduplication. Considering that each subroutine signature is a term, such sequences are encoded as vectors using TF-IDF and their similarities are measured by means of cosine similarity. In their experiments, they showed that the sequence of subroutine calls provides complementary and useful information for this task.

Feng et al. [70] demonstrated that user profile information and textual data from comments are valuable for duplicate bug report detection. In this work, a SVM is used to compute a weight for each comment in a report. Each weight measures the comment usefulness for fixing a bug and it is used to adjust the similarity score between a comment and textual data from a new report. Moreover, they generated features based on submission history that captures reporter knowledge and experience.

Hindle et al. [1] and Aggarwal et al. [66] proposed to improve the deduplication performance by generating contextual features through pre-defined word lists. While the former extracts such list from software-engineering textbooks and open-source project documentations, the latter employ non-functional requirement terms provided in [71] and a set of architecture words manually created by the authors. Contextual features are extracted by calculating BM25F score between each word list and textual data in the reports.

Cooper et al. [72] proposed a deep learning model to deduplicate bug reports using, in addition to textual data, the videos attached to them. Such model extracts features from videos by means of SimCLR [73], a Convolutional Neural Network (CNN) that converts video frames into a embedding. After such encoding, the similarity of two videos is the average of two scores: (i) the cosine similarity of video representations generated by a bag of visual words model based on k-means clustering algorithm and TF-IDF; and (ii) longest common subsequence (LCS) between the videos.

In the remaining of this section, we describe the proposed feature extraction techniques for textual data within summary and description fields.

Methods based on VSM Several works [9, 70, 72, 74, 75, 76, 77, 78] represented summary and description content in VSM using different term weighting schemes. The log of the term frequencies was used to compute vector weights in [76, 74], whereas TF-IDF was employed in [9, 72, 77, 79]. On the other hand, Yang et al. [80] and Lin et al. [62] calculated the vector values by means of BM25 term weighting: a scheme similar to Equation 2.3 that receives a term instead of a query. Moreover, Lin and Yang [78] proposed to employ Class-Feature-Centroid (CFC) weighting scheme for bug deduplication. Such scheme computes the weight of a term t in a field f as follows:

$$wc_{tf} = \mathrm{TF}(t, f) \times \mathrm{IDF}(t) \times b^{\frac{df_B(t)}{|B|}} \log(\frac{|B|}{bf(t)}), \tag{2.24}$$

where B is the bucket that contains the report of a field f, b is a parameter, $df_B(\cdot)$ is the number of reports in B that contains a term t, and $bf(\cdot)$ is the number of buckets in which t appears. Instead of generating a representation for each report, Prifti et al. [75] encoded a bucket B as a vector whose dimensions correspond to the number of reports within B that contains a specific term in the summary and description. Such dimension values are normalized by the numbers of reports in B. In these works, vector representations were compared using either cosine function [62, 74, 76, 78, 79] or Lucene's scoring function [72].

Methods based on BM25 Sun et al. [81] compared the similarity of two textual data by means of a special case of BM25 where $k1 = 0$, i.e., only IDF affects the similarity score in Equation 2.3. Moreover, since bug reports contain multiple textual fields, few works [82, 1, 66] employed BM25F for measuring textual similarity. One drawback of BM25F is that it was developed for traditional search engines where queries are usually short. Since reports can contain texts with more than one hundred words, Sun et al. [14] proposed to extend BM25F by adding a component based on the frequency of query terms. Such extension, called $BM25F_{EXT}$, is computed as follows:

$$BM25F_{EXT}(d, q) = \sum_{t \in d \cap q} \mathrm{IDF}(t) \times \frac{\mathrm{TF}_D(d, t)}{k_1 + \mathrm{TF}_D(d, t)} \times \frac{(k_3 + 1)\,\mathrm{TF}_Q(d, t)}{k_3 + \mathrm{TF}_Q(d, t)}, \tag{2.25}$$

where k_3 is a scalar value and TF_Q is defined as:

$$\mathrm{TF}_Q(q, t) = \sum_{f \in q} w_f \times \mathrm{TF}(t, f). \tag{2.26}$$

Methods based on Natural Language Preprocessing. Sureka and Jalote [83] proposed to employ a n-gram model at character-level for bug deduplication. For each text in a report, they extracted n-grams of lengths between 4 and 10 from all characters in the text (including symbols and spaces). The rationale behind this method is that, in comparison to the word-level, the character-level is more robust to noisy data, is more language independent, and can better handle abbreviation and morphological word variations. The authors measured the similarity of two reports based on the numbers of n-gram characters shared between textual data from two reports. Lazar et al. [63] proposed to extract features from textual data by means of a system, called *TakeLab* [84]. Such system captures the semantic similarity of short texts.

Methods based on Longest Common Subsequence. Multiple works [4, 85, 15] compare two texts by finding the Longest Common Subsequence between them. In Banerjee et al. [4], they proposed to normalize the value of the longest subsequence by the length of the matched textual data.

Methods based on topic modeling. Topic Modeling consists in unsupervised techniques that capture latent topics within a set of documents. Few studies [86, 87, 67] learned topics in bug reports by means of Latent Dirichlet Allocation (LDA). Rocha and Carvalho [67] compared topic distributions by means of MLP while cosine similarity was employed in Zou et al. [86] and Budhiraja et al. [87]. Based on LDA, Nguyen et al. [82] proposed a method, called *T-Model*, which considers that there are two different type of topics: one is shared among duplicate reports in the same bucket while the other is shared among reports in the BTS. T-Model learns these topics akin LDA and Jensen-Shannon divergence is used to compute the similarity of topic distribution between two reports.

Methods based on deep learning. Some works [62, 79, 88] used popular unsupervised techniques to learn word embeddings. These studies represented reports as the average vectors of their word embeddings and such representations were compared through cosine similarity.

Several siamese neural networks [67, 68, 69, 89, 90, 91] were proposed for bug deduplication. In [89, 90, 91], embeddings of textual data from summary and description were generated by the same encoder. While Budhiraja et al. [89] employed a simple encoder that computes the average of the word embeddings, Xie et al. [90] encoded texts as vector by means of a standard CNN. Instead of generating a single representation, Kukkar et al. [91] employed a CNN to encode each sentence within a text into an embedding. After the sentence embedding generation, a similarity matrix is derived by computing the cosine similarity of each possible pair of sentence representations within two bug reports. In contrast to previous works, Deshmukh et al. [68] proposed to separately encode summary and description to vectors by means of a LSTM and CNN, respectively. Instead of using LSTM and CNN, Rocha and Carvalho [67] independently generated the vector representation of summary and description using two distinct Bidirectional Encoder Representations from Transformers (BERTs), a well-known model based mainly on attention mechanisms.

In order to allow textual information interaction during feature extraction, Poddar et al. [19] proposed a siamese neural network based on attention that simultaneously learns latent topics and classifies whether a pair of report is duplicate or not. To generate a fixed-length representation for a report of interest r, an attention mechanism computes the attention score of each word embedding x_i^r in r by comparing x_i^r to the average vector of the word embeddings in the other report. Thus, regarding Equation 2.19, h_i^e is the i-th the word embedding x_i^r in r, and h_{t-1}^d is the average vector of the word embeddings in the other report. Following Equation 2.20, the representation of r is the weighted average of the embeddings in r whose weights are the attention scores. Latter, He et al. [92] proposed to jointly extract textual features from two reports by using a CNN with dual-channel. Each channel contains a matrix derived from the concatenation of word embeddings of a report.

2.6 Studies on Crash report deduplication

In the literature, the majority of techniques addressed crash report deduplication by mainly comparing stack traces. In [10, 93, 94], stack traces were vectorized by means of VSM and TF-IDF. As preprocessing step, Lerch and Mezini [93] and Campbell et al. [10] tokenized stack traces by punctuations and spaces, respectively. After that, the similarity of two stack traces was measured by Lucene's similarity function. In contrast to both works, Sabor et al. [94] extracted only the package names of subroutines from Java's stack traces to avoid the curse of dimensionality problem. Based on package information, they compared two stack traces by means of cosine similarity.

In order to preserve sequence order information, several methods [13, 21, 22, 95, 96] based on sequence matching algorithms were proposed to measure stack trace similarity. In Modani et al. [95], the similarity of two stack traces was equal to the length of the shared prefix between them divided by the longest stack trace length. Dhaliwal et al. [13] and Bartz et al. [96] applied edit distance algorithm to compare stack traces. While the first work employed the standard algorithm, the second one differentiate distinct subtypes of insertion, deletion and substitution operations that contained their own constant penalty values. For insertion or deletion, the penalty depends whether the new or deleted frame comes from the same C++ module of the previous frame. Three substitution penalties were defined based on the differences between modules, subroutine names, and offsets in two frames.

Related to edit distance, Brodie et al. [21] and Dang et al. [22] proposed variants of NW Algorithm. In the context of crash report deduplication, s_i and s'_j are, respectively, the i-th and j-th frames within the stack traces s and s'. Thus, match$(\cdot)$ is defined as follows in Brodie et al. [21]:

$$\text{match}(s_i, s'_j) = 1 - \frac{df_s(s_i)}{|S|} \times 1 - \frac{i}{|s|} \times e^{-\frac{|i-j|}{2}}, \tag{2.27}$$

where $df_s(s_i)$ is the number of stack traces that contains s_i, $|S|$ is the number of stack traces in the repository, and $|s|$ is the length of the stack trace s. As one can observe, the match value is variable and depends on the frame global frequencies and positions. On the other hand, mismatch and gap values are fixed in Brodie et al. [21]: gap$(s_k) = c$ and mismatch$(s_i, s'_j) = 0$, where c is a constant. Similarly, Dang et al. [22] employed constants values for gap and mismatch values: gap$(\cdot)$ = mismatch$(\cdot) = 0$. However, in contrast to Brodie et al. [21], besides only considering the position information of frames, they included parameters to match$(\cdot)$ which improves the method adaptability in different environments. In Dang et al. [22], the match value is computed as follows:

$$\text{match}(s_i, s'_j) = e^{-u \times \min(i,j)} e^{-v|i-j|}, \tag{2.28}$$

where u and v are scalar parameters that control, respectively, the effect of the absolute position and position difference of the frames on the match value. It is worthy to note that, in the sequence matching algorithm, two frames are considered equivalent when their identifiers are the same. Usually, a frame identifier is defined as the subroutine name.

Khvorov et al. [97] proposed a siamese neural network, called *S3M*, to measure the stack trace similarity. A Bi-LSTM encodes the sequence of frames into a embedding. In order to compare stack traces embeddings, three vectors are generated by computing the absolute difference, arithmetic mean, and multiplication component-wise between their distributed representations. Finally, based on such three vectors, a MLP calculates the similarity of the stack traces.

In contrast to the majority of the literature, three works [10, 94, 96] addressed bug deduplication by also employing information out of the stack traces. Sabor et al. [94] compared two reports by means of a linear combination of the stack trace similarity and two Boolean features based on differences of report severity and component fields. Campbell et al. [10] included the information related to the execution environment in the same vector space of the stack traces. In Bartz et al. [96], a logistic regression outputs the probability of two reports being duplicate based on the edit distance score between two stack traces and three Boolean features that compare the exception code, process name, and event type of the reports, respectively.

In [98, 99, 100], crash report deduplication is performed by comparing a new report to buckets. Kim et al. [98] proposed a method, called *CrashGraph*, in which buckets are represented as graphs. Considering all stack traces within a bucket, each subroutine is a node and each edge symbolizes two adjacent subroutines in a stack trace. The equivalent representation is performed for a new stack trace in a report. Thus, the percentage of edges shared between report and bucket graphs is used to measure the similarity between them. Koopaei and Hamou-Lhadj [99] proposed a method, called CrashAutomata. In CrashAutomata, n-grams are created for each stack trace and only n-grams whose frequencies are below a threshold are kept. Based on the generated n-grams, they build an automata for each bucket by means of the algorithm proposed by Jiang et al. [101]. New stack traces are included to buckets based on their automata. Ebrahimi et al. [100] proposed to train a Hidden Markov Model (HMM) for each bucket. The HMM computes the probability of a stack trace to belong to a bucket.

In order to reduce computational cost of comparing new reports to submitted ones, Dhaliwal et al. [13] proposed to filter reports based on the topmost frames before applying the edit distance algorithm. Latter, Moroo et al. [102] proposed a re-ranking scheme that first computes similarity scores between a new report and submitted ones based on Campbell et al. [10]. Then, a ranking list is created by selecting the top-k most similar submitted reports to a new one. After that, considering only reports in the ranking list, they compute a new similarity score by means of the method proposed by Dang et al. [22]. Finally, they updated the similarity score of k reports in the list by computing the weighted harmonic mean of the two previous computed scores.

In literature, three distinct evaluation methodologies were introduced to crash report deduplication: *ranking*, *binary classification* and *clustering*. The two first approaches are equivalent to the ones proposed for bug deduplication. The last one focus on measuring the quality of clusters generated by a method using clustering metrics (e.g., Purity and BCubed). In such approach, due to high computation cost associated to clustering techniques, a new report is usually assigned to a existing bucket when the most similar submitted report to a query is greater than a threshold, otherwise it is considered as a singleton.

In Appendix B, we compile the primary aspects of the studies proposed for crash report deduplication.

2.7 Discussion

In the literature, a vast number of works addressed bug deduplication based on classical techniques, e.g., TF-IDF, BM25, and LDA. Nonetheless, the proposed methods still exhibited performances that are still far from a satisfactory deployment of automated deduplication systems in real environments [68]. Thus, considering the current rise of deep learning, many works proposed methods that leverage the power of such model to achieve better performance. The majority of those methods are siamese neural networks that encode textual data of a report into a dense vector.

A shortcoming of siamese neural networks is that there is no interaction between textual data of two reports before generating a report embedding, i.e., the encoder always extracts the same set of features from a report independently of the comparison. However, such features might not be relevant for a particular scenario, even though they are discriminative in general. Poddar et al. [19] leveraged attention mechanism to attenuate such information loss by encoding the report of interest into a embedding based on the average vector of the word embeddings in the other report. However, such fixed-length representation of the other report might be a bottleneck since its generation process may lose information that are valuable for a particular deduplication case. We argue that a model could learn a more useful report representation for bug deduplication by allowing it to focus on distinct portions of the reports during feature extraction. This could mitigate information loss and, thus, improve the method performance.

As demonstrated by empirical evidences [8], bugs are usually contained in the topmost frames of the stack traces. Thus, it is intuitive to consider that such positions are more important than the ones at the bottom for crash report deduplication. Nevertheless, only three studies [21, 22, 97] have proposed methods which similarity score are affected by the frame positions. In Khvorov et al. [97], LSTM is able to learn position information of frames. However, deep learning models require a considerable amount of labeled data which is not always available. Brodie et al. [21] and Dang et al. [22] included the absolute frame positions and their differences on their method scoring scheme. However, both works only consider such information for matches. Thus, the alignment score is equally penalized by mismatch and gap alignments that occur at the top and bottom of the stack traces. This is counter-intuitive since we expect that dissimilarity on the topmost frames to be more important than the ones in the bottom. Moreover, neither of these two works used both frame rarity information and parameters in their scoring scheme. We believe that a method could be more effective by: (i) considering frame position and global frequency information for all alignments; and (ii) including parameters that control the effect of these pieces of information.

For crash report deduplication, a shortcoming of methods based on NW algorithm is its quadratic time complexity to compare two stack traces. In industry scenarios, such systems can receive million of reports per day. Thus, to achieve a more adequate throughput, real deduplication systems may employ less expensive methods to compare stack traces as prefix match [95] and those based on TF-IDF. Despite their linear time complexity for computing stack trace similarities, prefix match is high sensitive to minor differences in the top positions while techniques based on TF-IDF loses position information. Due to such shortcomings, they are less effective than the ones based on the NW algorithm. Therefore, a research avenue for crash report deduplication is to investigate techniques that compare stack traces in linear time without performance degradation.

In the literature of both bug deduplication and crash report deduplication, methods have not been extensively compared among themselves and no standard datasets were proposed for evaluation. Moreover, source code and data are not usually publicly available which negatively impact reproducibility. Finally, as presented before, distinct methodology approaches were proposed in both tasks. However, to the best of our knowledge, no study has investigated such methodologies and compare their advantages and limitations. Thus, there is a lack of consensus regarding the evaluation methodology.

CHAPTER 3 OVERVIEW

This thesis is composed of three articles that are presented in submission chronological order in chapters 4, 5 and 6. These chapters address each one of the research objectives defined in Chapter 1 and they are briefly described on the following.

In Chapter 4, we propose a *Soft Alignment Model for Bug Deduplication* (SABD) that generates a joint representation of textual data in bug reports. Leveraging attention mechanisms, SABD can dynamically focus on distinct segments of reports during feature extraction. This helps to mitigate information loss when fixed-length representations are generated. We evaluate SABD and competitive techniques following a methodology proposed by Sun et al. [14]. In our experiments, SABD outperforms siamese neural networks and strong baselines in all datasets. Moreover, an ablation study demonstrates that the proposed mechanism to jointly generate representations is crucial for bug deduplication.

In Chapter 5, we propose *TraceSim*, a novel optimal global alignment method for crash report deduplication. TraceSim computes a weight for each frame that depends on two pieces of information: frame position and global frequency. In contrast to previous works [21, 22], the values of mismatches, matches and gaps are variable and computed based on frame weights. Moreover, TraceSim contains parameters that control the impact of frame position and global frequency in the alignment score. To evaluate method effectiveness, we propose a new evaluation methodology for crash report deduplication that combines binary classification and ranking approaches from the literature of bug deduplication. In a nutshell, our methodology can evaluate a method capability to: (i) distinguish duplicate and non-duplicate reports; (ii) assign a report to its correct bucket; and (iii) rank similar reports. In our experiments, TraceSim consistently achieves competitive performances in all evaluation scenarios and, in the majority of them, TraceSim significantly surpasses the competitive methods. Moreover, an ablation study demonstrates: (i) the effectiveness of each TraceSim's component; and (ii) the importance to use frame position and global frequency for computing mismatch and gap values.

In Chapter 6, motivated by the high computational cost of optimal sequence alignment for comparing stack traces, we propose a Fast Stack Trace alignment method for crash report deduplication (*FaST*). In summary, *FaST* computes the similarity of two stack traces by independently aligning frames with the same identifier. The rationale behind this is that similar stack traces are expected to share important subroutines in near absolute positions. For aligning frames of a specific subroutine, *FaST* employs a simple alignment heuristic based on the fact that positions closer to the top of the stack traces tends to be more relevant than the ones at the bottom. In contrast to optimal sequence alignment methods, *FaST* compares two stack traces in linear time of their lengths. In our experiments, we demonstrate that *FaST* is not only substantially faster than TraceSim and other methods based on the NW algorithm, but it also achieves SOTA performance in all datasets according to varied metrics.

CHAPTER 4 ARTICLE 1: A SOFT ALIGNMENT MODEL FOR BUG DEDUPLICATION

Authors. Irving Muller Rodrigues, Daniel Aloise, Eraldo Rezende Fernandes, and Michel Dagenais. Published at the 17th International Conference on Mining Software Repositories (MSR 2020)[1]

Abstract. Bug tracking systems (BTS) are widely used in software projects. An important task in such systems consists of identifying duplicate bug reports, i.e., distinct reports related to the same software issue. For several reasons, reporting bugs that have already been reported is quite frequent, making their manual triage impractical in large BTSs. In this paper, we present a novel deep learning network based on soft-attention alignment to improve duplicate bug report detection. For a given pair of possibly duplicate reports, the attention mechanism computes interdependent representations for each report, which is more powerful than previous approaches. We evaluate our model on four well-known datasets derived from BTSs of four popular open-source projects. Our evaluation is based on a ranking-based metric, which is more realistic than decision-making metrics used in many previous works. Achieved results demonstrate that our model outperforms state-of-the-art systems and strong baselines in different scenarios. Finally, an ablation study is performed to confirm that the proposed architecture improves the duplicate bug reports detection.

Keywords. Bug Tracking Systems, Duplicate Bug Report Detection, Deep Learning, Attention Mechanism

4.1 Introduction

Bug fixing accounts for a substantial part of any software development project. Thus, many projects make use of a *bug tracking system* (BTS) to manage and track bug reports. One important task in such systems is to identify duplicate bug reports, i.e., distinct reports describing issues caused by the same bug in the software. It is crucial to perform this task as fast as possible in order to prevent developers from spending time looking for bugs already fixed. Usually, a triage team manually labels new reports as duplicate or not [14]. However, especially in open source projects, bug reports can be submitted by developers, testers and even end users. This heterogeneous environment leads to many duplicate bug reports. For

[1]Available at [103]

example, 12% of all reports are duplicate in one Eclipse instance [15]. Therefore, devising automatic methods to detect duplicate bug reports is crucial for efficient software maintenance. In the literature, such a task is called duplicate bug report detection, bug report deduplication or, simply, *bug deduplication*.

Typically, a bug report comprises a summary, a description, and some categorical fields (e.g., system, component and version). Regarding textual data, for simplicity, the terms *word* and *token* are considered interchangeable in this work. One core component of most methods in bug deduplication is a similarity function that compares a pair of reports. How this function is composed and used vary greatly from one method to another. A handful of studies [19, 68, 89, 90] employ deep neural networks in order to model similarity functions. Deshmukh et al. [68], Budhiraja et al. [89] and Xie et al. [90] works are based on Siamese neural networks [16] that generate the representation of one bug report without considering the other report content. This independent representation is limited specially for textual data, since it may focus on generic features that are not relevant for a specific comparison [18]. Poddar et al. [19] try to mitigate that shortcoming by employing an architecture that exchanges information between the reports during feature extraction. This approach generates a joint representation based on *attention* [20], in which the representation of a word in a report attends to a *pooled representation* of all words in the other report.

In this work, we propose a novel deep learning network that produces joint representations of reports based on a soft-attention alignment mechanism [104]. The key component of this model is a layer that compares each word in a report with a fixed-length representation of all words in the other one. While Poddar et al. [19] also use an attention mechanism, our proposed architecture is able to summarize relevant information within one report conditioned to a specific segment of the other report. This provides a more powerful representation of textual data.

Many previous works on bug deduplication have employed an evaluation methodology called *decision-making approach* [62]. This evaluation is based on pairs of reports labeled as positive when they refer to the same bug or negative otherwise. Positive pairs comprise all possible pairs within a set of duplicate reports. While negative pairs are generated using some sampling technique. Model performance is then measured by means of ordinary classification metrics (like accuracy, precision and recall) over the *generated set* of positive and negative pairs. The decision-making approach is quite unrealistic, since the real scenario presents a much larger set of negative candidates. When a new report is submitted to a BTS, all previously submitted reports are duplicate candidates. Thus, this evaluation methodology highly overestimate performance. Another popular evaluation methodology is the ranking approach.

It acknowledges that the current techniques are not accurate enough to automatically detect duplicate reports with no human intervention. Therefore, in this approach, for a given new report, the proposed methods generate a list of the K most likely duplicate reports. The triager then identify whether a report is duplicate considering only the reports from the recommendation list. Instead of searching and examining hundreds of reports in the BTS, the triager can focus on the K recommended reports.

We experimentally evaluate our model by means of a ranking methodology based on Sun et al. [81]. We report on experiments using four well-known datasets derived from BTSs of open-source projects, namely Eclipse, Mozilla, NetBeans and OpenOffice. Bug deduplication in open-source projects is particularly challenging because any user can submit a bug report in their BTS and the knowledge of these users about the system may vary significantly. State-of-the-art systems and some strong baselines are compared to the proposed model in several scenarios. Additionally, we perform an ablation study to assess different aspects of our model.

The main contributions of this paper are summarized as follows:

1. We propose a soft-alignment model that is based on a more powerful architecture than previous methods.

2. Our method and the baselines are evaluated using a more realistic methodology. This work is the first to compare different deep learning methods using the ranking approach.

3. Our method achieves state-of-the-art performance on all considered datasets.

4.2 Related Work

Several non-deep learning methods in the literature address the bug deduplication as a ranking problem. Runeson et al. [76] are the first to use NLP techniques to approach duplicate bug report detection. They measure report similarity by computing the cosine similarity between bag-of-words vectors. Wang et al. [9] detected duplicate reports by combining function calls during the system execution with textual data. Sun et al. [81] trained an SVM to estimate the probability of reports being duplicate by receiving 54 textual similarity features generated from different combinations of text origins, n-gram lengths and dictionaries. Sureka and Jalote [83] proposed a similarity function whose output is proportional to the quantity of n-gram of characters in common between two reports. Sun et al. [14] proposed $\text{BM25F}_{\text{ext}}$ and REP for bug deduplication. $\text{BM25F}_{\text{ext}}$ is an extension of BM25F specially designed to address scenarios in which queries are long sentences with few or no duplicate words. REP is a similarity function that linearly combines BM25F$_{\text{ext}}$ scores using unigram and bigram with features generated from categorical data comparisons. Prifti et al. [75] developed a method to rank reports using a time window and a unique representation for each master group. Nguyen et al. [82] proposed a method, called *DBTM*, that linearly combines the BM25F score and the topic similarity computed by a model based on Latent Dirichlet Allocation (LDA). Banerjee et al. [4] addressed the bug report deduplication by using the longest common subsequence between the bug reports. Zhou and Zhang [77] proposed a linear model, called *BugSim*, which is trained to minimize the fidelity loss of triplets using features inspired by Nallapati [105]. In Yang et al. [80], BM25 is used to weight the bag-of-words vectors which are compared by the cosine similarity. Banerjee et al. [85] generated a top-20 list for different similarity measures and aggregated them into a unique list using two fusion approaches: one retrieves the maximum score of a report in the lists while the other sums the similarity scores of the reports. Lerch and Mezini [93] proposed to use the stack trace in the bug report to better detect duplicate bug reports. Sabor et al. [94] improved Lerch and Mezini [93] by employing only packages names instead of full method names. Lin and Yang [78] combined TF-IDF with a weighting scheme based on the term relations within the clusters of reports. Lin et al. [62] trained an SVM to estimate the duplication probability using cluster-based correlation features, the BM25F score and the cosine similarity of word vectors. Yang et al. [79] designed similarity function whose output depends on product and component differences, the cosine similarity of TF-IDF vectors and the average of word embeddings. Budhiraja et al. [87] proposed LWE which combines LDA with the word embeddings.

Regarding deep learning methods, Xie et al. [90] proposed a convolutional neural network (CNN), called DBR-CNN, to classify pairs of duplicate bug reports. In their architecture, a shared CNN independently encodes the textual data of the pair of reports into two vectors. A logistic regression then classifies each pair of reports by receiving the cosine similarity of those vectors and a set of features related to categorical data. In NLP, statistical methods parse textual data from documents to discover latent themes, called topics, which are common between multiple documents [106] (e.g., bug reports that contain the words `combo` or `font` can be related to the topic *UI*). Poddar et al. [19] proposed a neural network that simultaneously learns to cluster reports based on topics while detecting duplicate pairs. A recurrent neural network (RNN) represents each word of a report as a vector. The k-first dimensions of these vectors are trained to yield high similarities to words that are in the same topic. For the classification, Poddar et al. [19] generate the joint-representation of a report as a weighted average of its word vectors. An attention mechanism calculates these weights by using the self-attention coefficients of the topic information and the element-wise multiplication of the word representations in the report with the mean pooling of all words in the other report.

The authors used only summary data from the reports in their experiments.

Budhiraja et al. [89] proposed a neural network, called DWEN, in which the fixed-length representation of a report is the mean of their word vectors and the classifier is a multi-layer perceptron (MLP) that receives only the representation of a pair of reports. Deshmukh et al. [68] proposed two siamese neural networks for bug deduplication. The authors used a feed-forward neural network, a CNN, and a bidirectional LSTM to encode, respectively, the categorical data, the description, and the summary into vectors. The concatenation of these encoder outputs generates the fixed-length representation of the reports. Deshmukh et al. [68] proposed two approaches to calculate the similarity of the report representations. The first one, called Siamese Triplet, is trained to minimize a hinge loss given a set of *triplets* and employs the cosine function to compute the similarity between two reports. The second one, called Siamese Pair, uses the binary cross-entropy loss of pair in the training and scores the similarity between reports using a MLP.

This paper presents a method that improves the representation generation found in the previous deep learning approaches. Different from Budhiraja et al. [89], Deshmukh et al. [68], and Xie et al. [90], our model exchanges information between the reports before encoding textual data into a fixed-length vector. Moreover, in the feature extraction, our method can dynamically focus on different segments of a report instead of providing a unique set of features from it, as done by Poddar et al. [19]. This more powerful architecture can reduce information loss in the representation generation thereby improving the duplicate bug report detection.

4.3 Soft Alignment Model for Bug Deduplication

In this section, we describe our proposed *Soft Alignment Model for Bug Deduplication* (SABD). This model receives a pair of bug reports: a new query report q and a candidate report c previously submitted to the repository. The model outputs the probability $P(y|q, c)$ of q being a duplicate of c, where y indicates whether the given reports are duplicate ($y = 1$) or not ($y = 0$). We consider a bug report to be composed of the categorical fields, a summary and a description. Given a query report q, the values of its categorical fields are represented as the tuple q^{cat} while the sequence of words of its summary and description are denoted as q^s and q^d, respectively. The same notation is employed for the candidate c.

Figure 4.1 depicts the SABD architecture. As we can see, SABD is composed of the categorical and textual modules (two sub-networks) that independently compare the categorical and textual data from both reports, respectively. The classifier receives these module outputs and

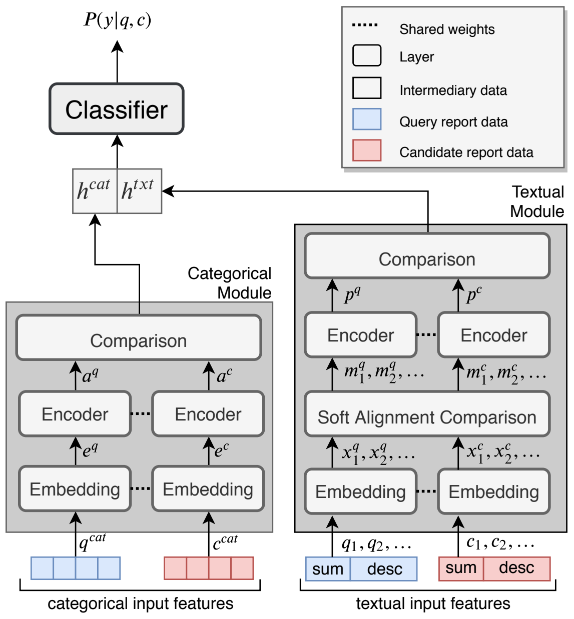


Figure 4.1 SABD architecture overview.

produces the final prediction $P(y|q,c)$. While the categorical module is a straightforward dense neural network, a more sophisticated architecture is employed by the textual module to handle text. The core of this module is the *soft alignment comparison layer* that allows the model to dynamically access distinct information from the text. This mechanism is expected to improve the model capacity to focus on relevant features in the textual data for the deduplication. In the remainder, we describe the details of SABD and its modules.

4.3.1 Categorical Module

The categorical module is composed of three layers: embedding, encoder, and comparison. In the embedding layer, each categorical field is related to a parameterized lookup table that links the field value to a real-valued vector. This representation is more powerful than using binary variables (e.g., feature is 1 if and only if field values are equal) since it allows the model to group similar field values. Given the query q, the embedding layer concatenates the real-valued vectors of each categorical value in q and outputs $e^q \in \mathbb{R}^{cl \cdot d^{cat}}$, where cl is the number of categorical fields in the report and d^{cat} is a hyperparameter indicating the categorical vector dimensions. The encoder layer receives the embedding layer output e^q and generates a fixed-representation a^q of the categorical data from q such that:

$$a^q = ReLU(W^a e^q + b^a), \tag{4.1}$$

where $W^a \in \mathbb{R}^{d^a \times (cl \cdot d^{cat})}$ is the weight matrix parameter, $b^a \in \mathbb{R}^{d^a}$ is the bias parameter, and d^a is a hyperparameter that controls the layer size. Analogously, the fixed representation a^c is produced for the categorical data of candidate c.

After encoding the categorical features into vectors a^q and a^c, the categorical comparison layer computes a comparative representation of these two vectors by a simple operation given by:

$$\text{cmp}(a^q, a^c) = [(a^q - a^c)^2; a^q \odot a^c], \tag{4.2}$$

where $[\ \cdot\ ; \ldots ;\ \cdot\ ]$ is the concatenation operator and $\odot$ represents element-wise multiplication. Finally, given the comparative representation, a fully connected (FC) layer computes the comparison layer output as:

$$h^{cat} = ReLU(W^h[a^q; a^c; \text{cmp}(a^q, a^c)] + b^h), \tag{4.3}$$

where $W^h \in \mathbb{R}^{d^h \times 4d^a}$ is the FC weight matrix, $b^h \in \mathbb{R}^{d^h}$ is the FC bias vector, and d^h is a hyperparameter.

4.3.2 Textual Module

Although categorical features are relevant to solving bug deduplication, the most informative features are the summary and description texts. Thus, the core of our model is the textual module that compares the textual features of the query and candidate reports (i.e., q^s, q^d, c^s and c^d). It comprises four layers: *textual embedding*, *soft alignment comparison*, *textual encoder*, and *textual comparison*.

Textual Embedding Layer

This layer independently transforms the words from the query and candidate texts into real-vectors (word embeddings). A pre-trained look-up table in this layer links each word in the summary and description of a report to an embedding. This word representation loses information about the word origin since the same look-up table is used for both textual fields. Previous works [14, 68] present evidences that they are both important for bug deduplication. Moreover, it is important to distinguish summary and description words since each field presents unique characteristics. Consequently, two distinct real-vectors are employed to distinguish whether a word comes from the summary or description. Such representation is denominated as *field embedding*. Given a query q, the summary q^s and description q^d are concatenated into a single sequence q^t whose i-th word is represented as: $v_i^q = [w_i^q; f_i^q] \in \mathbb{R}^{d^w + d^f}$, where w_i^q is the word embedding, f_i^q is the field embedding, and d^w

and d^f are hyperparameters indicating their respective vector dimensions.

Although the field embeddings are learned in the learning phase, word embedding vectors are not fine-tuned during the training because it increases computation cost, limits vocabulary size and can lead to overfitting [107]. Instead of updating word embedding parameters, we provide each word representation v_i^q to a fully connected layer (FC) along with residual connections:

$$x_i^q = v_i^q + \text{ReLU}(W^x v_i^q + b^x), \tag{4.4}$$

where $W^x \in \mathbb{R}^{d^x \times d^x}$ is the FC weight matrix, $b^x \in \mathbb{R}^{d^x}$ is the FC bias vector, and $d^x = d^f + d^w$. This solution not only reduces computation cost, memory usage, and model complexity but also allows the model to project words into a more meaningful feature space. In the end, textual embedding layer receives q^s and q^d and outputs a sequence of embedding vectors $x^q = (x_1^q, x_2^q, \ldots, x_{|q^t|}^q)$, where $|q^t|$ is the length of q^t.

Similarly, for the candidate report c, a sequence of embedding vectors $x^c = (x_1^c, x_2^c, \ldots, x_{|c^t|}^c)$ are provided by an identical embedding layer, where c^t is the concatenation of summary and description words in the candidate report and $|c^t|$ is the length of c^t. Again, as indicated in Figure 4.1, query and candidate textual embedding layers share their parameters.

Soft Alignment Comparison Layer

Previous deep learning methods for bug deduplication [19, 68, 89, 90] encode query and candidate reports without or with limited data exchange. SABD overcomes this limitation with an architecture that provides a more powerful feature interaction. The core of this layer is the soft-attention alignment [104]. This attention mechanism computes a similarity score s_{ij} between query token q_i^t and candidate token c_j^t as such:

$$s_{ij} = \frac{(x_i^q)^T \cdot x_j^c}{\sqrt{d^x}}. \tag{4.5}$$

The previous equation is known as the scaled dot-product [108].

In order to accentuate important features of the words contained in a report, the soft alignment comparison layer must have access to textual information from the other report. However, texts are variable-length data and can contain a large set of potential relevant features. Thus, this layer uses the similarity score to select features from the report words that are related to a specific word in the other report. These features are encoded into a fixed-length representation. More precisely, each word vector x_i^q in the query attends to all candidate vectors $x^c_1, x^c_2, \ldots, x^c_{|c^t|}$, in order to produce a fixed-length representation:

$$\overline{x}^q_i = \sum_{j=1}^{|c^t|} \alpha^{q_i}_j x^c_j, \tag{4.6}$$

where $\alpha^{q_i}_j = exp(s_{ij}) / \sum_{k=1}^{|c^t|} exp(s_{ik})$ is called *attention score* and represents the normalized similarity score. $\overline{x}^q_i$ is denominated as *query contextual vector* and is a weighted average of all word vectors from the candidate. The most similar words in the candidate to a word q^t_i have the largest impact in the query contextual vector. Analogously, each candidate token vector x^t_j attends to all query token vectors in order to produce a candidate context vector:

$$\overline{x}^c_j = \sum_{i=1}^{|q^t|} \alpha^{c_j}_i x^q_i, \tag{4.7}$$

where $\alpha^{c_j}_i = exp(s_{ij}) / \sum_{k=1}^{|q^t|} exp(s_{kj})$.

Finally, inspired by Wang and Jiang [109], each token vector of the query and candidate reports is compared with its corresponding context vector by means of the comparison function defined in Equation 4.2. Then, a fully-connected layer with a residual connection receives the resulting comparison vector and modifies the word vectors as follows:

$$m^q_i = x^q_i + \text{ReLU}(W^m \text{cmp}(\overline{x}^q_i, x^q_i) + b^m), \tag{4.8}$$

$$m^c_j = x^c_j + \text{ReLU}(W^m \text{cmp}(\overline{x}^c_j, x^c_j) + b^m), \tag{4.9}$$

where $\text{cmp}(\cdot,\cdot)$ is defined in Equation 4.2, $W^m \in \mathbb{R}^{d^x \times 2d^x}$ is the layer weight matrix, and $b^m \in \mathbb{R}^{d^x}$ is the layer bias.

Textual Encoder Layer

This layer takes the variable-size representation of a report text (query or candidate) and produces a fixed-size representation. This operation is independently performed for the query and candidate reports.

Considering a query q, a bi-directional long short-term memory (bi-LSTM) processes the soft alignment comparison output m^q :

$$\overrightarrow{o}^q_i = \overrightarrow{\text{LSTM}}(m^q_i, \overrightarrow{o}^q_{i-1}) \tag{4.10}$$

$$\overleftarrow{o}^q_i = \overleftarrow{\text{LSTM}}(m^q_i, \overleftarrow{o}^q_{i+1}) \tag{4.11}$$

for $i = 1, 2, \ldots, |q^t|$. The vectors $\overrightarrow{o}_i^q \in \mathbb{R}^{d^o}$ and $\overleftarrow{o}_i^q \in \mathbb{R}^{d^o}$ are concatenated into $o_i^q \in \mathbb{R}^{2d^o}$, where d^o is a hyperparameter indicating the hidden size of the forward $\overrightarrow{LSTM}$ and backward $\overleftarrow{LSTM}$. The intuition is that the bi-LSTM enriches the previous representation with contextual information and allows to capture long dependencies between the words. For the sake of brevity, we omit the technical details of bi-LSTM since it is a standard neural building block. For a detailed explanation of the model, we refer the reader to Goodfellow et al. [33].

Finally, the fixed-representation of the query text is generated as follows:

$$p^q = \text{Pooling}(o_1^q, o_2^q, \ldots, o_{|q^t|}^q), \tag{4.12}$$

where $p^q \in \mathbb{R}^{4d^o}$, and Pooling is a function that concatenates the results of the mean and max pooling operators. The first operator calculates the average vector of the sequence o^q while the second performs the max operation through each dimension of the bi-LSTM output. Similarly, the textual encoder layer generates the fixed representation of the candidate text, denoted as p^c. As depicted in Figure 4.1, query and candidate textual encoder layers share their parameters.

Textual Comparison Layer

This layer compares the textual representations of both reports. As the categorical comparison layer, the $\text{cmp}(\cdot, \cdot)$ function (Equation 4.2) is used to generate a comparative representation. In the sequel, a fully connected layer generates the actual textual comparison:

$$h^{txt} = \text{ReLU}(W^u[p^q; p^c; \text{cmp}(p^q, p^c)] + b^u), \tag{4.13}$$

where $W^u \in \mathbb{R}^{d^u \times 16d^o}$ is the FC weight matrix, $b^u \in \mathbb{R}^{d^u}$ is the FC bias vector, and d^u is a hyperparameter.

4.3.3 Classifier

The SABD output layer comprises two sub-layers: a fully-connected layer and a classification layer. The input of the FC layer is the concatenation of two vectors: the categorical comparison output h^{cat} and the query representation h^{txt}. The classification layer is a standard logistic regression. Thus, the output layer is given by:

$$P(y|c, q) = \text{sigmoid}(W^s \text{ReLU}(W^r x + b^r) + b^s), \tag{4.14}$$

where $x = [h^{cat}; h^{txt}] \in \mathbb{R}^{(d^h+d^u)}$ is the input described above; $W^r \in \mathbb{R}^{d^r \times (d^h+d^u)}$, $b^r \in \mathbb{R}^{d^r}$, $W^s \in \mathbb{R}^{1 \times d^r}$, $b^s \in \mathbb{R}$ are parameters; and d^r is a hyperparameter.

4.4 Experimental Setup

In this section, we describe the main steps of our experimental setup: the evaluation methodology, training procedure, used datasets, and competing methods. The data used in this work and the developed code are freely available [2].

4.4.1 Evaluation Methodology

Towards a more realistic evaluation setup than those used by previous deep learning methods, we evaluate our models using a ranking-based methodology similar to Sun et al. [14]. First, we sort the bug reports in a BTS by their creation date. Then, the reports are chronologically read and inserted in the training set until a specific date t. All the subsequent reports are used to create the test set. Finally, we group the reports in the training set that describe the same bug into buckets. In each bucket, the first submitted report is considered the master report and the remaining ones are the duplicate reports.

In Table 4.1, we exemplify a BTS with five bug reports. Considering that t is 21/12/2018, we generate a training set composed of R1, R2, and R3 and a test set consisting of R4 and R5. The training set thus comprises two buckets: $B_1 = \{\text{R1}, \text{R3}\}$ and $B_2 = \{\text{R2}\}$. During evaluation, we chronologically pick each report r in the test set. When r is a duplicate bug report, we generate a ranked list of the buckets in the system. In this work, the score of a bucket B_i is the highest score yielded by a method when it compares r with each report in B_i. After checking whether r is duplicate, we consider it as submitted and insert r into its correct bucket. Following this procedure, for example, we first pick the report R4 in the scenario of Table 4.1. A ranked list is not produced because R4 is not duplicate and a new bucket $B_3 = \{\text{R4}\}$ is created. After that, the next report R5 is selected. Since it is duplicate, we generate a ranked list composed of $B1$, $B2$, and $B3$. Then, R5 is inserted in $B1$.

Regarding the evaluation methodology used by other ranking approach methods, Budhiraja et al. [89] do not describe how the test dataset was generated nor the procedure to create the ranked list. Both are crucial elements of the evaluation methodology and can considerably impact the achieved performances. Deshmukh et al. [68] extract pairs of bug reports from a BTS and *randomly* split them into training and test datasets. In their evaluation, for each duplicate bug report, their method outputs a recommendation list composed of only reports

[2] `https://github.com/irving-muller/soft_alignment_model_bug_deduplication`

Table 4.1 Fictional BTS to exemplify the evaluation methodology.

Bug report ID	**Creation Date**	**Master Report**
R1	01/12/2018	-
R2	12/12/2018	-
R3	20/12/2018	R1
R4	30/12/2018	-
R5	31/12/2018	R1

within the test set. This artificially reduces the number of reports that must be searched for each queried report, which makes the problem much easier [15]. Furthermore, in the BTSs, we only have access to data that was reported before a current time x. Thus, the model can only be trained using data from this period. After training a model, it only examines bug reports that were created after x. Randomly shuffling the data allows the model to be trained with reports created from the future (after x) and to retrieve candidates that were submitted after the queried report. Additionally, this randomization makes the problem easier because it spreads more uniformly the features through the dataset and can mitigate the concept drift issue. Therefore, we believe our experimental setting is more realistic. We compare our methods with those proposed in Budhiraja et al. [89] and Deshmukh et al. [68]. However, due to the methodological differences aforementioned, and since source code was not provided by authors, we implemented those methods to the best of our knowledge, as described in Section 4.4.5.

Like Sun et al. [14], we evaluate a method using two metrics: mean average precision (MAP) and recall rate@k (RR@k). Both metrics are based on the ranking of reports according to the scores computed by a method. MAP is a general ranking metric. In our setting, rankings only need to contain one relevant item per query to be considered a hit. Thus, MAP can be simplified as:

$$MAP = \frac{1}{Q}\sum_{i=1}^{Q}\frac{1}{p_i}, \tag{4.15}$$

where Q is the number of duplicate bug reports in the evaluation set and p_i is the position of the correct bucket in the ranked list. RR@k is equal to the ratio of duplicate reports whose correct buckets are within the top-k buckets in the given ranking to the number of duplicate bug reports. RR@k is defined as:

$$\text{RR@}k = \frac{n_k}{Q}, \tag{4.16}$$

where n_k is the number of query reports in the test set for which the corresponding bucket appears in the top-k positions of the ranking computed by a method.

4.4.2 Datasets

We use parts of the datasets published by Lazar et al. [110] in our experiments[3]. They retrieved and curated reports submitted until 2014 from four BTSs: OpenOffice, Eclipse, NetBeans and Mozilla. OpenOffice contains a set of open-source tools that aim to help the office activities. NetBeans and Eclipse are popular open-source integrated development environments (IDEs) that support many different languages. The Mozilla BTS manages bugs of several open source projects, such as Thunderbird (email client) and Firefox (Web browser).

Sun et al. [14] assess their methods using small portions of the aforementioned datasets. More specifically, they use reports within a three-year period for the OpenOffice dataset and within a one-year period for the other three datasets. This choice ignores reports submitted before this period, which overestimates their method [111]. For each BTS, we use the reports employed by Sun et al. [14] as our test datasets[4]. The reports submitted before these periods are split into training and validation datasets. Validation sets comprise the latest 5% reports and the remaining earlier reports comprise training sets. Statistics of these datasets are presented in Table 4.2.

Table 4.2 Statistics of datasets. *Period* column indicates the period comprising each dataset as a whole (Train+Val+Test). *Start Date* column indicates the first day included in test datasets.

Dataset	**Period**	**Training**		**Validation**		**Test**			**Total**
		Duplicate	**All**	**Duplicate**	**All**	**Start Date**	**Duplicate**	**All**	
Eclipse	10/10/01 - 31/12/08	27,481	198,183	1,446	14,703	01/01/08	4,380	45,794	258,680
Mozilla	07/04/98 - 31/12/10	122,199	438,806	6,431	44,014	01/01/10	9,701	65,940	548,760
OpenOffice	16/10/00 - 31/12/10	13,570	80,786	714	4,109	01/01/08	4,664	31,333	116,228
Netbeans	21/08/98 - 31/12/08	16,639	116,351	875	5,548	01/01/08	5,009	31,667	153,566

4.4.3 Time Window

As the number of reports submitted increases over time, it becomes computationally expensive to detect duplicate bug reports in BTSs since each new report has to be compared with all the reports submitted before it. This growth indeed degrades the performance of the method, which can negatively affect the triage process [15]. A simple solution for this problem, called time window or time frame, consists of searching for reports that were submitted within a specific range of days before the new report. In this study, a bucket is considered to be a candidate when at least one of its reports is within the defined time window.

[3]`http://alazar.people.ysu.edu/msr14data/`

[4]We decided to use the same period of Eclipse for Netbeans since [14] did not evaluate their method on this BTS.

Table 4.3 Percentage of duplicate bug reports in the test set that reaches the correct bucket in the time window of one year and three years.

Dataset	**1 year**	**3 years**
Eclipse	88.53%	97.48%
OpenOffice	75.27%	90.97%
Netbeans	93.51%	98.88%
Mozilla	88.45%	96.73%

Table 4.3 shows the fraction of duplicate bug reports in the test sets for which one of the reports in their associated buckets can be reached within time windows of one and three years. We consider that three years is a reasonable time frame to be used in real environments, especially for popular software BTSs that daily receive many bug reports, e.g., Eclipse BTS receives on average around 99 reports per day [112]. Except for OpenOffice, the use of the time window significantly reduces the computation demand at the cost of a small negative impact on the performance upper bound – less than 3.4% of duplicate reports will not have their bucket in the ranked list. We also test the methods with a time window of one year to measure how its size affects performance.

4.4.4 Training

We preprocess the textual data by replacing all the non-alphanumeric characters with spaces [14]. After that, the text is converted to lower-case and tokenized on white space characters. This preprocessing separates tokens concatenated by punctuation, e.g., module paths, file paths, and function calls. Our intuition is that package, file, and class names are relevant for this task. Analyzing a small sample of long reports, we observed that many of them append lengthy stack traces and log files to their description. Thus, we limit the text length to 350 tokens in order to clean less important elements without missing much relevant information. Although we acknowledge that this value can be suboptimal, it appears to be sufficient to achieve reasonable results. Categorical features comprise the following fields: component, product, severity and priority.

Following Deshmukh et al. [68], we initialize the word embedding using an instance of pre-trained vectors[5]. Words that appear in the training dataset but not in that instance are pre-trained using *GloVe* [113] and textual data from the reports in the training dataset. To avoid overfitting, the word embeddings are not fine-tuned.

SABD is a binary classifier that takes two reports (query q and candidate c) and outputs the

[5]`http://nlp.stanford.edu/data/glove.42B.300d.zip`

probability $P(y|q,c)$ of report q being a duplicate of report c. Thus, it is trained over a set of pairs of reports along with their labels in order to minimize the cross entropy loss function:

$$J(\theta) = -\frac{1}{|S|} \sum_{(q,c,y) \in S} y \log P(y|q,c) + (1-y) \log(1 - P(y|q,c)), \tag{4.17}$$

where $S = \{(q,c,y)\}$ is the training set composed of pairs of reports (q,c) along with their labels y ($y = 1$ when q and c are duplicates, otherwise $y = 0$). We optimize SABD for 12 epochs using ADAM optimizer [114] with a learning rate of 0.001 and a batch size of 256.

Building S is challenging since it is used to train a binary classifier to perform a ranking task such that, at test time, there are more negative examples than positive ones. In order to describe how S is built, let us split it into two sets $S = S^{+} \bigcup S^{-}$ such that S^{+} is the set of positive examples, i.e, those whose $y = 1$; and S^{-} is the set of negative examples. S^{+} comprises all pairs of duplicate reports in the set of training reports. On the other hand, S^{-} is generated before the start of each training epoch by sampling non-duplicate pairs until $\frac{|S_{+}|}{|S_{-}|} = rt$, where rt is the rate between the pairs of duplicate reports by the non-duplicate ones. Moreover, a negative pair is only included in S^{-} if $-\log(P(y = 0))$ is larger than a given threshold λ, i.e., if the example is not too easy for the current classifier. This sampling is inspired by [115, 18, 116, 117] and speeds up training by providing more informative examples. SABD has achieved optimum results for $rt = 1$ and $\lambda = 10^{-3}$. The hyperparemeter values were tuned using the validation set and their values are presented as follows: $d^{cat} = 20$, $d^{a} = 40$, $d^{h} = 40$, $d^{w} = 300$, $d^{f} = 5$, $d^{o} = 150$, $d^{u} = 600$, and $d^{r} = 300$.

4.4.5 Competing Methods

We compare SABD with five other methods from the literature. The $\text{BM25F}_{\text{ext}}$ and REP are ranking-approach methods that were proposed by Sun et al. [14]. These are popular methods and their implementations are available.[6] Besides, we compare SABD with the following deep learning methods: DWEN [89], Siamese Pair [68] and Siamese Triplet [68]. Although these works have used ranking-based evaluation methodologies, as described in Section 4.4.1, such methodologies present relevant issues. Moreover, since their implementations are not available, we implemented them to the best of our knowledge. We found that the following minor modifications in the method architecture or training have improved their performance in the validation dataset: 1) the feed-forward neural network of DWEN receives categorical features generated in a similar way to Siamese Pair [14]; 2) bi-LSTMs followed by average and max poolings are used to encode the summary and description in Siamese Pair and Siamese

[6]`https://chengniansun.bitbucket.io/projects/bug-report/fast-dbrd.tgz`

Triplet; 3) the last sub-network of Siamese Pair receives, in addition to the original inputs, the squared difference and element-wise multiplication of the final report representations; and 4) we train these methods using the procedure described in Section 4.4.4 to generate negative examples.

Models evaluated by means of decision-making methodology cannot be fairly compared to those that employ ranking-based approaches, since the underlying problem differs. However, SABD is indirectly compared with Xie et al. [90], as this model is very similar to the Siamese Pair baseline. Both models independently generate the fixed-representation of report pairs using standard neural network blocks (e.g., CNN and LSTM) and exploit categorical data. Poddar et al. [19] propose a technique to simultaneously learn latent topics from reports and train a classifier for bug deduplication. This technique could be adapted to SABD with some minor changes due to its generality.

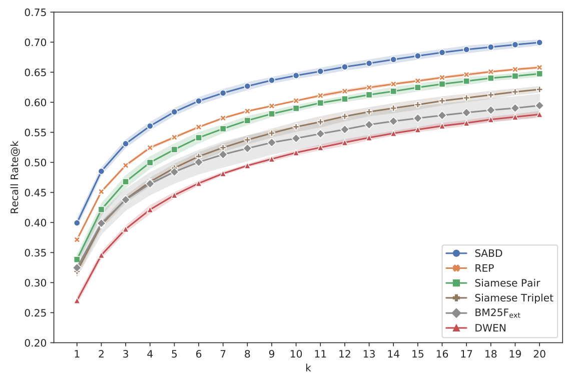
4.5 Experimental Results

Since the competing methods and SABD are stochatics, we perform *five distinct runs* for each experimental configuration[7]. We report in this section average performance in terms of MAP and RR@k, as well as standard deviations illustrated as *error bands* in figures and *inside brackets* in tables. Following Sun et al. [14], the RR@k is calculated for each $k = 1, 2, \ldots, 20$. It is important to notice that, when evaluating a model, we include duplicate reports whose buckets are not within the considered time window. These duplicate reports are considered misses for RR@k computation, and their terms $\frac{1}{p_i}$ in the MAP expression are 0.

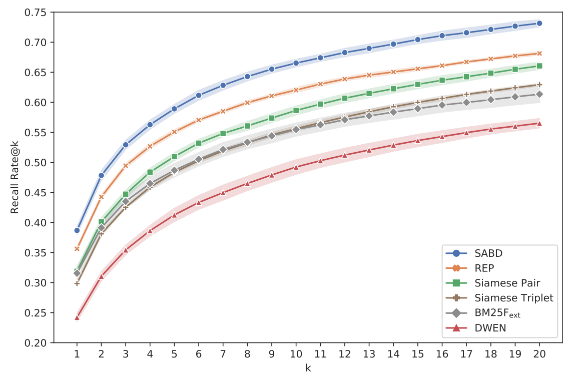
4.5.1 Main Analysis

In the right column of Figure 4.2, we depict RR@k of all methods in the four datasets using a time window of three years. In all datasets, SABD constantly achieves the best RR@k among the compared methods. It outperforms the second best method by 3.06%–5.01% in Eclipse, 3.34%–6.35% in OpenOffice, 2.66%–6.64% in Netbeans, and 4.17%–5.19% in Mozilla. Table 4.4 reports the method results on the MAP metric in each test set using time window of one and three years. Considering the time window of three years, SABD also achieves higher MAP values than all methods in all datasets. The improvement of SABD over the second best method on the MAP metric is 3.5%, 4.3%, 3.9%, and 4.5% in Eclipse, OpenOffice, Netbeans, and Mozilla, respectively.

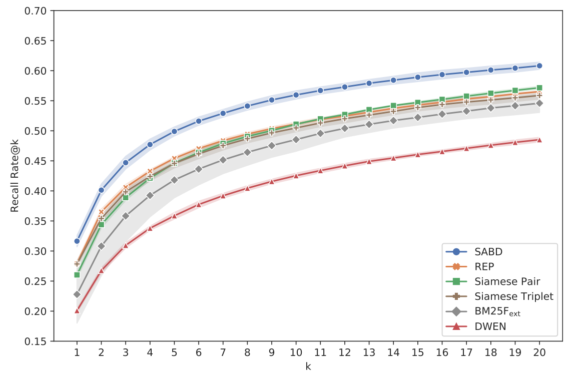
[7]BM25F$_{\text{ext}}$ and REP are run 10 times in NetBeans since they generated large standard deviation.



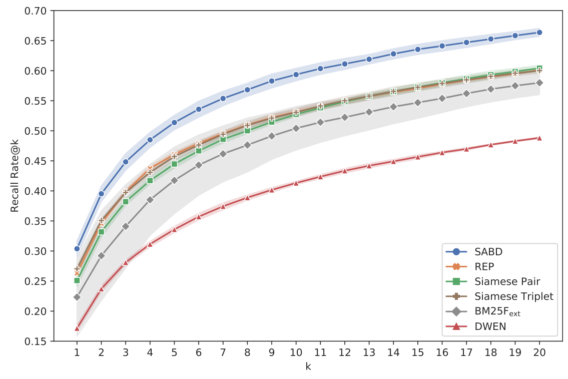
(a) Eclipse - 1 year



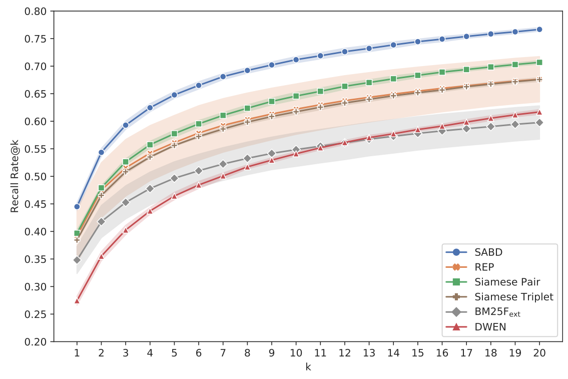
(b) Eclipse - 3 years



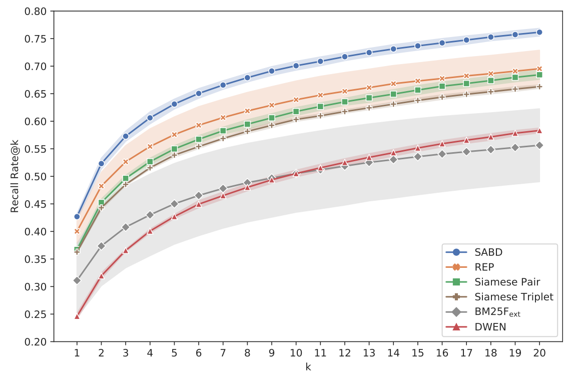
(c) OpenOffice - 1 year



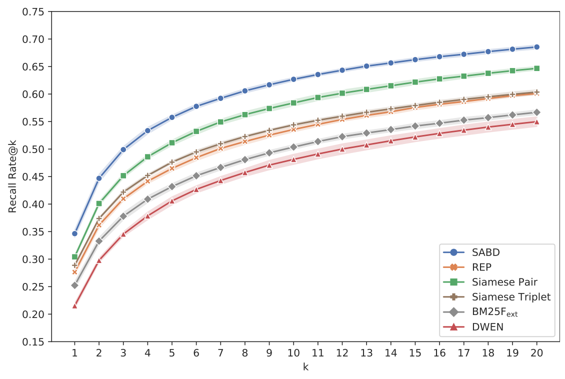
(d) OpenOffice - 3 years



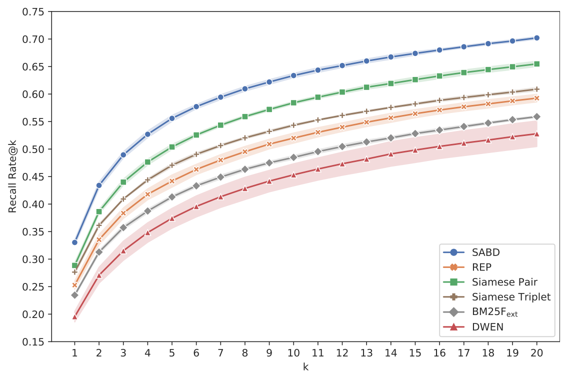
(e) Netbeans - 1 year



(f) Netbeans - 3 years



(g) Mozilla - 1 year



(h) Mozilla - 3 years

Figure 4.2 Recall Rate@k in test sets of Eclipse, OpenOffice, Netbeans and Mozilla.

Table 4.4 MAP in test sets.

Method	**Eclipse**		**OpenOffice**		**Netbeans**		**Mozilla**	
	1 year	**3 years**	**1 year**	**3 years**	**1 year**	**3 years**	**1 year**	**3 years**
DWEN	0.353[0.004]	0.325[0.007]	0.276[0.003]	0.252[0.004]	0.365[0.006]	0.333[0.004]	0.305[0.004]	0.282[0.012]
$\text{BM25F}_{\text{ext}}$	0.402[0.016]	0.398[0.010]	0.315[0.037]	0.313[0.054]	0.417[0.027]	0.378[0.066]	0.338[0.005]	0.320[0.002]
Siamese Triplet	0.401[0.005]	0.387[0.002]	0.356[0.005]	0.358[0.004]	0.466[0.001]	0.446[0.002]	0.376[0.002]	0.367[0.003]
Siamese Pair	0.425[0.006]	0.410[0.005]	0.346[0.003]	0.343[0.007]	0.482[0.004]	0.454[0.004]	0.401[0.003]	0.390[0.004]
REP	0.452[0.001]	0.447[0.003]	0.361[0.003]	0.355[0.003]	0.472[0.045]	0.483[0.024]	0.365[0.004]	0.343[0.007]
SABD	0.484[0.004]	0.482[0.006]	0.400[0.008]	0.401[0.011]	0.538[0.006]	0.522[0.006]	0.443[0.005]	0.435[0.005]

Deshmukh et al. [68] compared the Siamese Triplet with Siamese Pair using only accuracy and, according to them, the former significantly outperforms the latter. However, as mentioned before, the accuracy is less adherent than RR@k and MAP for real environments. We found that, in fact, Siamese Pair achieves significantly better RR@k and MAP values than Siamese Triplet in three of four test sets. Moreover, our results show that DWEN achieves poor MAP and RR@k values on the four datasets and it is significantly outperformed by all methods in Eclipse, Netbeans, and Mozilla repositories.

Considering only the deep learning models and the time window of three years, the improvement of SABD over the second best neural network on the RR@k metric is 6.78%–8.21%, 5.29%–6.91%, 5.97%–8.49% and 4.17%–5.19% in Eclipse, OpenOffice, Netbeans and Mozilla, respectively. In terms of MAP, SABD surpasses the second best neural network by 7.2% in Eclipse, 4.3% in OpenOffice, 6.8% in Netbeans and 4.5% in Mozilla. To the best our knowledge, we are the first to compare different neural networks in the bug deduplication using the ranking methodology. Amongst all studies in the literature, Poddar et al. [19] was the first to compare different deep learning methods for this task, although they evaluate them using the decision-making methodology.

Regarding the methods proposed by Sun et al. [14], the first relevant point is that $\text{BM25F}_{\text{ext}}$ achieves a similar curve regarding RR@k and a slightly better MAP value than the Siamese Triplet in Eclipse. Moreover, REP outperforms the two siamese neural networks in Eclipse and Netbeans, and it has comparable results to Siamese Triplet and Siamese Pair in OpenOffice. Even though $\text{BM25F}_{\text{ext}}$ and REP are simpler methods than the siamese neural networks that contain thousands of parameters, they are able to perform similarly or better than these deep learning models. Finally, it is important to point that REP and $\text{BM25F}_{\text{ext}}$ have a large standard deviation in OpenOffice and, exclusively $\text{BM25F}_{\text{ext}}$, in Netbeans.

Time Window Analysis

In the left column of Figure 4.2, we present RR@k for all the considered methods, for $k = 1, 2, \dots, 20$, on the four test sets using a time window of one year. The MAP results of these methods in the same experimental setups are reported in Table 4.4. Despite some minor differences, the findings using a window of one year are similar to the ones with a longer frame of three years – including the fact that SABD constantly outperforms the methods in terms of MAP and RR@k in all datasets.

Extending the window span from one to three years decreases the number of duplicate bug reports whose ranked list never contains the correct master reports. However, we found that this does not necessarily correspond to performance improvements in terms of RR@k. For instance, in Figure 4.3, we compare the curve of RR@k achieved by SABD in each dataset and time window. Increasing the time frame positively impacts, in general, the SABD performance in OpenOffice and Eclipse, while it marginally reduces its performance in Netbeans and, partially, in Mozilla. We believe that this occurs due to the trade-off between two factors related to the time window length. Expanding the time window raises the upper bound of the RR@k. However, at the same time, it increases the quantity of reports that are searched, making the bug deduplication more difficult [15]. Finally, as shown in Table 4.4, increasing the time window from one to three years reduces the performance of the methods in terms of MAP. This indicates that MAP is more sensitive to the quantity of reports that must be searched than RR@k.

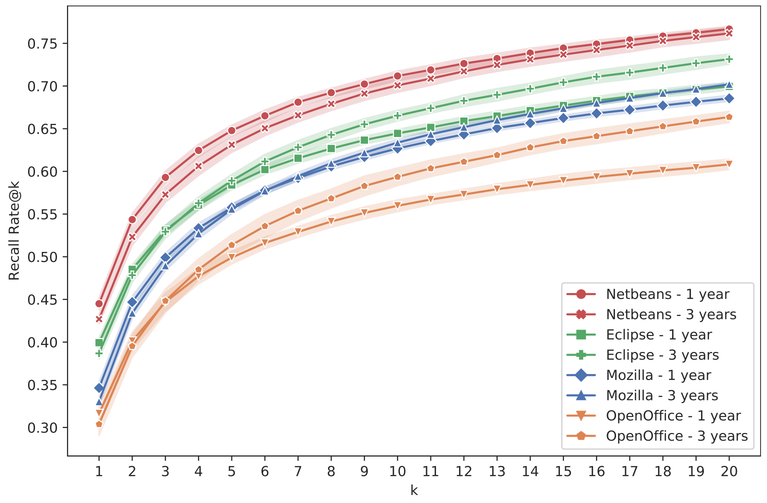


Figure 4.3 Comparison of SABD performance in terms of RR@k.

4.5.2 Ablation Study

In this section, we perform an ablation study to evaluate the effectiveness of different components of SABD. Ablation study consist in removing a single component from the original architecture, and measuring how much this isolated modification impacts the model performance. The more a component affects the performance, the more effective it is considered.

We test two distinct configurations related to the soft alignment comparison layer. Setup (1) measures the impact of the data exchange by removing the soft alignment comparison layer, thus independently generating the report representations as Sun et al. [14], Xie et al. [90], and Budhiraja et al. [89]. Although this setup may show the layer importance, it is not clear which part of the layer is the most significant. Thus, one also needs to evaluate the importance of SABD capacity to dynamically focus on different parts of a report.

If the model is able to compress the report into a fixed-length vector without losing any relevant information, then SABD will achieve similar results because the FC layer in the soft alignment comparison layer produces similar outputs. However, if SABD is negatively impacted, the summarization of a report into a fixed-length representation is the bottleneck that needs to be replaced by a more powerful mechanism such as the soft-attention alignment.

Setup (2) studies the need for the soft-attention alignment by replacing it with a mechanism similar to that of Poddar et al. [19], which is more powerful than a simple mean-pooling since the fixed-length representation of a report depends on the other report. In (2), the context vector of the query $\overline{x}_i^q$ is the attention mechanism result given by Equation 4.6 in which the k-th attention coefficient is proportional to the scaled dot-product:

$$\alpha_k^{q_i} \propto \frac{x_k^c \cdot \text{Mean}[x_1^q, \ldots, x_{|q^t|}^q]}{\sqrt{d^x}}, \tag{4.18}$$

where x_k^c is the k-th word vector of the candidate and Mean[...] is the result of the mean-pooling operator over the word vectors in the query. The candidate context vectors $\overline{x}_j^c$ are produced likewise.

Furthermore, we test four additional setups. In (3), the categorical module is removed from SABD, i.e, only textual data is used for detecting duplicate reports. In (4), we remove the fully-connected layer (Equation 4.4) that modifies the concatenation of the word and field embedding vectors (v_i^q and v_j^c). In (5), we remove the bi-LSTM in the textual encoder layer, i.e., the mean and max-pooling generate the fixed-length representation of the reports. In (6), field embedding is not concatenated with word embeddings and, thus, the words from the summary and description are identically represented.

Figure 4.4 and Table 4.5 report the RR@k and MAP achieved by the seven configurations in the validation dataset of Eclipse. As shown in Figure 4.4 and Table 4.5, the soft-alignment comparison is the most crucial component of our model since removing this layer from SABD significantly degrades its performance. Besides, the setup (1) is marginally outperformed by setup (2). Both results corroborate the hypothesis that data exchange improves the representations. We also observe that the model performance substantially decreases when the soft-attention alignment is replaced by a less powerful mechanism. This confirms our assumption that summarizing report information into fixed-length representation is the bottleneck of the Poddar et al. [19] model. An architecture that dynamically focuses on distinct information from a report is less prone to lose information and, therefore, performs a better report comparison. We also observe that removing the FC sublayer from the textual embedding layer decreases SABD performance. This result confirms our hypothesis about the importance of projecting the words into a dimension space that better captures word relevance for the bug deduplication. As expected, SABD performs significantly worse when categorical data is not used. This data provides additional information about the report which can help the bug deduplication (e.g., the probability of two reports being duplicate from two different software components is usually low). Further, removing the bi-LSTM considerably decreases the model performance which demonstrates our hypothesis that contextual information about the words is useful for deduplication.

Finally, we find that removing the field embedding does not significantly affect the SABD performance. This is an unexpected result since words from different fields were supposed to have distinct relevance.

Table 4.5 Ablation study in terms of MAP.

Method	**MAP**	**Diff.**
SABD	0.500[0.005]	-0.000
(6) Remove Field Embedding	0.505[0.002]	+0.005
(5) Remove bi-LSTM	0.468[0.005]	-0.032
(4) Remove FC in Textual Embedding	0.465[0.008]	-0.035
(3) Remove Categorical Encoder	0.467[0.008]	-0.033
(2) $\overline{x}_i^q$ and $\overline{x}_j^c$ produced by [19]	0.440[0.009]	-0.060
(1) Remove Soft-alignment Comparison	0.424[0.006]	-0.076

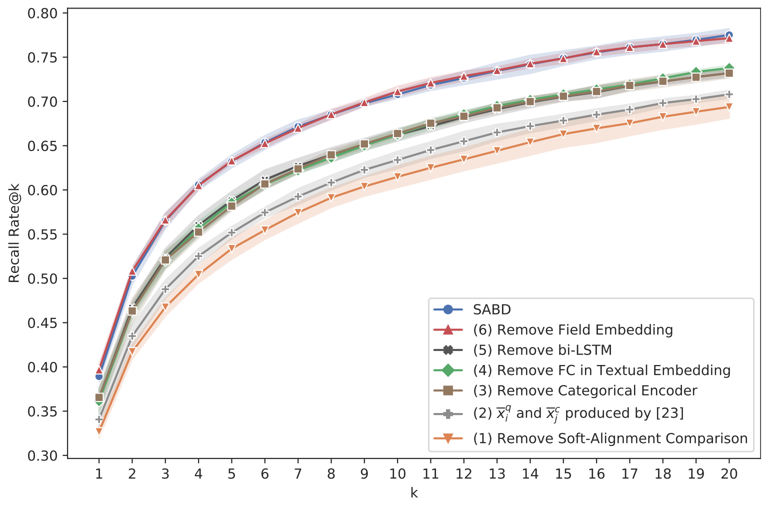


Figure 4.4 Ablation study in terms of RR@k.

4.6 Concluding Remarks

We proposed SABD, a novel soft alignment method for bug deduplication. In contrast of Siamese neural networks, SABD exchanges data between the reports before generating their fixed-length representations. The mechanism responsible for this data interchange is more powerful than the one proposed in Poddar et al. [19] because it can dynamically focus on distinct information of a report during the feature extraction of the other report. We experimentally evaluate SABD and competing methods (including two non-deep learning ones) using a methodology based on Sun et al. [14]. This methodology is more adherent to real environments than the ones often used in the literature [19, 68, 89, 90]. SABD significantly outperforms the other methods in all experimental setups.

It is important to notice that, even though competing deep learning methods were implemented to the best of our knowledge, it is not possible to guarantee that those are identical to the ones used in the studies. As shown in the ablation study, the soft-aliment positively impacts the model performance. However, this mechanism has a cost in terms of runtime. As the fixed-length representations of the reports are jointly generated, it is not possible to save computation time by storing them. Therefore, our model is slower than the methods based on siamese neural networks.

CHAPTER 5 ARTICLE 2: TRACESIM: AN ALIGNMENT METHOD FOR COMPUTING STACK TRACE SIMILARITY

Authors. Irving Muller Rodrigues, Aleksandr Khvorov, Daniel Aloise, Roman Vasiliev, Dmitrij Koznov, Eraldo Rezende Fernandes, George Chernishev, Dmitry Luciv, and Nikita Povarov. Published at the Empirical Software Engineering journal, 2022[1].

Abstract. Software systems can automatically submit crash reports to a repository for investigation when program failures occur. A significant portion of these crash reports are duplicate, i.e., they are caused by the same software issue. Therefore, if the volume of submitted reports is very large, automatic grouping of duplicate crash reports can significantly ease and speed up analysis of software failures. This task is known as crash report deduplication. Given a huge volume of incoming reports, increasing quality of deduplication is an important task. The majority of studies address it via information retrieval or sequence matching methods based on the similarity of stack traces from two crash reports. While information retrieval methods disregard the position of a frame in a stack trace, the existing works based on sequence matching algorithms do not fully consider subroutine global frequency and unmatched frames. Besides, due to data distribution differences among software projects, parameters that are learned using machine learning algorithms are necessary to provide more flexibility to the methods.

In this paper, we propose TraceSim – an approach for crash report deduplication which combines TF-IDF, optimum global alignment, and machine learning (ML) in a novel way. Moreover, we propose a new evaluation methodology for this task that is more comprehensive and robust than previously used evaluation approaches. TraceSim significantly outperforms seven baselines and state-of-the-art methods in the majority of the scenarios. It is the only approach that achieves competitive results on all datasets regarding all considered metrics. Moreover, we conduct an extensive ablation study that demonstrates the importance of each TraceSim's element to its final performance and robustness. Finally, we provide the source code for all considered methods and evaluation methodology as well as the created datasets.

Keywords. Duplicate Crash Report, Crash Report Deduplication, Duplicate Crash Report Detection, Automatic Crash Reporting, Stack Trace

[1]Available at [118]

5.1 Introduction

Many software products are nowadays equipped with automated crash reporting systems such as Apport[2], Mozilla Socorro[3], and CrashPad[4]. These systems detect software crashes, collect data related to user environment, system state and execution information [7], and group such data into a so-called *crash report*. While automated crash reporting systems reduce the dependence on users to collect relevant information about failures, they drastically increase the number of crash reports. For instance, according to Campbell et al. [10], Mozilla Firefox received around 2.2 million crash reports in the first week of 2016. A significant portion of these crash reports were duplicates, i.e., multiple reports related to the same software bug. For example, we found that 72% of the reports of the IntelliJ Platform (a JetBrains product family) were duplicates.

In software projects, duplicate crash reports are grouped into clusters called *buckets*. This grouping helps to prioritize the bug fixing, provides supplemental information about a failure, and reduces the effort required to fix a bug [12, 13]. However, due to the massive volume of crash reports submitted daily, it is unfeasible to manually allocate new reports to buckets. For instance, considering that 13,000 crash reports were submitted per hour for Mozilla Firefox [10] and, that a "superhuman" could review one report per second, a triager would take around 3.6 hours to identify the buckets of these new reports. Hence, it is vital for large software projects to automatically assign crash reports to buckets. This task is known as *duplicate crash report detection*, *crash report bucketing* or *crash report deduplication* [10, 22].

During the program lifetime, a stack (named *call stack*) keeps track of active subroutines. We consider a subroutine active if it is under execution or waiting for the completion of other subroutines. Call stacks are composed of frames: data structures that store information on a single active subroutine (such as its return address and arguments). These frames are stored following the LIFO (last in, first out) principle, i.e., the frames related to the last executed subroutines are on the top of the stack. The *stack trace* is hence a snapshot of the call stack in memory which is captured and presented to the user when a system crashes.

In Figure 5.1, we illustrate a crash report. This example presents the details about the system and environment in **lines 1–7**. This information is variable and depends on the application. Moreover, reports may include user descriptions of bugs and how they could be reproduced. In Figure 5.1, **lines 9–43** represent a stack trace. It encompasses valuable information for developers to understand and fix an error [8].

[2]`https://wiki.ubuntu.com/Apport`
[3]`https://crash-stats.mozilla.com/`
[4]`https://goto.google.com/crash/root`

```
Date: 2016-01-20T22:11:40.034Z
Product: XXXXXXXXXXXX
Version: 144.3143
Action: null
OS: Mac OS X
Java: Oracle Corporation 1.8.0_40-release
Message: new child is an ancestor

java.lang.IllegalArgumentException: new child is an ancestor
  at javax.swing.tree.DefaultMutableTreeNode.insert(DefaultMutableTreeNode.java:179)
  at javax.swing.tree.DefaultMutableTreeNode.add(DefaultMutableTreeNode.java:411)
  at com.openapi.application.impl.ApplicationImpl$8.run(ApplicationImpl.java:374)
.....
  at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617)
  at java.lang.Thread.run(Thread.java:745)
  at org.ide.PooledThreadExecutor$2$1.run ....
```

Figure 5.1 Crash report example

The majority of previous studies addresses crash report deduplication mainly by measuring similarity between the stack traces of two crash reports. Lerch and Mezini [93], Campbell et al. [10], and Sabor et al. [94] use traditional information retrieval techniques to compute this similarity. These works propose to encode stack traces as vectors whose dimensions are related to subroutine names (fully-qualified names of functions) and values are calculated using *Term Frequency – Inverse Document Frequency* (TF-IDF) [23]. One key drawback of information retrieval methods is that they ignore the order of the subroutines in a stack trace. Other studies [21, 22, 95, 96] explicitly consider the sequence of function calls within stack traces and employ variants of sequence matching algorithms (such as edit distance, longest common subsequence and optimal global alignment) to measure the similarity between two stack traces.

Brodie et al. [21] were the first to use a sequence matching algorithm to compute the similarity between two stack traces. Moreover, in the matching algorithm, they considered two pieces of information to weight the importance of a subroutine in a given stack trace: its position in the stack trace and its frequency in a large database. The rationale behind this is twofold: (i) bugs are more likely to be related to subroutines in the top positions of the stack trace [8]; and (ii) rare subroutines are more relevant than frequent ones, similarly to TF-IDF. Later, Dang et al. [22] proposed PDM, a method that also considers subroutine position when comparing two stack traces by means of a sequence matching algorithm. Different from Brodie et al. [21], PDM includes a machine learning (ML) algorithm to learn the parameters that control the impact of frame position in the matching algorithm. In that way, PDM can adapt to different data distributions from varied software projects or even to temporal shifts in the same project. One neglected aspect in the current literature is the proper consideration of subroutines that exist in only one of the two stack traces under comparison. We call these subroutines *unmatched* as opposed to matched subroutines that are present in both stack traces. In Brodie et al. [21], for instance, position and frequency are considered only for matched subroutines. For unmatched ones, the similarity score is given by a negative constant value. In Dang et al. [22], unmatched subroutines are ignored when computing the similarity score. However, these missing subroutines may be important to estimate the similarity (or, in this case, the dissimilarity) between stack traces, especially the ones that are rare and lie in the top positions.

In this work, we propose *TraceSim*, the first technique that structurally combines TF-IDF, optimum global alignment, and machine learning for crash report deduplication. To compute the similarity between two stack traces, TraceSim finds the optimum global alignment between them by means of the Needleman-Wunsch (NW) algorithm. Differently from previous approaches, we leverage the flexibility of this global alignment algorithm to consider all subroutines, either matched or unmatched, to compute the similarity score between two stack traces. Moreover, for all subroutines, TraceSim considers both their frequency in a large database (TF-IDF) and their position in the stack traces. Additionally, TraceSim employs a ML algorithm to learn parameters that regulate the influence of all these elements within the NW algorithm. Hence, TraceSim can be viewed as a generalization of the methods proposed by both Brodie et al. [21] and Dang et al. [22].

Inspired by works on bug deduplication, we also propose a new evaluation methodology for crash report deduplication. Our comprehensive methodology considers different aspects of the problem, such as the system capacity to separate non-duplicates from duplicates, the accuracy of report assignment to buckets, and the system performance for ranking. By means of this methodology, we compare TraceSim to state-of-the-art and baseline systems using four datasets from open-source projects (Ubuntu, Eclipse, Netbeans and Gnome) and one industrial dataset (JetBrains). Additionally, a detailed ablation study is performed to assess distinct elements of TraceSim, namely TF-IDF, global aligment and ML.

The main contributions of this paper are summarized as follows:

1. We propose TraceSim, a novel method for crash report deduplication that combines TF-IDF, global alignment, and machine learning. We experimentally demonstrate that each one of these methodological choices significantly contributes to TraceSim's performance and robustness.

2. We report on the most comprehensive experimental evaluation which includes many previous methods in the literature and assesses different aspects of these methods. The experiments are performed on five distinct projects involving distinct programming languages and characteristics. TraceSim significantly outperforms the competitive

methods in the majority of the scenarios, and it is the only method that consistently performs well in all scenarios.

3. We provide our full evaluation framework[5] which comprises: datasets generated from open source projects, implementations of all considered methods, and implementation of the proposed evaluation methodology. The provided framework is crucial for reproducibility, so that future works can easily compare new methods with Tracesim and other relevant methods from the literature within a comprehensive and unified framework.

The remainder of this paper is organized as follows. Section 5.2 describes the proposed method. Section 5.3 presents the existing techniques that address crash report deduplication. Section 5.4 presents the proposed evaluation methodology and the experimental setup. In Section 5.5, we experimentally compare TraceSim to competitive techniques and report on several ablation studies. Section 5.6 discusses the possible threats to the validity of our work. Finally, concluding remarks are given in Section 5.7.

5.2 TraceSim

A common assumption in the literature is that failures caused by the same bug are represented by similar system executions. Since a stack trace can capture the state of a system execution right before a crash, crash deduplication usually proceeds by mainly comparing the similarity between stack traces. In order to compute this similarity, many studies employ sequence matching algorithms so that they can measure the overlapping between two stack traces keeping track of the order of the compared frames.

A classic sequence matching method is the Needleman–Wunsch (NW) algorithm [119], which finds the optimal global alignment between two sequences. A global alignment consists in aligning the elements of two sequences end-to-end. In Figure 5.2, we illustrate a global alignment of two stack traces ($stack_1$ and $stack_2$). Filled rectangles represent frames and empty ones symbolize gaps: special structures that allow to shift the position of an element in the alignment. As can be observed, frames can be lined up to gaps, where each gap represents that a specific subroutine is missing in that position of the stack trace. We call a *match* the alignment of identical frames (e.g., the frames `Maps.difference` in $stack_1$ and $stack_2$). In contrast, we name the alignment of two different frames a *mismatch* (e.g., the frames `ValidatorPage.performOk` and `BuilderPage.schedCleaner`). In this work, a mismatch is considered as equivalent to two gap alignments since it is unlikely that two different subroutines

[5]`https://github.com/irving-muller/TraceSim_EMSE`

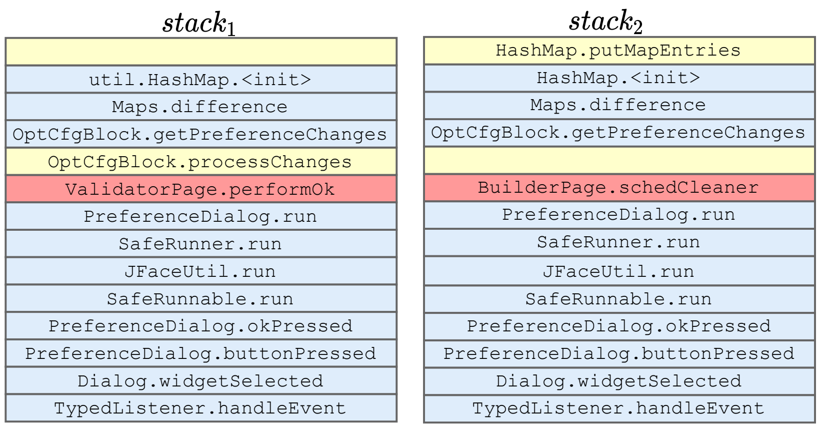


Figure 5.2 Best global alignment example: 12 matches, 1 mismatches, and 2 gaps. Matches, mismatches and gaps are represented by blue, red, and yellow, respectively.

possess interchangeable functionality [21]. For instance, `ValidatorPage.performOk` and `BuilderPage.schedCleaner` are clearly different and, therefore, it is appropriate to align these frames to gaps.

Two stack traces can be aligned in multiple ways. The optimal global alignment problem is formulated as a maximization problem such that each possible frame alignment (match, mismatch, and gap) has an assigned value. Thus, the optimum solution for the problem corresponds to the global alignment for which the sum of the match values minus the sum of the mismatch and gap values is maximum. A common scheme is to define the values for matches, mismatches and gap alignments as constants, such that a specific alignment is always associated to the same value.In order to leverage the peculiar characteristics of stack traces, we employ a scheme that computes the alignment values in a more effective way for the crash deduplication task. In this scheme, weights are assigned to each frame. These weights estimate the frame importance to discriminate two stack traces and are used to compute match, mismatch and gap values. Our hypothesis is that some frames are more relevant than others for comparison and, hence, their correct or wrong alignment should have higher impact on the computed similarity score.

The NW algorithm is more suitable for our proposed scheme than other classic sequence matching algorithms, e.g. longest common subsequence (LCS) and Levenshtein distance (also known as edit distance). In the LCS, the similarity is only affected by the matches since mismatches are not allowed and gap values are zero. Hence, the incorrect alignment of important frames is not considered for measuring the similarity. Conversely, the matches of relevant and irrelevant frames are not distinguishable in the edit distance because the match values are always zero. The NW algorithm is the method that allows us to fully consider the weights of all frames to measure the similarity of stack traces.

In the remainder of this section, we describe *TraceSim*, our proposed method for crash report deduplication. This method computes the similarity of two stack traces s^q and s^c from a new query report q and a candidate report c, respectively. Each stack trace is represented as a sequence of frames, i.e., $s^q = (s^q_1, s^q_2, \ldots, s^q_{|s^q|})1$ and $s^c = (s^c_1, s^c_2, \ldots, s^c_{|s^c|})$, where s^q_1 and s^c_1 are the frames at the top of the stack traces, and $|s^q|$ and $|s^c|$ are the number of frames in s^q and s^c, respectively. Following the majority of the studies, we only consider two frames as equal when their subroutine names are exactly the same. In order to compare s^q and s^c, TraceSim first assigns a weight to each frame of these two stack traces. Then, by means of the NW algorithm, our method finds the optimal maximum global alignment between s^q and s^c by considering the match, mismatch, and gap values proportional to the frame weights. The NW algorithm runs in $O(|s^q| \cdot |s^c|)$ time. Finally, the score of the optimal alignment is normalized by a technique based on the *Jaccard index* [120]. The normalization is fundamental for an effective comparison of alignment scores since it adjusts the scores by the frame weights of the stack traces.

5.2.1 Frame Weight Computation

Bugs are more frequently located in the top frames of stack traces [8]. Hence, it is natural to consider that frames near the top are more relevant for crash deduplication than the bottom ones. Nonetheless, frames, including those in the top positions, could be associated to subroutines that are common through the database, e.g, subroutines that are related to logging, thread pooling, error-handling and entry points. These subroutines poorly accommodate the discrimination of similar stack traces since they appear in multiple crashes caused by different errors. Therefore, we consider that a frame's importance for crash report deduplication (*frame weight*) depends on two factors: its position in the stack trace (*local weight*) and its frequency in the database (*global weight*).

The local weight of the i-th frame in a stack trace s is computed as follows:

$$\mathrm{lw}(s_i) = \frac{1}{i^\alpha}, \tag{5.1}$$

where $\alpha \in \mathbb{R}_{>0}$ is a parameter that controls the function smoothness. Equation (5.1) assigns larger local weight values to frames located at the top of a stack trace.

The global weight is defined based on a well-known information retrieval technique: Term Frequency – Inverse Document Frequency. In our context, the term frequency (TF) is always equal to 1 since the global weight is computed for a frame in a specific position within the stack trace. This is due to the fact that the alignment algorithm is intrinsically dependent on the order of the frames and not only on their frequencies, like in the original TF-IDF. Therefore, given a crash report database S, the inverse document frequency (IDF) of a frame s_i is simply defined as:

$$\mathrm{IDF}(s_i) = \frac{|S|}{\mathrm{df}(s_i)}, \tag{5.2}$$

where $|S|$ is the total number of stack traces in the database S and $\mathrm{df}(s_i)$ is the document frequency of the subroutine s_i, i.e., the number of stack traces that contain a subroutine s_i among the set S of stack traces. Hence, the global weight of the i-th frame in a stack trace s is computed as follows:

$$\mathrm{gw}(s_i) = e^{-\frac{\beta}{\mathrm{IDF}(s_i)}}, \tag{5.3}$$

where $\beta \in \mathbb{R}_{>0}$ is a parameter that controls the function smoothness. The rarer a subroutine is, the larger are the values computed by Equation (5.3).

Finally, the weight of a frame s_i is defined as:

$$\mathrm{w}(s_i) = \mathrm{lw}(s_i) \times \mathrm{gw}(s_i). \tag{5.4}$$

5.2.2 Optimal Global Alignment

Being based on the NW algorithm, TraceSim applies dynamic programming to find the optimal global alignment between stack traces s^q and s^c as follows. Let us define a matrix M in which $M_{i,j}$ is the optimal alignment score between the subsequences $s^q_1, s^q_2, \dots, s^q_i$ and $s^c_1, s^c_2, \dots, s^c_j$. The matrix M is iteratively computed using a bottom-up strategy:

$$M_{i,j} = \max \begin{cases} M_{i-1,j} + \mathrm{gap}(s^q_i) \\ M_{i,j-1} + \mathrm{gap}(s^c_j) \\ M_{i-1,j-1} + F(s^q_i, s^c_j) \end{cases}, \tag{5.5}$$

where $F(s^q_i, s^c_j)$ is given by:

$$F(s^q_i, s^c_j) = \begin{cases} \mathrm{mismatch}(s^q_i, s^c_j), & \text{if } s^q_i \neq s^c_j \\ \mathrm{match}(s^q_i, s^c_j), & \text{otherwise} \end{cases}. \tag{5.6}$$

The *match*, *mismatch* and *gap* values are calculated by the functions match(·), mismatch(·), and gap(·), respectively. These values are proportional to the frames' weights and represent their discriminative power.

The first and second lines in Equation (5.5) are associated with frames aligned to gaps. Since this is an incorrect alignment of only one frame, the gap value for a frame s' is computed as:

$$\mathrm{gap}(s') = -\,\mathrm{w}(s'). \tag{5.7}$$

The first line in Equation (5.6) denotes a mismatch between s_i^q and s_j^c (the two frames are different). Since gap alignments are preferable to mismatches for crash deduplication, the mismatch value is equivalent to lining up the two frames to gaps. This is expressed as:

$$\mathrm{mismatch}(s_i^q, s_j^c) = -\,\mathrm{w}(s_i^q) - \mathrm{w}(s_j^c). \tag{5.8}$$

The second line in Equation (5.6) denotes a match between s_i^q and s_j^c. Inspired by Dang et al. [22], the function match$(\cdot)$ in TraceSim is defined as:

$$\mathrm{match}(s_i^q, s_j^c) = \max(\mathrm{w}(s_i^q), \mathrm{w}(s_j^c)) \times \mathrm{diff}(s_i^q, s_j^c). \tag{5.9}$$

We assume that stack traces emerging from the same error contain subroutines in the same region of the stack trace. Therefore, in Equation (5.9), the maximum weight between the two matched frames is normalized by the diff$(\cdot)$ function that measures the alignment offset of two frames as follows:

$$\mathrm{diff}(s_i^q, s_j^c) = e^{-\gamma|i-j|}, \tag{5.10}$$

where $\gamma \in \mathbb{R}_{>0}$ is a parameter that controls the exponential function smoothness. Thus, we penalize matches in which the positions of the matched frames are discrepant.

Finally, the score of the best global alignment between s^q and s^c is defined as:

$$\mathrm{align}(s^q, s^c) = M_{|s^q|,|s^c|}. \tag{5.11}$$

5.2.3 Normalization

Figure 5.3 illustrates three stack traces ($stack_3$, $stack_4$ and $stack_5$) and their respective frame weights. By applying the algorithm described in Section 5.2.2, we obtain $\mathrm{align}(stack_3, stack_4) = -0.66$ and $\mathrm{align}(stack_3, stack_5) = -0.67$.

According to these alignment scores, $stack_3$ is slightly more similar to $stack_4$ than to $stack_5$. However, this is counter-intuitive since all frames in $stack_3$ and $stack_4$ are different while $stack_3$ and $stack_5$ share four subroutines in the same positions. This problem occurs because all frames in $stack_4$ have a low weight and, consequently, the resulting gaps and mismatches

$stack_3$

A	B	C	D	E
0.01	0.2	0.4	0.01	0.01

$stack_4$

F	H	I
0.01	0.01	0.01

$stack_5$

A	B	G	D	E
0.01	0.2	0.5	0.01	0.01

Figure 5.3 Normalization report example

do not present a significant impact to the maximum score. As such, it is ineffective to compare alignment scores because they depend on the frame weights of the compared stack traces.

Moreover, in order to help users to interpret similarity scores, it is desired to limit scores within a range. Note that according to (5.11), the alignment score is unbounded, i.e., its value might vary between $-\infty$ to $+\infty$. Besides, there exists an asymmetry in the algorithm since match$(\cdot)$ depends on the maximum of the frame weights whereas mismatch$(\cdot)$ is defined by the sum of the weights. Due to these two characteristics, it is challenging to normalize the alignment scores using canonical normalization, e.g. via min-max scaling. Based on the assumption that the proportion of shared subroutines between two stack traces is a good indicator for the deduplication, we propose a normalization inspired by the weighted Jaccard index [121], which computes the similarity between two documents Z and Y as:

$$\text{jaccard}(Z, Y) = \frac{\sum_{k=1}^{|T|} \min(z_k, y_k)}{\sum_{k=1}^{|T|} \max(z_k, y_k)}, \tag{5.12}$$

where T is the set of unique terms in the dataset (called *vocabulary*); and $z, y \in \mathbb{R}^{|T|}$ are vector representations of Z and Y, respectively. Each dimension of z and y corresponds to a specific term in the vocabulary T. This representation is called vector space model (VSM) [23].

In crash deduplication, stack traces can be cast as documents and subroutines in the frames as vocabulary terms. Let us consider V as a vocabulary of subroutines. Thus, a stack trace $s = (s_1, \ldots, s_{|s|})$ can be represented as a vector $x \in \mathbb{R}^{|V|}$ whose k-th dimension is given by:

$$x_k = \sum_{i=1}^{|s|} \text{w}(s_i) \times \mathbb{1}[s_i = t_k], \tag{5.13}$$

where t_k is the k-th term in V and $\mathbb{1}[s_i = t_k]$ returns 1 when the subroutine in a frame s_i

is equal to t_k, and 0 otherwise. In summary, we assign zero to the dimensions of x whose associated subroutines in V do not appear in a stack trace. In the opposite case, the dimension value is the sum of all frame weights $\mathrm{w}(s_i)$ of a specific subroutine t_k.

Given the corresponding vectors x^q and x^c for the stack traces s^q and s^c, respectively, we normalize the score of the maximum global alignment by:

$$\mathrm{sim}(s^q, s^c) = \frac{\mathrm{align}(s^q, s^c)}{\sum_i^{|V|} \max(x_i^q, x_i^c)}. \tag{5.14}$$

Thus, $\mathrm{sim}(s^q, s^c)$ belongs to the interval $[-1, 1]$.

5.2.4 Machine learning

The three parameters of TraceSim (α, β, and γ) are tuned via a machine learning technique: *Tree-structured Parzen Estimator* (TPE) – a Bayesian hyperparameter optimizer [122]. TPE finds parameter values that maximize the sum of two metrics on a given tuning set. Then, such parameters are used on a subsequent validation set for final evaluation. In Section 5.4.4, we describe the training and evaluation processes, including the optimized metrics. In Sections 5.4.2 and 5.4.3, we introduce four additional parameters, also tuned using TPE, that control some preprocessing procedures.

5.3 Related Work

The optimal global alignment as well as the longest common subsequence, the edit distance, and the longest prefix match are well-known sequence matching problems. These problems have been extensively studied over the decades and have been applied to many different domains. In the literature, many studies have addressed crash report deduplication as one of these sequence matching problems.

Brodie et al. [21] proposed a variant of the NW algorithm to compare two stack traces. Similarly to TraceSim, its match value depends on three factors: the position and document frequency of the matched frame in the new report and the alignment offset of the two matched frames. However, unlike TraceSim, the method of Brodie et al. [21] does not contain parameters that control the influence of the frame position and document frequency on the match value. Thus, it cannot adapt to the software project particularities, e.g., the relevance of frame positions for crash deduplication may vary among applications. Moreover, its gap and mismatching values are constant, i.e., they do not depend on frame positions and subroutine document frequencies. Therefore, the optimal alignment score is equally affected by incorrectly matching 1) rare subroutines located at the top, and 2) frequent ones located at the bottom. Finally, the alignment scores are not regularized by the stack trace length and the document frequencies.

Two studies – Bartz et al. [96] and Dhaliwal et al. [13] – proposed techniques based on edit distance for crash report deduplication. Edit distance measures the dissimilarity between two sequences as the minimum number of edit operations (insertions, removals and substitutions) required to convert one of the sequences into another (see e.g. [123]). It was shown by Sellers [124] that edit distance and optimal global alignment are equivalent problems. Bartz et al. [96] designed a logistic regression to calculate the probability of crash reports being duplicate. As features, this method uses the edit distance between two stack traces and categorical data comparisons (event type, process name and exception code). For computing edit distance, the substitution cost depends on the modules, offsets and subroutines of the frames. Besides that, insertion and deletion penalties have different values when a new group (a subsequence of frames with the same module) is created or removed. This method assumes that module and offset information are always present in C/C++ stack traces, which is not necessarily true. Dhaliwal et al. [13] proposed to organize crash reports with a two-level grouping scheme. First, they created a first-level group (coarse granularity) that contains reports with the same frame in the top position. After that, they reorganized the reports in the first level into subgroups (fine granularity). These subgroups are generated based on the edit distance between the reports. The drawback of these two studies is that they ignore two important pieces of frame information: position and document frequency.

Modani et al. [95] reported that prefix match achieves better precision and recall values than the edit distance and the technique proposed by Brodie et al. [21]. Prefix match considers the similarity of two stack traces as the length of the longest common prefix between the stack traces normalized by the size of the longest stack trace. One drawback of Prefix match is that small differences in the top and middle positions can highly affect the computed similarity.

Dang et al. [22] applied an agglomerative hierarchical clustering technique to cluster crash reports. To compute stack trace similarities, they proposed a method called position dependent model (PDM) that finds the optimal common subsequence of two stack traces. PDM employs an algorithm similar to the NW algorithm but for which the gap and mismatch values are zero, i.e., they do not affect the final solution score. Like TraceSim, the match value is computed using the position of the nearest frame to the top and the alignment offset of the matched frames. However, PDM does not consider the document frequency of the frames to compute the similarity score. Therefore, frequent subroutines in the top positions of the stack can highly affect the similarity, even though these subroutines may occur in the top positions of many unrelated stack traces. Moreover, the similarity score is not affected by neither mismatches nor gaps.

Another group of studies proposed techniques based on information retrieval. Lerch and Mezini [93] proposed to use the TF-IDF technique (implemented by Lucene[6]) to calculate the similarity of stack traces. Campbell et al. [10] compared the TF-IDF method (implemented by ElasticSearch[7]) with signature-based methods. According to them, these methods are appropriate for industrial projects since such environments require a search complexity of $O(n \log n)$ where n is the number of reports. In their work, two crash reports were considered duplicate when their similarity score was greater than a defined threshold. The authors found TF-IDF to be superior to other techniques. Besides, the authors proposed a new tokenization that tokenizes camel-cased texts, achieving better cluster metric values by using all data from crash reports.

Sabor et al. [94] proposed DURFEX – a new technique for crash report deduplication. In order to reduce sparsity, DURFEX employs package names instead of fully-qualified method signatures. Besides that, the n-grams of the package names are generated to keep the temporal order of the frames. After preprocessing, DURFEX converts the stack traces to vectors using TF-IDF. The similarity between two reports is then given by the linear combination of the features generated from categorical data comparisons with the cosine similarity of stack trace vectors.

Moroo et al. [102] proposed a re-ranking scheme to combine sequence matching methods with information retrieval. First, they use the TF-IDF method to generate a ranked list of the most similar reports to a query. Then, PDM is employed to calculate a new similarity score for the top-k reports. Finally, the top-k reports are reordered based on the weighted harmonic mean of TF-IDF and PDM. This combination of techniques is limited since a subroutine's document frequency and positions are considered independently for the comparison. TraceSim can compare the frame orders of stack traces using this supplementary information.

Three studies propose methods for crash report deduplication focusing on report and bucket comparison. Kim et al. [98] developed a method called CrashGraph that represents both stack traces and buckets as graphs. The nodes of a graph represent the subroutines, and the edges link nodes whose subroutines are adjacent within the stack traces. The similarity between a stack trace and a bucket is computed as the percentage of edges shared between their graph representations. Koopaei and Hamou-Lhadj [99] proposed CrashAutomata: a method that generates n-grams for each stack trace, and then prunes the n-grams whose

[6]`https://lucene.apache.org/`

[7]`https://www.elastic.co/elasticsearch/`

frequencies exceed a given threshold. An automata is generated for each bucket based on the extracted n-grams of the stack traces. CrashGraph and CrashAutomata can be negatively affected by bucket heterogeneity and they ignore the document frequencies of the subroutines. Ebrahimi et al. [100] trained a Hidden Markov Model (HMM) for each bucket of crash reports. These HMM models are used to detect whether a crash report belongs to a bucket. This method is not scalable since an HMM model has to be trained for each bucket. Moreover, it cannot be applied in software projects whose buckets can contain only one report.

Unlike the existing matching algorithms, TraceSim combines the position and document frequency of frames to compute weights, providing an estimation of the frame's importance to crash deduplication. As supported by the results in Section 5.5, we believe that this scheme improves the method's capability to distinguish relevant frames from irrelevant ones and, consequently, it helps the method to better adjust the similarity score based on the correct (or wrong) matching of frames. Moreover, unlike information retrieval techniques, TraceSim leverages the document frequency of the subroutines without losing track of the frame order.

5.4 Experimental Setup

In this section, we present the main components of our experimental setup: the datasets used in the experiments, preprocessing steps, strategies to compare reports with multiple stack traces, our evaluation methodology, and competing baseline methods. The datasets from open-source applications and the developed code are available online[8].

5.4.1 Datasets

Open data sources that contain crash reports are scarce and the few available are unfit for investigating crash report deduplication. For instance, Mozilla maintains a repository[9] of crash reports related to their products, but bucket assignment is performed automatically by their own system which hinders an accurate evaluation. Therefore, in the literature, one popular alternative for this problem is to mine bug tracking systems (BTS) of open-source applications in order to extract crash reports that include stack traces. Some of these BTSs include manually assigned buckets.

We use four datasets generated from BTS data of open-source projects. Campbell et al. [10] created a crash report dataset from bug reports in the Ubuntu bug repository[10] comprising issues from 617 different software systems for Ubuntu that are compatible with the C debugger.

[8] `https://github.com/irving-muller/TraceSim_EMSE`
[9] `https://crash-stats.mozilla.org/`
[10] `https://bugs.launchpad.net/`

We have generated three other datasets from bug reports of three popular BTSs: Eclipse[11], Netbeans[12] and Gnome[13]. We only considered reports submitted before January 1st 2020. NetBeans and Eclipse are well-known integrated development environments (IDEs) developed in Java, while Gnome's BTS keeps track of bugs from 648 software projects (applications, libraries, bindings, among others) developed for the GNOME desktop environment. We extracted stack traces from the description field and attached files of bug reports. To better imitate real crash reports, we remove the attachments uploaded at most ten minutes after the report creation. This timespan has shown to be suitable for removing files that were uploaded after the bug report inspection by the triaging team. The parser developed by Lerch and Mezini [93] was employed to extract stack traces from Eclipse and NetBeans BTSs, while the Parse::StackTrace[14] module was used to extract stack traces from Gnome BTS.

At some point after a new report is submitted, a user of a bug tracking system analyzes and assigns it to either an existing (duplicate report) or a new bucket (new bug). Thus, there is a time gap between the report submission and the triage assessment. In the meantime, a recently submitted report can be incorrectly labeled. We believe that the span of one year substantially reduces label instability. Therefore, in the Eclipse dataset, we only keep reports created before 2019. NetBeans started to gradually migrate their reports to another BTS in the middle of 2017. Thus, we only consider reports submitted until 2016 for Netbeans. Finally, for Gnome's BTS, we have found a significant reduction in the number of submitted reports with stack traces after 2011 (only $\sim 2.2\%$ of all reports in Gnome were created between 2012 and 2018). The cause of this decrease is unknown, thus we have decided to remove reports submitted during this period to avoid undesirable bias.

Besides open-source projects, we evaluate our method using data from the JetBrains crash report processing system Exception Analyzer. This system handles reports from various products of the IntelliJ Platform product family, which includes IntelliJ Idea, PyCharm, Kotlin Plugin and others. Its products have a large user base, and their maintainers receive several hundreds of crash reports per day. If a product from the IntelliJ Platform crashes, then Exception Analyzer receives the generated report. Newly arrived crash reports are fed to the classification algorithm, which either assigns the report to an existing issue (which means that the report corresponds to an existing bug) or leaves it without treatment. In the latter case, the report status is unclassified and it is considered that a new bug is encountered. Next, unclassified crash reports are grouped together using a clustering algorithm. These

[11]`https://bugs.eclipse.org/bugs/`
[12]`https://bz.apache.org/netbeans/`
[13]`https://bugzilla.gnome.org/`
[14]`https://metacpan.org/pod/Parse::StackTrace`

groups are then manually inspected by an on-duty QA engineer who is selected among the developers of IntelliJ Platform every day. The QA engineer can accept the generated issue candidate and thus, create a new issue, or decline it. The clustering algorithm can create meaningless issues by combining crash reports belonging to different bugs. In this case, the QA engineer can manually move reports belonging to the faulty issue to other issues or leave them untouched. Moreover QA engineer can analyze and move reports from one issue to another, a more suitable one, doing that not only for new issues or new reports, but also for existing ones. QA engineer provides continued activity of supporting stack trace database in a consistent state. The latter approach may be reasonable since new, but similar crash reports can arrive later and then automatic clustering can create a new issue correctly. Finally, new issues are passed to the developers of IntelliJ Platform for fixing. Both classifying and clustering algorithms rely on stack trace comparison.

The statistics of Ubuntu, Netbeans, Eclipse, Gnome, and JetBrains datasets are presented in Table 5.1. For Jetbrains, we show the total number of crash reports (including the automatically classified ones) and the number of manually labeled reports inside parenthesis. For the other datasets, we use manually labeled data only. It is important to highlight that Jetbrains, Nebtbeans, and Eclipse are composed of Java stack traces whereas Ubuntu and Gnome consist of C/C++ stack traces.

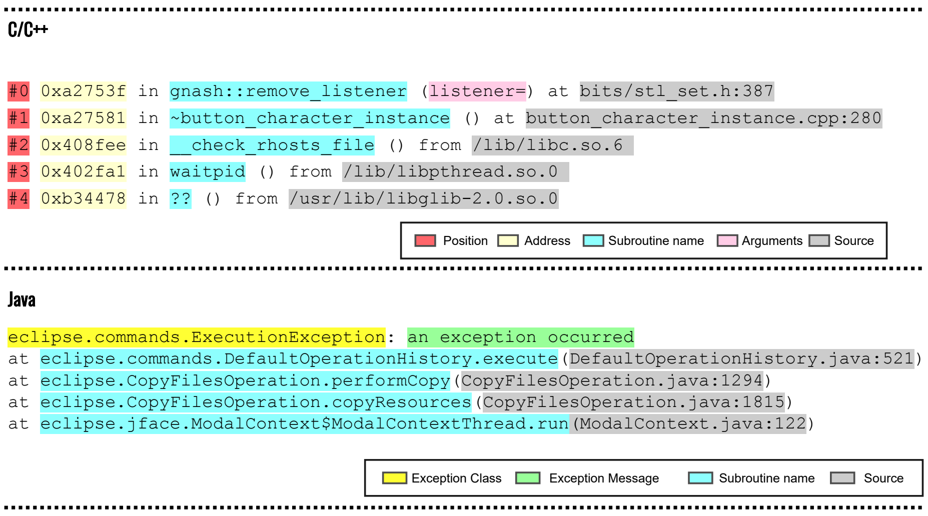
Table 5.1 Statistics of datasets. The number of manually labeled reports are shown inside parenthesis for JetBrains. In the datasets of open-source projects, only manually labeled data are used.

Dataset	**Period**	**# Duplicates**	**# Reports**	**# Buckets**
Ubuntu	2007/05/25 - 2015/10/18	11,468	15,293	3,825
Eclipse	2001/10/11 - 2018/12/31	8,332	55,968	47,636
Netbeans	1998/09/25 - 2016/12/31	13,703	65,417	51,714
Gnome	1998/01/02 - 2011/12/31	117,216	218,160	100,944
Jetbrains	2018/08/09 - 2020/05/20	880,476 (6,516)	925,233 (51,273)	44,757

5.4.2 Preprocessing

The structure of stack traces depends on the programming language. An example of a stack trace in both C/C++ and Java is depicted in Figure 5.4.

As shown in this figure, Java stack traces are typically composed of exception class name and message, subroutine names (fully-qualified name of the method) and location in source code. A subroutine source consists of the file and the line where a subroutine was paused.



C/C++

```
#0 0xa2753f in gnash::remove_listener (listener=) at bits/stl_set.h:387
#1 0xa27581 in ~button_character_instance () at button_character_instance.cpp:280
#2 0x408fee in __check_rhosts_file () from /lib/libc.so.6
#3 0x402fa1 in waitpid () from /lib/libpthread.so.0
#4 0xb34478 in ?? () from /usr/lib/libglib-2.0.so.0
```

🟥 Position 🟨 Address 🟦 Subroutine name 🟪 Arguments ⬜ Source

Java

```
eclipse.commands.ExecutionException: an exception occurred
at eclipse.commands.DefaultOperationHistory.execute(DefaultOperationHistory.java:521)
at eclipse.CopyFilesOperation.performCopy(CopyFilesOperation.java:1294)
at eclipse.CopyFilesOperation.copyResources(CopyFilesOperation.java:1815)
at eclipse.jface.ModalContext$ModalContextThread.run(ModalContext.java:122)
```

🟨 Exception Class 🟩 Exception Message 🟦 Subroutine name ⬜ Source

Figure 5.4 Stack trace example

Stack traces in C/C++ may present a wide variety of information about each frame but their content depends on the debugger and system libraries. For instance, in Figure 5.4, the C/C++ stack trace contains frame positions, frame pointer addresses, subroutine names, arguments, and sources. For both programming languages, *we only extract the subroutine names and the position of the frames*. All remaining information is ignored for crash report deduplication.

We found some subroutine names inconsistencies in C/C++ stack traces. In order to correct them, subroutine names were preprocessed using the following steps. First, since in some cases the parser could not accurately separate arguments from subroutine names, we had to search for these inconsistencies and remove them from subroutine names manually. After that, we stripped the pattern __GI__ and underscore symbols (_) from the beginnings of names since these prefixes are likely inserted by the debugger or compiler. Therefore, following these steps, for instance, __GI____libc_free (mem=0x3) and __libc_free are transformed to libc_free.

When a debug package of a software system is not installed on a machine, the stack trace may contain frames for which information about the subroutine call is missing. In these frames, subroutines are represented as ?? in C/C++ and HIDDEN.HIDDEN in Java. We refer to these subroutines as *unknown subroutines*. In our experiments, we test two different strategies to handle such subroutines: the first approach considers them as equivalent for stack trace comparison, while the second treats them as different. Although the first strategy prevents the wrong comparison of different subroutines, the second one can detect patterns of subsequences with unknown subroutines.

Removing recursion is an important preprocessing step [21, 95]. We test two different recursion removal algorithms. The first algorithm, proposed by Brodie et al. [21], removes subsequent frames with the same subroutine names. The second one, developed by Modani et al. [95], strips all frames that occurred between two similar frames of the same subroutine. Finally, to remove uninformative functions, we employed the unsupervised algorithm created by Modani et al. [95]. In this method, a frame is considered uninformative when the document frequency percentage of its subroutine name is higher than a threshold. Consecutive uninformative frames in the top and bottom positions are removed from the stack traces.

5.4.3 Multiple Stack Traces

Crash reports may include multiple stack traces, mainly due to multi-processing and multi-threading systems. When such systems crash, each process or thread usually generates a specific stack trace. Since the information of which thread/process that caused the crash may be unknown, all stack traces are considered for deduplication. Another cause is related to the data characteristics. In Netbeans and Eclipse BTSs, bug reports can include multiple stack traces within their description and attached files. According to Schroter et al. [8], these extra stack traces provide additional information about an issue. Thus, we keep all stack traces found in a report. Finally, stack traces from a nested exception were considered as different stack traces since some of them are related to process/thread executions and the extraction method proposed by Lerch and Mezini [93] incorrectly considers a significant amount of nested exceptions to be single stack traces. In Table 5.2, we present the number of reports with multiple stack traces.

In order to compute the similarity between two crash reports q and c that contain multiple stack traces, we compute the similarity of all possible pairs of stack traces, in which one member of the pair belongs to the query crash report q and the other comes from the candidate crash report c. Thus, a similarity matrix $S \in \mathbb{R}^{m \times n}$ is created where $S_{i,j}$ is the similarity between the i-th and j-th stack traces in q and c, respectively. We then assess six different strategies to reduce this matrix to a real number. The first strategy performs the reduction

Table 5.2 Number of reports with multiple stack traces (ST), total number of reports, and the ratio of these two quantities for each dataset.

BTS	**# Reports w/ Multiple ST**	**# Reports**	**Ratio**
Ubuntu	5	15,293	0.03%
Eclipse	13,641	55,968	24.37%
Netbeans	40,147	65,417	61.37%
Gnome	174,841	218,160	80.14%

as follows:

$$\text{max\_stg}(S) = \max_{1 \leq i \leq m, 1 \leq j \leq n} S_{i,j}. \tag{5.15}$$

Basically, this strategy returns the highest value in the similarity matrix.

The next five strategies perform matrix reduction by applying a maximum operation followed by a mean operation. The first one is defined as follows:

$$\text{query\_stg}(S) = \frac{1}{m} \sum_{i}^{m} \max_{1 \leq j \leq n} S_{i,j}. \tag{5.16}$$

query_stg(·) computes the average of the maximum similarity of each stack trace in the query. A similar strategy can be applied to the stack traces in the candidate:

$$\text{cand\_stg}(S) = \text{query\_stg}(S^{\intercal}). \tag{5.17}$$

Instead of considering the stack trace sources, the third strategy calculates the mean of maximum values of the shortest report (report that contains the smallest number of stack traces):

$$\text{short\_stg}(S) = \begin{cases} \text{query\_stg}(S), & \text{if } m \leq n \\ \text{cand\_stg}(S), & \text{otherwise.} \end{cases} \tag{5.18}$$

The opposite strategy is defined as follows:

$$\text{long\_stg}(S) = \begin{cases} \text{query\_stg}(S), & \text{if } m \geq n \\ \text{cand\_stg}(S), & \text{otherwise.} \end{cases} \tag{5.19}$$

Finally, the last approach is:

$$\text{avg\_stg}(S) = \frac{\text{query\_stg}(S) + \text{cand\_stg}(S)}{2}. \tag{5.20}$$

5.4.4 Proposed Evaluation Methodology

In previous literature, there is no widely adopted methodology for evaluation of crash report deduplication systems. Nevertheless, two common approaches are *ranking* [93, 94] and *binary classification* [95, 96]. Both approaches have their own strengths, as well as some key limitations. In ranking approaches, a query crash report q is given and its similarity to each previously submitted report is computed. Then, a ranked list of candidate reports, sorted by decreasing similarity to q, is evaluated by means of classic ranking metrics. The higher the system ranks duplicates of q in that list, the better its performance is. However, ranking metrics are usually not defined for singleton (non-duplicate) queries. This is a key drawback of ranking methodologies because they disregard the ability of a system to filter out singleton crash reports. This is highly undesirable given the large volume of crash reports submitted to a typical crash report system. In contrast, binary classification approaches focus exactly on this filtering task. Such approaches tackle the classification problem of predicting if a query crash report is either duplicate or non-duplicate. However, they ignore one, if not the most, important aspect of crash report deduplication: identification of reports concerning the same software bug. In summary, the blind spot of binary classification approaches is covered by ranking approaches, and vice-versa.

Other popular approaches [10, 22, 102] treat crash report deduplication as a clustering problem by considering buckets of reports as clusters. The clustering metrics consider the *global solution* to measure the grouping quality, i.e., all reports are used to compute the evaluation metrics. However, in practice, software projects frequently contain an initial repository in which submitted reports are already pre-assigned to buckets. Since these previously submitted reports are considered in our evaluation methodology, we have opted to not using clustering metrics here. Instead, we have adopted metrics that are not affected by the presence of earlier reports.

Bug report deduplication is a problem related to crash report deduplication. Its input is a textual description of a software issue. The body of literature regarding this problem is larger than that for crash report deduplication. Inspired by Banerjee et al. [15], which proposed an evaluation methodology for bug report deduplication, our evaluation methodology combines both ranking and binary classification metrics. For ranking, we use two classic metrics: Mean Average Precision (MAP) and Recall Rate@k (RR@k) [14]. For binary classification, we use the classic Area Under the ROC Curve (AUC). Thus, our methodology can measure the system capacity to filter duplicate and non-duplicate reports, compute the percentage of duplicate reports correctly assigned to buckets, and evaluate ranking quality. Additionally, Rakha et al. [111], also in the context of bug report deduplication, suggest to evaluate a system on different portions of a dataset in order to better study how varies performance due to data changes. Moreover, this idea inspired us to develop our own methodology to crash report deduplication.

Query and Candidate Sets

In our methodology, a dataset of crash reports is always organized in chronological order, as this better reflects the real scenario of software engineering projects. In order to compute each metric, we first select a *query set* Q of consecutive crash reports within a given dataset.

In Figure 5.5, we illustrate a chronologically-ordered dataset, a query set Q within it, and other key aspects of our methodology. When evaluating a system, each query report $q \in Q$ is considered as a newly submitted crash report; and reports submitted before q are considered *candidate reports*, i.e. possible duplicates of q. More specifically, for each query report $q \in Q$, we define a corresponding candidate set $C(q)$ with reports in the dataset that were submitted before q.

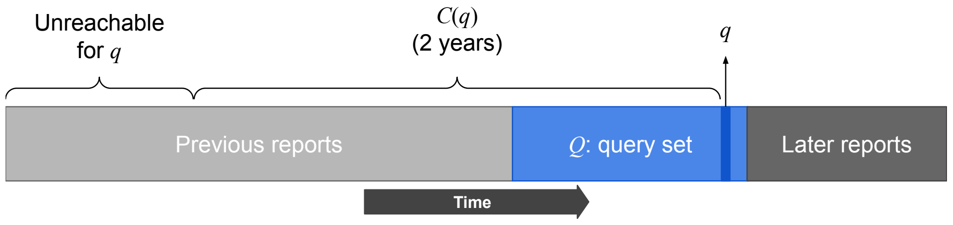


Figure 5.5 Illustration of a chronologically-ordered dataset in which we select a query set Q (blue span). Given a query report $q \in Q$, its corresponding candidate set $C(q)$ is shown.

Same as bug report datasets, crash report datasets can be very large. Banerjee et al. [15] suggest to limit the candidate set $C(q)$ to a certain *time window* in the context of bug report deduplication. This way, we reduce both the computational cost of duplicate report detection and the performance degradation due to the repository growth over time. As depicted in Figure 5.5, we limit $C(q)$ to a time window of two years. That is, for any $q \in Q$, $C(q)$ comprises all reports in the dataset submitted at most two years before q. Reports submitted more than two years before q are not reachable by the systems being evaluated.

In Table 5.3, we illustrate a dataset comprising 11 reports identified as $C1$, $C2$, . . . , $C11$. The query set $Q = \{C7, C8, C9\}$ is highlighted in blue. The first step of evaluating a system for a given query set consists of computing $\text{sim}(q, c)$ for all $q \in Q$ and $c \in C(q)$. For our example, we present these values in the columns labeled as $\text{sim}(q, \cdot)$ for $q \in Q$. In these columns, a value of UR in a row c indicates that $c \notin C(q)$, i.e., report c is unreachable for query q. For example, when $q = C9$, reports $C1$, $C2$, and $C3$ are unreachable because they were submitted more than two years before $C9$. Additionally, all reports submitted after q are unreachable for q.

Limiting $C(q)$ by a time window is usually not a big issue since, in most cases, crash reports related to a bug are frequently submitted until the bug is fixed. That is, for most bugs, there is not a large gap between duplicate reports. In order to show that this is true for our datasets, we present in Table 5.4 the percentage of query reports that have at least one duplicate bug within a time window of two years. In the worst case (Eclipse), for less than 2.7% of all

Table 5.3 Example of a dataset: query set $Q = \{C7, C8, C9\}$ (blue rows) and similarities computed by a fictitious system between each query $q \in Q$ and candidate $C(q)$. An UR label indicates that a candidate report c is *UnReachable* for a query q.

Id	**Creation Date**	**Bucket**	$\mathrm{sim}(C7, \cdot)$	$\mathrm{sim}(C8, \cdot)$	$\mathrm{sim}(C9, \cdot)$
$C1$	2014/12/02	B_{C1}	0.0	0.2	UR
$C2$	2014/12/24	B_{C1}	0.0	0.5	UR
$C3$	2015/01/01	B_{C3}	0.2	0.1	UR
$C4$	2015/06/12	B_{C3}	0.7	0.0	0.8
$C5$	2016/02/22	B_{C3}	0.3	0.3	0.1
$C6$	2016/05/25	B_{C6}	0.6	0.4	0.3
$C7$	2016/05/26	B_{C6}	UR	0.0	0.2
$C8$	2016/12/01	B_{C8}	UR	UR	0.1
$C9$	2017/05/25	B_{C3}	UR	UR	UR
$C10$	2017/05/26	B_{C10}	UR	UR	UR
$C11$	2017/11/02	B_{C8}	UR	UR	UR

possible query reports, no previous duplicate report is reached using a window of two years.

For JetBrains, we keep the original two-month time window employed in their system. We have found that 96.60% of query reports in the Jetbrains data can reach at least one duplicate report in this time window.

Table 5.4 Percentage of query reports that reaches at least one duplicate report in a time window of two years.

Dataset	**2 years**
Ubuntu	99.47%
Eclipse	97.36%
Netbeans	98.68%
Gnome	99.50%

Bucket-Level Metrics

In our exemplary dataset in Table 5.3, we include a column that indicates the bucket of each report. A bucket is identified as B_m where m corresponds to its master report, i.e., the first submitted (oldest) report in the bucket. For instance, bucket $\{C6, C7\}$ is denoted B_{C6}. When evaluating a system, we consider that the correct buckets in $C(q)$ are known. Thus, a system does not need to predict duplicate reports, but duplicate buckets, instead. In that way, our metrics are defined in terms of *candidate buckets* instead of candidate reports.

We denote $C^B(q)$ the set of candidate buckets for a query q. This set is derived from $C(q)$, that is, $C^B(q)$ comprises buckets whose reports are in $C(q)$. For example, we have that $C^B(C7) = \{B_{C1}, B_{C3}, B_{C6}\}$. We then define the similarity $\text{sim}(q, B)$ between a query report $q \in Q$ and a bucket $B \in C^B(q)$ as the maximum similarity between q and a candidate report $c \in B$, that is:

$$\text{sim}(q, B) = \max_{c \in B} \text{sim}(q, c). \tag{5.21}$$

In our example, we have $\text{sim}(C7, B_{C3})$=0.7. As shown by Equation (5.21), all reports from B, even those outside the two-year time window, are considered to compute $\text{sim}(q, B)$. This is natural since we want to capture the full similarity between B and q.

In Table 5.5, we present the similarity for all pairs $q \in Q$ and $B \in C^B(q)$ for this dataset. We can observe that some buckets in the dataset are unreachable for some queries (the UR value in the table). That is the case when, for a given query, all reports of some bucket are unreachable. For instance, when $q = C9$, bucket B_{C1} is unreachable because all its reports ($C1$ and $C2$) in the dataset are unreachable for $C9$. Regarding the same query, B_{C3} is reachable since at least one of its reports is reachable for $C9$, e.g., $C_4 \in C(C9)$. Thus, all the reports in B_{C3} are considered to calculate $sim(C9, B_{C3})$ including $C3$ that is not within the time window.

Table 5.5 Similarity matrix $\text{sim}(q, B)$ between each query report $q \in Q$ and each bucket $B \in C^B(q)$ for the dataset in Table 5.3. A UR value indicates that all reports in a bucket B are unreachable for a query q. In the last column, we present the correct bucket for each query report.

Query q	**Buckets**				**Correct Bucket**
	B_{C1}	B_{C3}	B_{C6}	B_{C8}	
$C7$	0.0	0.7	0.6	UR	B_{C6}
$C8$	0.5	0.3	0.4	UR	B_{C8}
$C9$	UR	0.8	0.3	0.1	B_{C3}

Based on the similarities between queries and their candidate buckets, our methodology evaluates a method for crash report deduplication by using *ranking* and *binary classification* metrics.

Ranking Metrics

As mentioned before, ranking metrics disregard a query that corresponds to a singleton report. For the query set $Q = \{C7, C8, C9\}$ in Table 5.3, when $q = C8$, our methodology considers $C8$ as a singleton report since it is the first report of its bucket (master report). Although $C11$ is a duplicate of $C8$ in the dataset, $C11$ is not considered in this case since it is submitted after $C8$. Therefore, we only consider two duplicate reports in Q ($C7$ and $C9$) to compute the ranking metrics. The set of duplicate reports within Q is denoted $Q^d \subset Q$. Given the similarities between a duplicate query $q \in Q^d$ and each of its candidate buckets $C^B(q)$, we sort this set in descending order of similarity. We define this sorted list as $L(q) = (B^s_1, B^s_2, \ldots, B^s_{|C^B(q)|})$, where $B^s_i \in C^B(q)$. In our example, $Q^d = \{C7, C9\}$ and we have: $L(C7) = (B_{C3}, B_{C6}, B_{C1})$ and $L(C9) = (B_{C3}, B_{C6}, B_{C8})$.

The first ranking metric is MAP which is the mean of the Average Precision (AP) for all queries in Q^d:

$$\text{MAP} = \frac{\sum_{q \in Q^d} \text{AP}(q)}{|Q^d|}. \tag{5.22}$$

In our scenario, AP is very simple because, for a query $q \in Q^d$, there is only *one* relevant bucket in $C^B(q)$ that is the correct bucket for q. For a query $q \in Q^d$, AP is given by:

$$\text{AP}(q) = \frac{1}{p}, \tag{5.23}$$

where p is the position of the correct bucket for q in the sorted list of candidate buckets $L(q)$. In our example, $\text{AP}(C7) = 1/2$ and $\text{AP}(C9) = 1$. An AP equal to one means that the system ranked the correct bucket in first place.

MAP is a relevant ranking metric, especially when comparing different ranking systems. However, when we consider a realistic scenario in which a manual triage of possible duplicate reports is necessary, the Recall Rate@k metric is more informative. This metric is defined as:

$$\text{RR@}k = \frac{\sum_{q \in Q^d} \mathbb{1}_k(q)}{|Q^d|}, \tag{5.24}$$

where $k \geq 1$ is an integer parameter of the metric and $\mathbb{1}_k(q)$ is an indicator function whose value is one when the correct bucket for q is ranked within the first k positions in $L(q)$. This way, RR@k is the percentage of ranked lists in which the correct bucket is within the top-k positions. RR@1 is the percentage of duplicate queries for which the correct bucket is ranked first, corresponding to the accuracy of a completely autonomous system.

Due to the two-year time window, the correct bucket of a query q might not appear in $C^B(q)$ and, therefore, its position is undefined in $L(q)$. This case occurs in real scenarios and it negatively affects the performance of crash report deduplication systems. To reproduce this impact on real environments, we consider $\text{AP}(q) = 0$ and $\mathbb{1}_k(q) = 0$ for each query q whose correct bucket is not in $L(q)$.

Binary Classification Metric

As discussed before, ranking metrics have a relevant limitation: they ignore non-duplicate reports in the query set. When considering a realistic scenario in which manual triage is necessary, the ability of a system to filter out non-duplicate reports is highly valuable. In the following, we explain how to cast a similarity-based system as a binary classifier that predicts if a query report is duplicate or not.

Given a query $q \in Q$ (including singletons) and the corresponding sorted list of candidate buckets $L(q) = (B_1^s, B_2^s, \ldots)$, we use the highest similarity among all candidate buckets, that is, $\text{sim}(q, B_1^s)$, as a classification score. For the example in Table 5.5, we have the classification scores: $\text{sim}(C7, B_1^s)$=0.7, $\text{sim}(C8, B_1^s)$=0.5 and $\text{sim}(C9, B_1^s)$=0.8. Since $\text{sim}(q, \cdot) \in [-1, 1]$, we can derive a binary classifier by defining a threshold t such that q is considered duplicate if $\text{sim}(q, B_1^s) \geq t$. In our evaluation, we do not need to choose t, because we use the classic Area Under the ROC Curve metric. The ROC curve is a plot of the true positive rate versus the false positive rate for every possible classification threshold. AUC summarizes the ROC curve in one meaningful number between zero and one. For example, the AUC for the query set in Table 5.5 is equal to one.

Parameter Tuning and Model Validation

TraceSim and other methods include some parameters that need to be tuned. In order to avoid reporting overestimated performance, we tune parameters on a query set denoted *tuning set* T and then, using the best parameters, we report final performance on a consecutive and non-overlapping query set denoted *validation set* V. Since some concept drift along time in most crash report datasets is expected, the tuning set comprises the reports immediately preceded by the reports in the validation set. In Figure 5.6, we depict these two sets within a chronologically-ordered dataset. In the figure, we highlight two query reports: $q^t \in T$ and $q^v \in V$. We can notice that the candidate sets $C(q^t)$ and $C(q^v)$ can overlap, but the corresponding query sets T and V do not. The time period of a validation set V is one year. The corresponding tuning set T is delimited such that $|T^d| = 250$, that is, T contains 250 duplicate reports along with all singleton reports submitted in the same period. We have found that $|T^d| = 250$ leads to good results, and that larger tuning sets did not improve overall performance.

As mentioned in Section 5.2.4, we tune parameters by means of TPE, a machine learning technique. We run TPE for 100 iterations[15] and choose the best parameters based on the sum

[15]We conducted a preliminary investigation to find the best number of iterations.

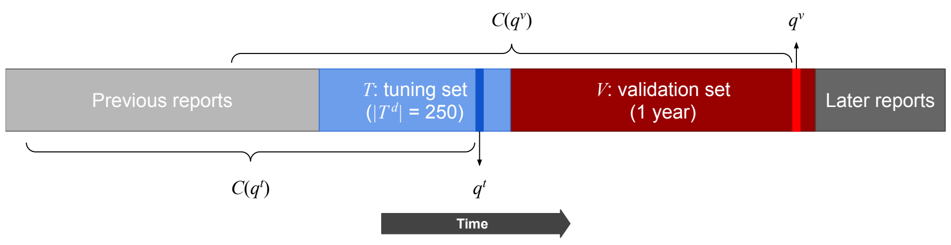


Figure 5.6 Depiction of tuning and validation sets within a dataset.

of MAP and AUC values on the tuning set. The selected parameters are then used to compute the three considered metrics (MAP, AUC and RR@k) on the corresponding validation set. The tuned parameters are:

- the ones that control the frame weights used by the optimal global alignment algorithm: α, β and γ (Sections 5.2.1 and 5.2.2);
- the approaches to handling unknown subroutines (Section 5.4.2);
- the threshold to consider a subroutine as uninformative (Section 5.4.2);
- recursion removal algorithms (Section 5.4.2);
- the strategies to reduce the similarity matrix (Section 5.4.3).

Because of the natural concept drift in our datasets, the performance of a single method usually varies a lot from one query set to another, even within the same dataset. Thus, Rakha et al. [111] suggested to perform experimental evaluation on different portions of the dataset. Based on that suggestion, we perform our experiments as follows. Along each dataset, we randomly sample 50 validation sets. For each validation set, a corresponding tuning set is selected comprising reports submitted immediately before the validation reports. In Figure 5.7, we illustrate an example of five randomly selected validation sets within a dataset, along with the corresponding tuning sets. As one can observe, the selected query sets may overlap.

Unlike the datasets derived from open source projects, Jetbrains contains much more reports that were labeled by an automated system. To replicate a similar experimental setup to the production environment of Jetbrains, we consider such reports for the experiments. However, to mitigate the impact of mislabelling on the evaluation, automatically classified reports in the tuning and validation sets are disregarded for computing the ranking and binary classification metrics. That is, we do not consider these reports as queries, even though they can be in the candidate sets and they are used to compute the document frequency of the subroutines.

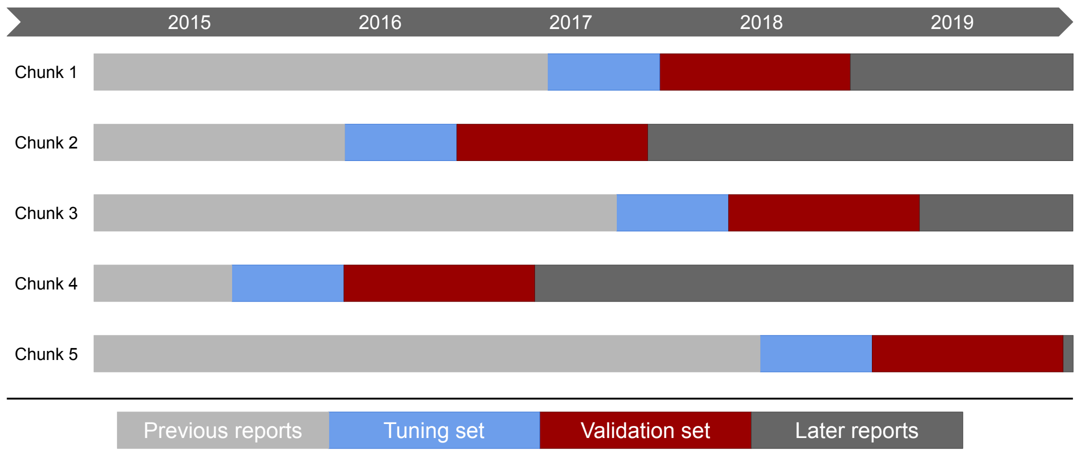


Figure 5.7 Five randomly selected validation sets along with the corresponding tuning sets within the same dataset.

Moreover, since the Jetbrains dataset contains much more reports than the other datasets, the time period of their validation is set to only one month and the number of iterations for TPE is decreased to 50. The number of sampled validation sets and the size of the tuning set are not modified.

5.4.5 Competing Methods

In order to evaluate TraceSim we have selected a number of baselines. For this, competing methods from the information retrieval, string matching, and machine learning areas were selected. They have been selected due to TraceSim being a structural composition of the methods belonging to these groups.

In information retrieval category we have selected two techniques: TF-IDF and DURFEX. TF-IDF was implemented in Apache Lucene. Campbell et al. [10] showed that it achieves poor performance when using only stack trace information. In fact, following Lerch and Mezini [93], we treat subroutine names as single terms. DURFEX is only tested in Eclipse and Netbeans datasets since it was specifically developed for Java stack traces. Besides that, in this method, we only consider the cosine similarity of the stack trace vectors for crash report deduplication. We also compare TraceSim to five other methods: PDM, Prefix Match, the original NW algorithm, the matching algorithm proposed by Brodie et al. [21], and the reranking method designed by Moroo et al. [102]. These methods are described in Section 5.3. Hereafter, Prefix Match is abbreviated to *PrefixM*, and we denote the techniques proposed by Brodie et al. [21] and Moroo et al. [102] as *Brodie* and *Moroo*, respectively. Finally, for the sake of fairness, all methods have access only to the positions and subroutine names of the frames in the stack traces.

We disregard some previous methods in our experiments due to different reasons. Top signature-based methods have been shown to achieve worse performances than TF-IDF [10]. The technique proposed by Bartz et al. [96] depends on features (frame offsets and module names) that are not available in a significant portion of the stack traces in the datasets. CrashGraph [98] leverages all crash reports of a bucket to generate a bucket representation. As the majority of the works in the literature, our paper focuses on the similarity of stack traces and, therefore, CrashGraph is beyond the scope of this study. Ebrahimi et al. [100] propose a method that requires predefined buckets because an HMM is trained for each bucket. Koopaei and Hamou-Lhadj [99] also assume a fixed number of buckets to evaluate CrashAutomata and it is uncertain how to employ this technique correctly in a scenario that does not hold such an assumption. Since our evaluation methodology considers that singletons and new buckets may be generated during evaluation, which is typical in real projects, we do not consider these two previous methods.

Regarding the competing methods with learned parameters, their original studies either do not describe the training process or use grid search to tune the hyperparameters. Since Tree-structured Parzen Estimator achieves similar or better performance than grid search [122, 125, 126], this Bayesian optimization technique is used to tune the parameters of the competing methods for each chunk. Table 5.6 presents all tuned parameters for each method. For all methods, except Durfex and TF-IDF[16], we also tune the following preprocessing choices: the strategies to reduce the similarity matrix; the approaches to handling unknown subroutines; the threshold of considering a subroutine as uninformative; and the recursion removal algorithms.

5.5 Experimental Results

In this section, we compare TraceSim to the competing methods regarding AUC, MAP, and RR@1 on Ubuntu, Eclipse, Netbeans, Gnome, and JetBrains datasets. The statistics of these datasets are presented in Table 5.1 (Section 5.4.1). We also present an ablation study to assess the main components of TraceSim and investigate the effectiveness of our method of computing mismatch and gap values. As previously described, we consider 50 random validation sets for each dataset. We use the same validation sets, and the corresponding tuning sets, to evaluate all methods using the three aforementioned metrics. We then report, for each method, the distribution of each metric in the 50 validation sets using violin plots [127].

[16]These strategies were designed for techniques that consider the frame order. Since these information retrieval techniques are based on the bag-of-words model, such strategies are not effective for them.

Table 5.6 Tuned parameters for each method. For competing methods, we keep the original names.

Method	**Parameter**
TF-IDF	No learnable parameters
PrefixM	No learnable parameters
DURFEX	N-gram
NW algorithm	Match, mismatch and gap values
Brodie	Gap value
PDM	c and o
Moroo	c, o, α, and M
TraceSim	α, β and γ

These plots are produced by the standard kernel density estimation (KDE) as implemented in the `seaborn` library [128]. For each violin plot, we present: the estimated distribution curve; three dashed lines indicating the 25th, the 50th, and the 75th percentiles; and a white dot indicating the mean metric value.

Additionally, when comparing TraceSim to a competing method (including different versions of itself in the ablation study), we compute the difference between the performance obtained by TraceSim and the competing method in each validation set. These differences in terms of AUC, MAP, and RR@1 are denoted, respectively, ΔAUC, ΔMAP, and ΔRR@1. These values are positive whenever TraceSim outperforms a competing method on a validation set. For each competing method, we plot the distribution of the 50 differences by means of ordinary box plots. In these plots, we include a white point to indicate the mean difference between TraceSim and the competing method. Finally, following [111], we apply the Wilcoxon signed-rank test [129] to evaluate whether the obtained performance differences are statistically significant. The statistical hypotheses are:

H_0: The two methods have the same performance.

H_1: The two methods have different performance.

The null hypothesis (H_0) is rejected in favor of the alternative hypothesis (H_1) whenever $p < 0.01$. We indicate statistical significance by appending the symbol ★ to the name of the competing method within the corresponding box plot.

5.5.1 Results

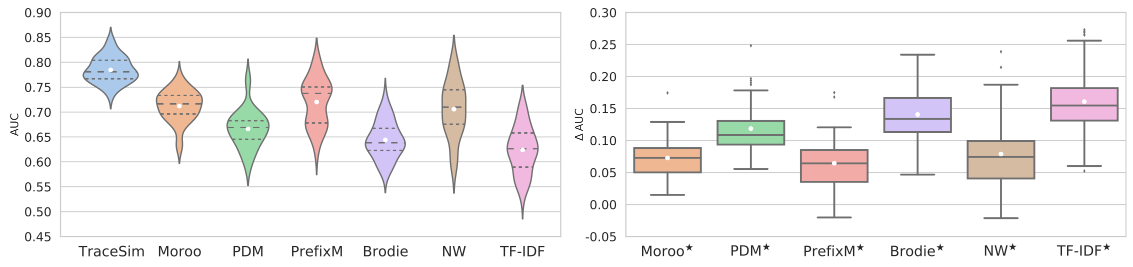
In Figure 5.8 (left), we present the distributions of the AUC values achieved by TraceSim and each competing method on the five considered datasets. In turn, in Figure 5.8 (right), we depict the performance differences between TraceSim and other methods in terms of AUC, i.e., ΔAUC.

TraceSim consistently achieves competitive AUC values in all datasets. It significantly outperforms the second best technique by, on average, 6.44% in Ubuntu, 2.01% in Eclipse, and 1.39% in Netbeans. In Gnome, TraceSim substantially surpasses all methods except the NW algorithm which presents similar performance (the mean of ΔAUC is approximately zero). Finally, in the JetBrains dataset, our method yields AUC values comparable to Moroo, PDM, and PrefixM. The average of ΔAUC between TraceSim and these techniques are +0.40%, -0.06%, and -0.22%, respectively. However, these differences are not statistically significant.

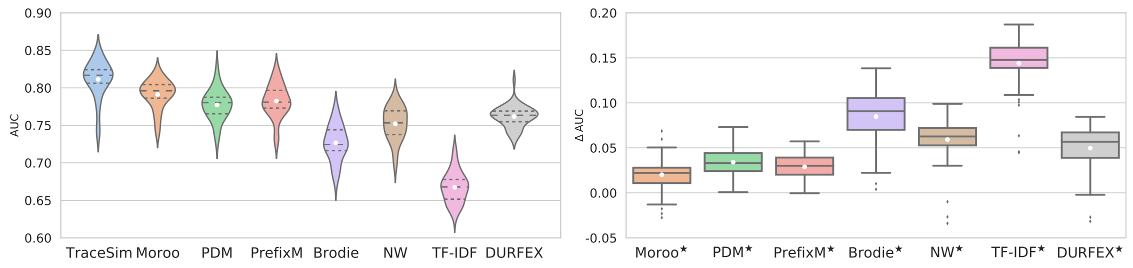
In Figure 5.9 , we present the results of the same experiments but now with respect to the MAP values obtained by TraceSim and each competing method on the five datasets. In general, TraceSim significantly outperforms the majority of the methods regarding MAP in all datasets. The exceptions are TF-IDF in Gnome and PDM in Eclipse, Netbeans, and JetBrains. Although TraceSim surpasses TF-IDF by 2.35% on average in Gnome, ΔMAP is not statistically significant in this case (the variance varies from -10.46% to 18.87% and the median is relatively close to 0). Regarding PDM, the average of ΔMAP between TraceSim and this method are +0.76% in Eclipse, -0.04% in Netbeans, and +0.25% in JetBrains. However, we consider the performance of these methods comparable since the differences in their results are not significant.

Regarding the RR@1 evaluation metric (Figure 5.10), TraceSim significantly outperforms most of the methods in all datasets. Similar to the MAP analysis, we do not find statistically significant differences in three exceptional cases: TF-IDF in Gnome, and PDM in both Eclipse and JetBrains. However, we observe distinctive findings in NetBeans: TraceSim significantly underperforms PDM (-1.11% on average), achieving results comparable to PrefixM and Moroo. Even though TraceSim is not dominant in Netbeans regarding RR@1, it consistently achieves competitive RR@1 values across all datasets. For example, on average, TraceSim greatly surpasses PDM by 7.47% in Ubuntu and 12.93% in Gnome.

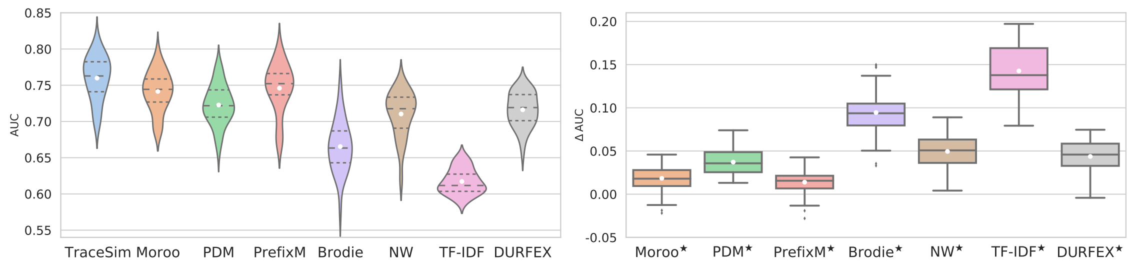
In general, TraceSim consistently achieves competitive performance for each different combination of dataset and metric. Actually, in the majority of the experimented scenarios, it significantly outperforms all competing methods. Our method is surpassed by PDM in a unique scenario (RR@1 on Netbeans). However, in this same dataset, TraceSim substantially



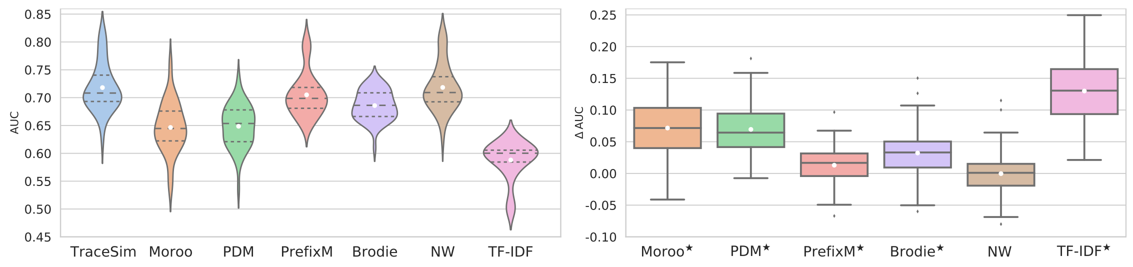
(a) Ubuntu



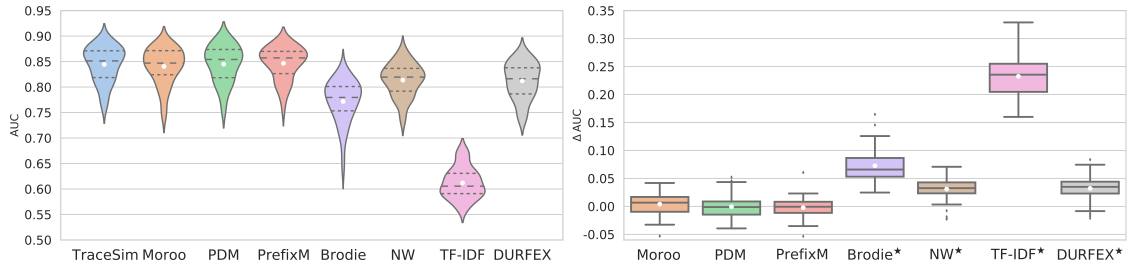
(b) Eclipse



(c) Netbeans



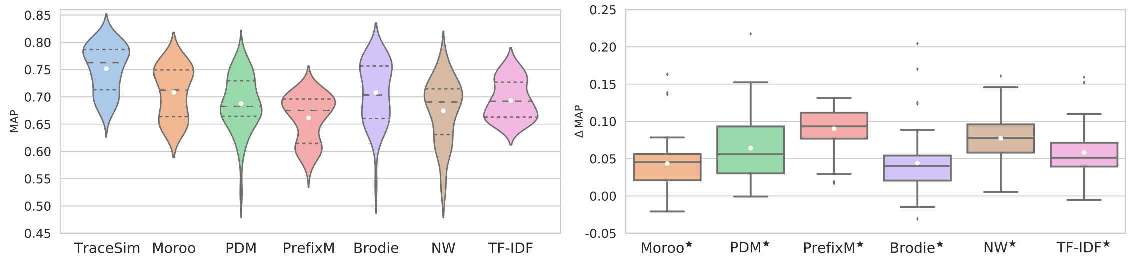
(d) Gnome



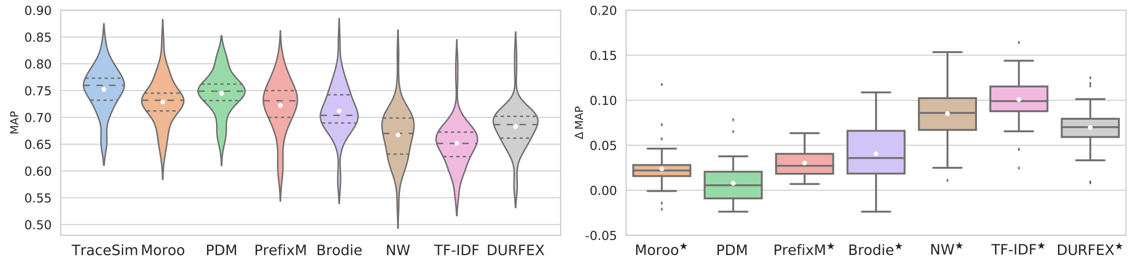
(e) JetBrains

★ $p < 0.01$

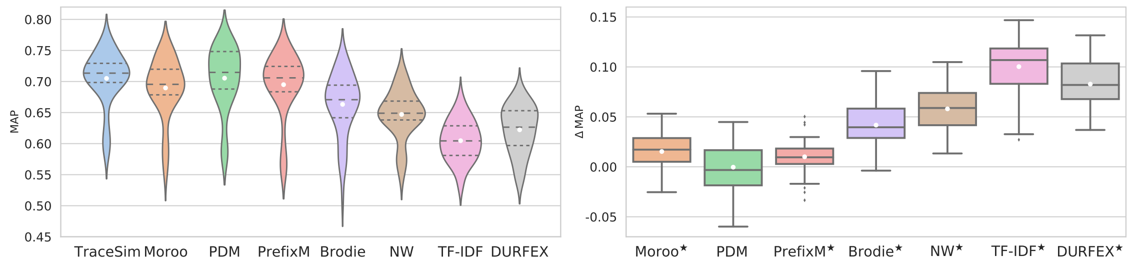
Figure 5.8 Results regarding AUC. *Left*: distribution of AUC achieved by TraceSim and each competing method in all validation sets of each dataset. *Right*: ΔAUC between TraceSim and each competing method.



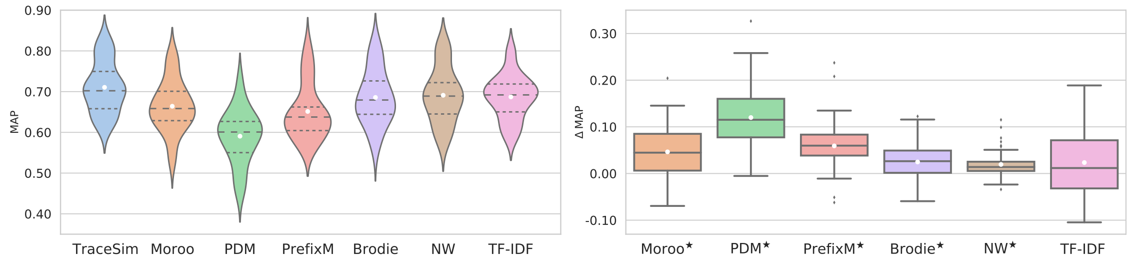
(a) Ubuntu



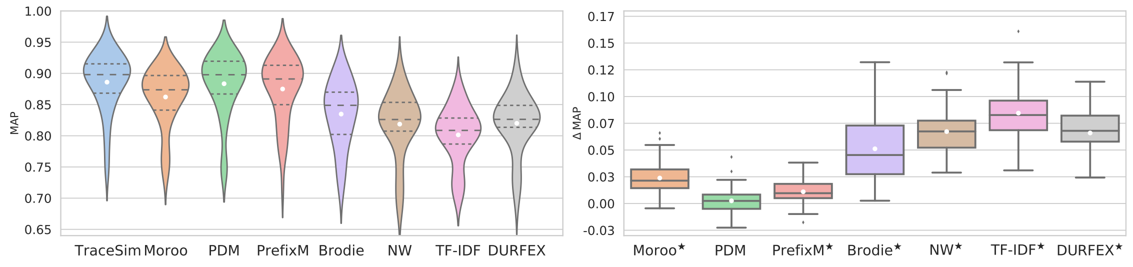
(b) Eclipse



(c) Netbeans



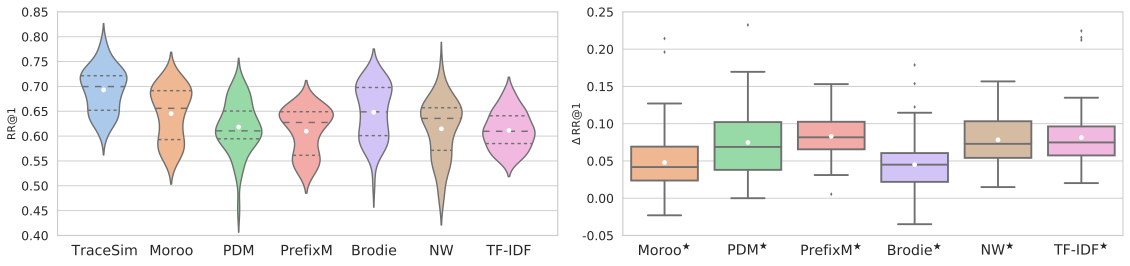
(d) Gnome



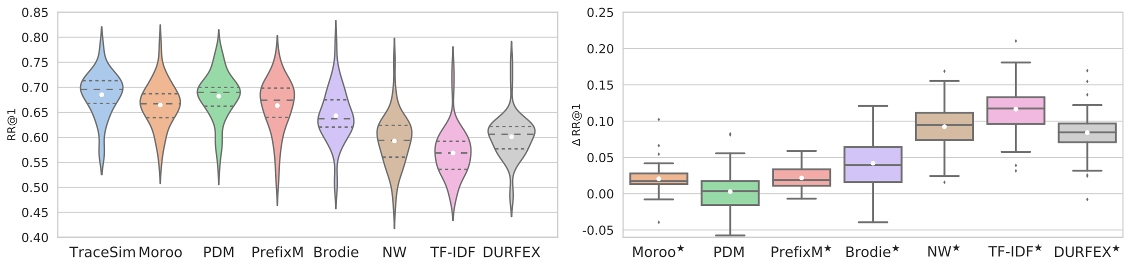
(e) JetBrains

★ $p < 0.01$

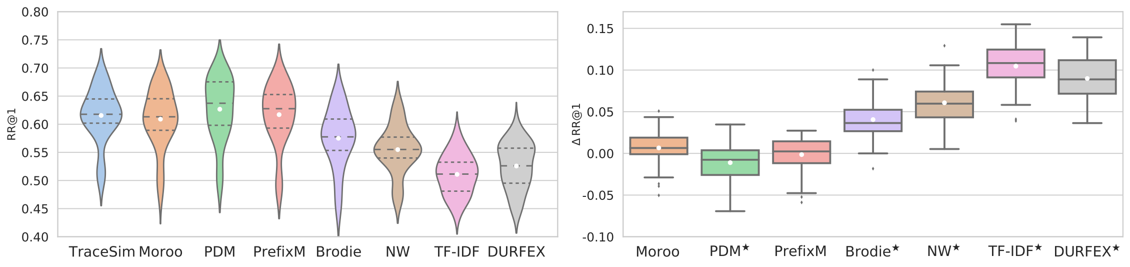
Figure 5.9 Results regarding MAP. *Left*: distribution of MAP achieved by TraceSim and each competing method in all validation sets of each dataset. *Right*: ΔMAP between TraceSim and each competing method.



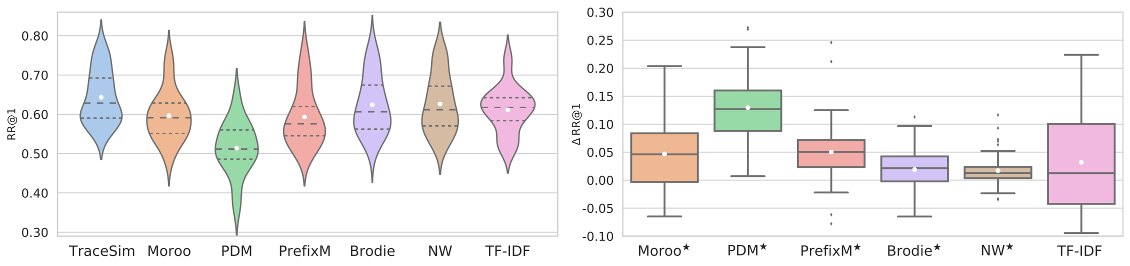
(a) Ubuntu



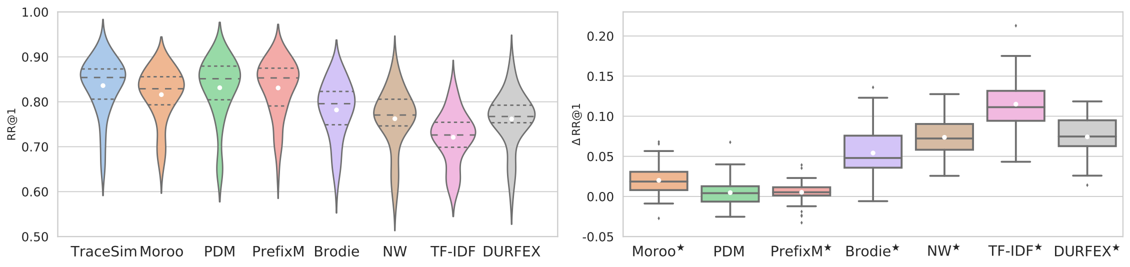
(b) Eclipse



(c) Netbeans



(d) Gnome



(e) JetBrains

★ $p < 0.01$

Figure 5.10 Results regarding RR@1. *Left*: distribution of RR@1 achieved by TraceSim and each competing method in all validation sets of each dataset. *Right*: ΔRR@1 between TraceSim and each competing method.

outperforms PDM in terms of AUC by 6.94% on average. In fact, none of the competing methods were able to outperform TraceSim across all metrics in a specific dataset. Finally, TraceSim is the only method that consistently performs well on different programming languages. For instance, in terms of MAP, there is no significant difference between PDM and TraceSim on Java datasets (Eclipse, Netbeans, and JetBrains). However, the improvement of TraceSim over PDM regarding MAP is, on average, 6.41% in Ubuntu and 11.99% in Gnome.

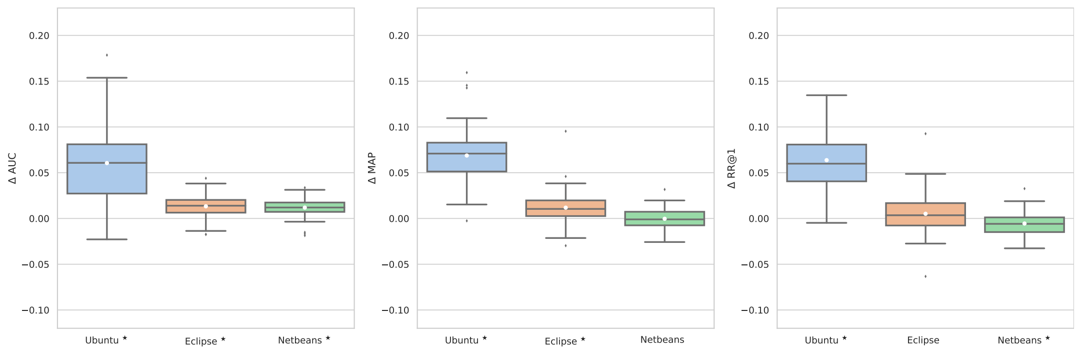
5.5.2 Ablation Study

An ablation study aims to assess specific model components by measuring performance degradation when each component is independently removed. In this section, we first conduct an ablation study to assess four important TraceSim components, namely global weight, local weight, the diff$(\cdot)$ function, and normalization. We then evaluate whether the approach to compute mismatch and gap values based on frame weights is more effective than previous strategies proposed in the literature. These two studies are performed only on Ubuntu, Eclipse, and Netbeans datasets due to the high computational cost of conducting such extensive experiments on Gnome and JetBrains.

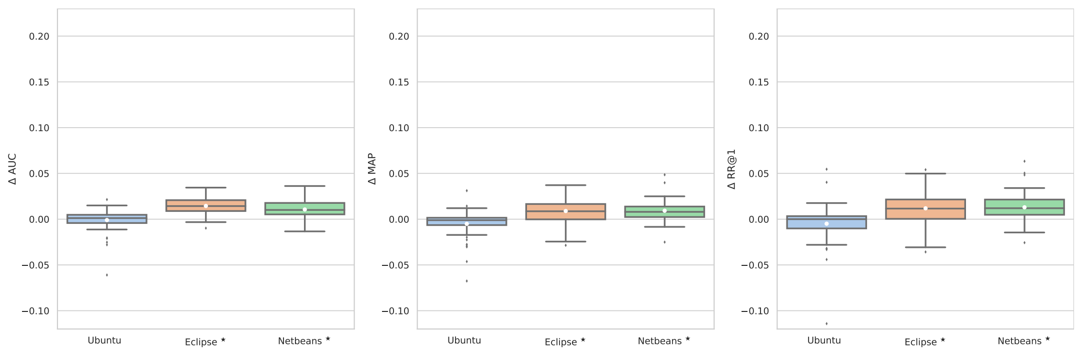
TraceSim Components

We evaluate four important TraceSim components:

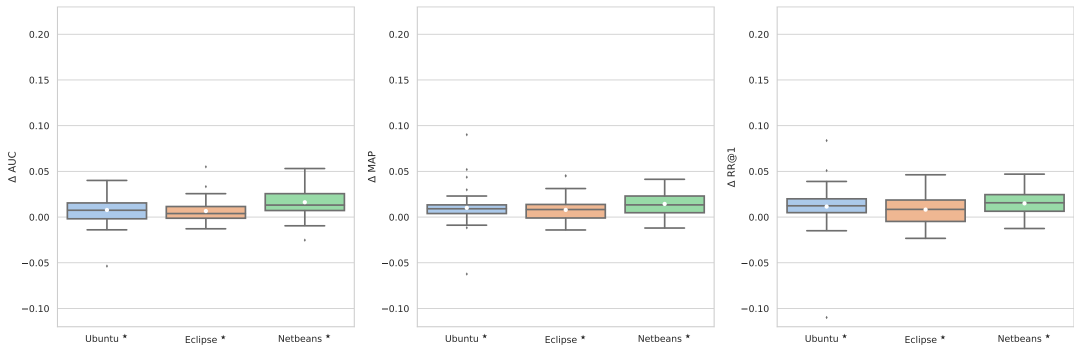
- *Global Weight.* In order to investigate the importance of TF-IDF for TraceSim, we ignore the global weight when computing the weight of a frame. This is achieved by setting $\mathrm{gw}(\cdot) = 1$ in Equation (5.4).
- *Local Weight.* Although frame positions are known to be valuable features for crash report deduplication, we measure the importance of this information for TraceSim. Thus, the local weight term is ignored by setting $\mathrm{lw}(\cdot) = 1$ in Equation (5.4).
- The diff$(\cdot)$ *function.* The difference between the positions of two matched frames is incorporated in PDM and Moroo. However, the corresponding papers do not include a study regarding the importance of this aspect. We ignore the diff$(\cdot)$ function in the match score function by setting $\mathrm{diff}(\cdot) = 1$ in Equation (5.9).
- *Normalization.* Many methods [22, 93, 94, 95] normalize similarity scores, although none of them have investigated the importance of this procedure.



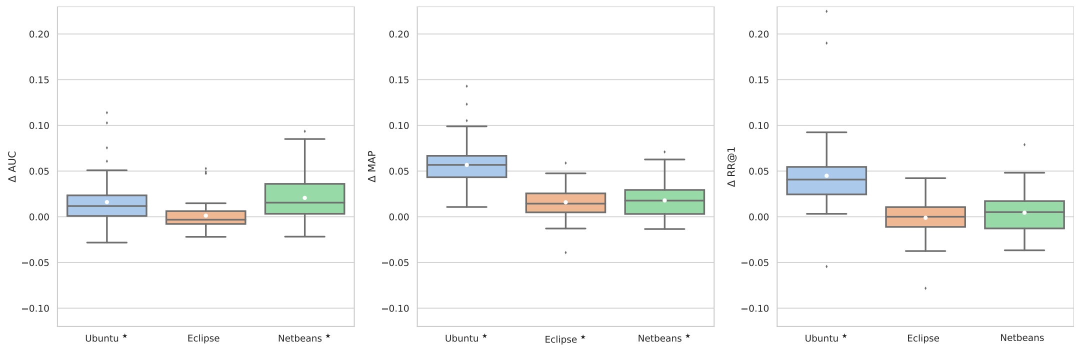
(a) Global Weight removed.



(b) Local Weight removed.



(c) The diff$(\cdot)$ function removed.



(d) Normalization removed.

$\bigstar\ p < 0.01$

Figure 5.11 Ablation study results: distributions of ΔAUC (left), ΔMAP (middle) and ΔRR@1 (right) between full TraceSim and TraceSim with different components removed.

In Figure 5.11, we depict performance differences (ΔAUC, ΔMAP, and ΔRR@1) between the full TraceSim and its modified versions for which we remove each component listed above.

As shown in Figure 5.11a, ignoring global weights significantly reduces performance in six out of the nine considered settings. The only three exceptions are RR@1 on Eclipse as well as RR@1 and MAP on NetBeans. Moreover, the only setting for which *Global Weight* deteriorates TraceSim's performance is RR@1 on NetBeans. These results corroborate the hypothesis that the global frequency of subroutines provide valuable information to discriminate important frames. In Figure 5.11b, we observe that TraceSim performs significantly worse on NetBeans and Eclipse when *Local Weight* is removed. On the Ubuntu dataset, this component appears to have no significant impact. As shown in Figure 5.11c, the position difference of matched frames significantly improves model performance in all datasets. This corroborates the hypothesis that duplicate stack traces contain important frames in close positions. Finally, in general, performance degrades significantly when normalization is not applied. As illustrated in Figure 5.11d, this component is not significantly relevant only in three settings: RR@1 on Netbeans, RR@1 on Eclipse, and AUC on Eclipse.

In Appendix C, we additionally report the performance differences between full TraceSim and each of the meaningful combinations that has at least two of four components removed. Overall, the findings are similar to the ones reported in this section.

Mismatch and Gap Values

In previous works, mismatch and gap values are defined as zero or some other constant. TraceSim, in constrast, defines mismatch and gap values based on frame weights, just as match values. The intuition is that the importance of unmatched frames should also be considered by the alignment algorithm. In order to compare the effectiveness of our strategy, we test all meaningful combinations in which mismatch and gap values are set by one of the following strategies:

- *Zero.* Values are set to zero, so that they have no cost in the optimal alignment.
- *Constant.* Values are constant real numbers tuned by the ML algorithm from the set $\{0.0, 0.1, \ldots, 6.0\}$. These predefined set of values achieved the best and consistent results.
- *Variable.* Gap and mismatch values are given by Equations (5.7) and (5.8), respectively. This corresponds to the proposed strategy used in TraceSim for both values: mismatch and gap.

There are nine possible combinations when considering the three aforementioned strategies to set mismatch and gap values individually. However, when Gap=Zero (i.e. gap values are set using the Zero strategy), we have that the mismatch operation is useless regardless of its value, since we can always replace a mismatch by two subsequent gaps with no cost in these cases. Thus, we have that ⟨Mismatch=Constant; Gap=Zero⟩ and ⟨Mismatch=Variable; Gap=Zero⟩ are both equivalent to ⟨Mismatch=Zero; Gap=Zero⟩. This leaves us with seven meaningful strategies. Thus, in the following, we analyse performance differences (ΔAUC, ΔMAP, and ΔRR@1) between the full TraceSim method ⟨Mismatch=Variable; Gap=Variable⟩ and the remaining six strategies.

In Table 5.7, we show whether ΔAUC, ΔMAP, and ΔRR@1 are statistically significant for each one of the six competing strategies in Ubuntu, Eclipse, and Netbeans (the datasets names are abbreviated to U, E, and N, respectively). A ★ symbol in a cell indicates that TraceSim significantly outperforms the strategy on the dataset and metric corresponding to that cell. It is important to highlight that no competing strategy achieves better average performance than TraceSim in these experiments.

Table 5.7 Performance differences between TraceSim and the six meaningful strategies to set mismatch and gap values which are statistically significant. Cells marked with a ★ indicate that the performance difference on the corresponding dataset and metric is statistically significant. Due to space constraints, we abbreviate Ubuntu, Eclipse, and NetBeans as U, E, and N, respectively, in the column labels.

Strategy		**Equiv**	**ΔAUC**			**ΔMAP**			**ΔRR@1**		
Mismatch	**Gap**		**U**	**E**	**N**	**U**	**E**	**N**	**U**	**E**	**N**
Zero	Zero	PDM	★	★	★	★			★		
Zero	Constant	Brodie	★	★	★	★	★		★		
Zero	Variable	–	★	★	★	★	★	★	★		
Constant	Constant	–	★	★	★	★	★		★		
Constant	Variable	–									
Variable	Constant	–	★	★	★	★	★		★	★	

First, we focus on the results of the three strategies in which mismatch values are defined as zero ⟨Mismatch=Zero; Gap=*⟩. These results correspond to the first three rows in Table 5.7. The ⟨Mismatch=Zero; Gap=Zero⟩ and ⟨Mismatch=Zero; Gap=Constant ⟩ strategies are equivalent to the strategies used in PDM and Brodie methods, respectively. Overall, the strategy in which mismatch is set to zero negatively affects the method performance. For instance, the ⟨Mismatch=Zero; Gap=Zero⟩ strategy, the best approach among the three ones, significantly degrades the performance in Ubuntu regarding all metrics and in Eclipse and Netbeans in terms of AUC. According to the results, we can conclude that the TraceSim strategy is more effective than the ones employed in Brodie and PDM techniques.

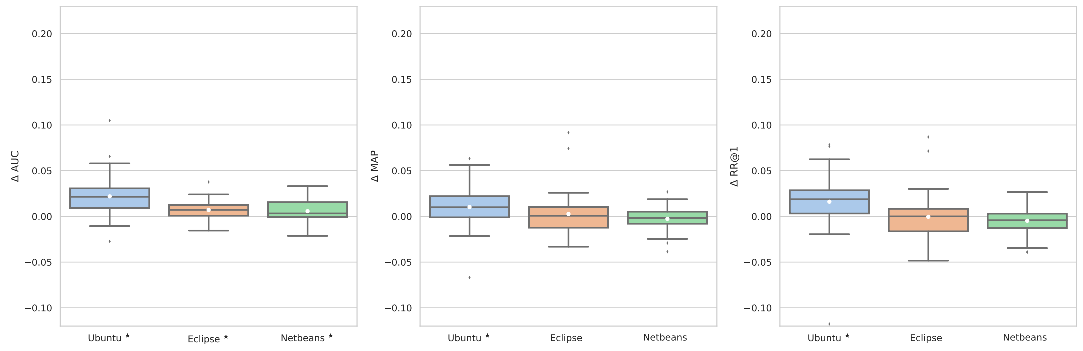
We now consider the results of the three remaining strategies in which mismatch values are set according to either Constant or Variable strategies. Both ⟨Mismatch=Constant; Gap=Constant⟩ (fourth row) and ⟨Mismatch=Variable; Gap=Constant⟩ (last row) strategies perform significantly worse in, respectively, six and seven out of the nine evaluation settings. On the other hand, the ⟨Mismatch=Constant; Gap=Variable⟩ strategy presents negligible effect on model performance, resulting in no significant difference in any evaluation setting. After analysing these results more carefully, we found that the constant mismatch value was set to high values in most of the cases by the TPE technique. More specifically, this value was equal to or greater than 2.0 in 126 out of 150 chunks, i.e. 84% of the cases. Since the upper bound of a gap value is 1.0 (see Equation (5.7)), a mismatch value equal to or greater than 2.0 means that the alignment algorithm will basically avoid mismatches. Recall that in TraceSim two subsequent gaps can always replace a mismatch with no effect in the optimal alignment cost. Thus, the ⟨Mismatch=Constant; Gap=Variable⟩ strategy, when using such high mismatch values, is equivalent to TraceSim. Overall, the results corroborate the hypothesis that gaps should be prioritized over mismatches, since gaps are more flexible. Moreover, we conclude that the proposed strategy to set mismatch and gap values based on frame importance is relevant to TraceSim's performance.

In addition to Table 5.7, for more details, we provide box-and-whiskers plots of the performance differences between TraceSim and the six strategies in Figure 5.12 and Figure 5.13.

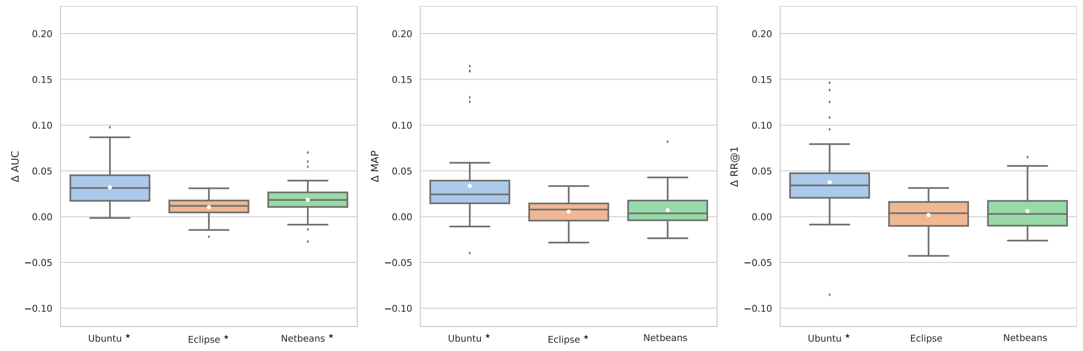
5.5.3 Time Efficiency

The comparison of a query to candidates within a dataset is the most critical efficiency issue in crash report deduplication. Due to the use of an inverted index, information retrieval methods are more efficient than matching algorithms to generate the ranked list of a query q. In one hand, the complexity of information retrieval methods with inverted index is $O(|qb| \log |V| d)$, where $|qb|$ is the set of subroutines in the query, $|V|$ is the vocabulary size, d is equal to the number of candidates in $C(q)$. In the other hand, the complexity of matching algorithms is $O(|q||c|_{max}|C(q)|)$, where $|q|$ is the query length, and $|c|_{max}$ is the longest candidate in $C(q)$. In real applications, d tends to be much smaller than $|C(q)|$ since information retrieval techniques only compute the similarity of candidates that contain subroutines shared with the query. This explains the considerable superiority of information retrieval techniques over matching algorithms in terms of efficiency.

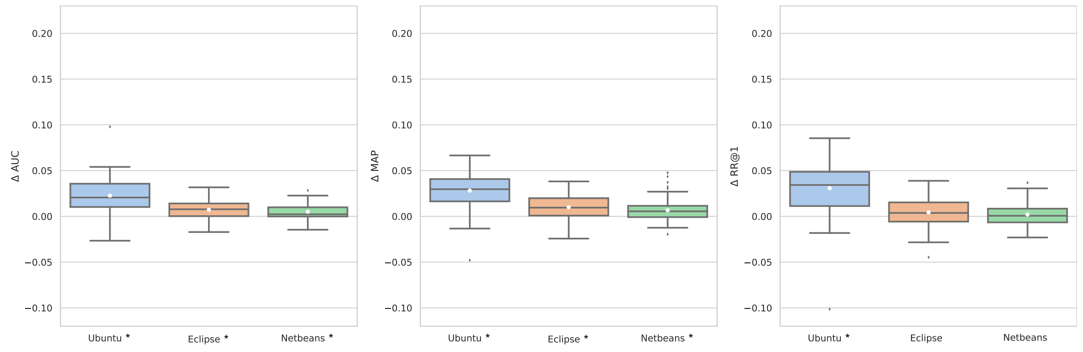
In practice, two simple approaches can be employed to speed up matching algorithms. The first one consists of using time windows to reduce the number of candidates to be considered



(a) Mismatch=Zero/Gap=Zero.



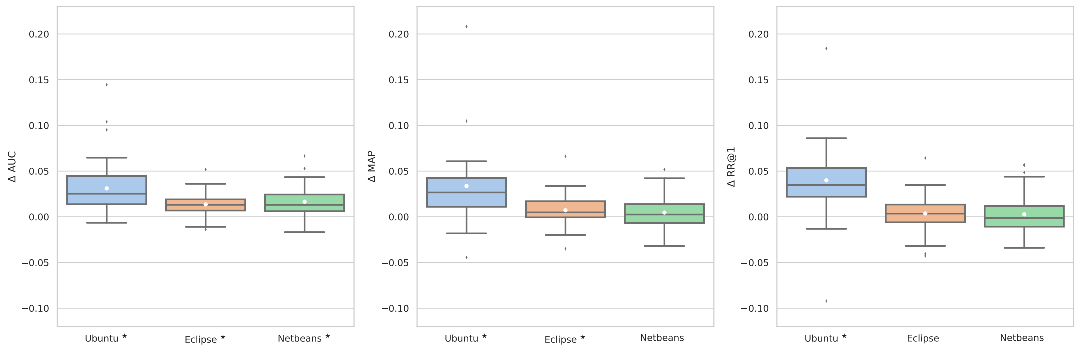
(b) Mismatch=Zero/Gap=Constant.



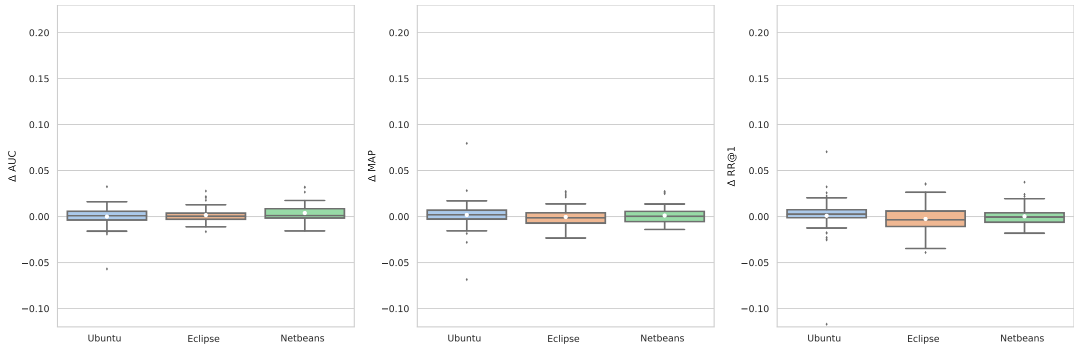
(c) Mismatch=Zero/Gap=Variable.

★ $p < 0.01$

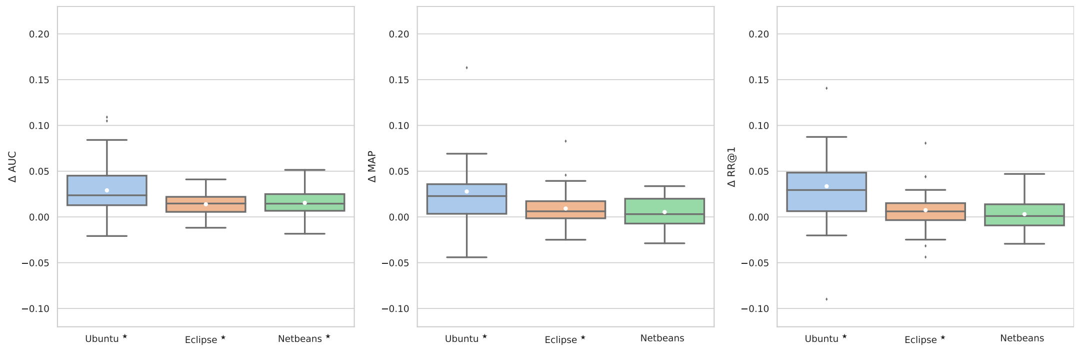
Figure 5.12 Distributions of ΔAUC (left), ΔMAP (middle) and ΔRR@1 (right) between complete TraceSim and three strategies in which mismatch values are defined as zero.



(a) Mismatch=Constant/Gap=Constant.



(b) Mismatch=Constant/Gap=Variable



(c) Mismatch=Variable/Gap=Constant.

$\star$ $p < 0.01$

Figure 5.13 Distributions of ΔAUC (left), ΔMAP (middle) and ΔRR@1 (right) between complete TraceSim and three strategies in which mismatch values are set according to either Constant or Variable strategies.

for deduplication. The second approach, called *re-ranking*, first creates a temporary ranked list based on information retrieval techniques. Then it recalculates the similarity score of the top-k candidates in the list using a more complex algorithm.

We examine the throughput of TraceSim (with and without time window), TF-IDF, and Moroo (a re-ranking technique) on the two largest datasets: Gnome and JetBrains. These techniques are executed on one of the original validation sets in a controlled and homogeneous environment[17]. Excluding the reports submitted after the validation sets, the experiment setups in Gnome and JetBrains contain 216,646 and 901,015 reports, respectively. As we can observe, the number of reports is very close to the total number of reports in the datasets (see Table 5.1).

In Table 5.8, we show the throughput (queries/second) of TraceSim with and without time window, TF-IDF, and Moroo in Gnome and JetBrains data sets. The time window approach improves the algorithm throughput by around 5.0 and 2.5 times in Gnome and JetBrains, respectively. However, even using a time frame, TF-IDF is substantially more efficient than TraceSim – the speedup is approximately 4.37 and 7.58 in Gnome and JetBrains. As expected, Moroo is slower than TF-IDF but the gap is much smaller in comparison to TraceSim – TF-IDF is 1.32 and 2.41 times faster than Moroo in Gnome and JetBrains.

Table 5.8 Throughput (queries / second) of TraceSim with and without time window, TF-IDF, and Moroo in Gnome and JetBrains.

Method	Gnome	JetBrains
TraceSim without Time window	0.5319	0.1080
TraceSim with Time window	2.6897	0.2569
TF-IDF	11.7687	1.9476
Mooro	8.8822	0.8051

In brief, it is difficult to determine whether the throughput of TraceSim presented in Table 5.8 is viable or not for a particular software project since a throughput of 2.62 or 0.25 may be satisfactory depending on the application. In cases that TraceSim is not viable due to its run time, besides calibrating the time window, re-ranking could be employed to accelerate it. However, more investigation is needed to figure out whether re-ranking would degrade the quality of deduplication.

[17]The experiments in Section 5.5.1 and 5.5.2 were run in a shared and heterogeneous environment. Therefore, it is difficult to compare the run times based on these experiments.

5.6 Threats to Validity

In this section, we discuss some threats to the validity of our study.

Quality of labeled data. Duplicate crash reports are identified by human triagers. Since this is not a trivial task, reports might be incorrectly classified as non-duplicate or inserted into an incorrect bucket. To mitigate this threat, we avoided using the most recent reports from the repositories. The assumption is that most incorrect labels tend to be corrected over time. Moreover, we employed data from well-known applications that have been used in literature for crash deduplication and duplicate bug report detection.

Subject selection bias. The performance of the considered methods is significantly dependent on data. Thus, the superiority of a method over other techniques might differ concerning other software projects. To mitigate this problem, we have conducted our experiments on five distinct software projects, which contain stack traces from different programming languages (C/C++ and Java). Moreover, four of our datasets come from different open-source applications while the other is an industrial dataset from IntelliJ Platform products.

Stack trace extraction. In the four open source datasets, we extracted the stack traces from the textual data of bug reports using different parsers for each programming language. Since textual data is unstructured, parsers might: extract only partial stack trace information, or miss stack traces (false negatives), or wrongly detect a fraction of texts as correct stack traces (false positives). We mitigate these possible issues by using parsers that are well-known in the literature by the community.

Competing method implementations. Except for the work of Campbell et al. [10], existing studies did not make available their implementations and/or the data used for experimentation. Hence, we had to implement all the baselines and state-of-the-art methods. Even though we have carefully followed the technique descriptions in the papers, our implementations might not fully match the originals since crucial components and preprocessing steps of the techniques might not have been described with complete accuracy, or even been reported in the study at all.

5.7 Conclusions

In this paper, we proposed TraceSim, a novel technique for crash report deduplication. TraceSim computes the similarity between a pair of stack traces by finding the optimal global alignment of their frame sequences. To compute the alignment score, we assign weights to frames that indicate their discriminativeness for stack trace comparison. These weights depend on two factors: 1) the position of the frames, and 2) the frequency of the subroutines in the dataset. The influence of these factors on the similarity is regulated by parameters that are learned using ML algorithms. Unlike previous techniques, the alignment scores are influenced by the weights of all frames, matched and unmatched, in the stack traces.

TraceSim and seven competing methods were experimentally evaluated on five datasets (four generated from open-source projects and one derived from industrial data) using a new methodology that combines ranking and binary classification metrics. Except for industrial project data, the full evaluation framework – including datasets and the source code of methods and evaluation methodology – is freely available online. We have found that TraceSim outperforms the majority of the existing methods in the literature. Moreover, our method performs consistently well in all distinct scenarios including datasets with distinct programming languages. In summary, compared to the previously proposed methods:

1. TraceSim distinguishes duplicate reports from non-duplicate ones more accurately.
2. TraceSim assigns reports to their correct buckets more often.
3. TraceSim generates better recommendation lists when the system needs human assistance.

Furthermore, we conducted an ablation study to investigate the effectiveness of TraceSim components and its scheme to compute mismatch and gap values. The results corroborated that the frame position (local weight) and document frequency (global weight) are valuable for crash report deduplication, as well as normalization and the use of the position difference of the matched frames. Finally, the experimental results confirmed the hypothesis that the rarity and the position of the frames should be considered for computing mismatch and gap values.

In terms of run time, TraceSim is similar to previously proposed sequence matching algorithms. However, due to the use of inverted index, information retrieval methods are more efficient in comparing a query to all n reports within a repository. Besides employing re-ranking, we plan to investigate additional approaches to reduce the computational cost without negatively affecting TraceSim's performance.

CHAPTER 6 ARTICLE 3: *FaST*: A LINEAR TIME STACK TRACE ALIGNMENT HEURISTIC FOR CRASH REPORT DEDUPLICATION

Authors. Irving Muller Rodrigues, Daniel Aloise, and Eraldo Rezende Fernandes. Accepted at Mining Software Repositories (*M*SR 2022)[1]

Abstract. In software projects, applications are often monitored by systems that automatically identify crashes, collect their information into reports, and submit them to developers. Especially in popular applications, such systems tend to generate a large number of crash reports in which a significant portion of them are duplicate. Due to this high submission volume, in practice, the crash report deduplication is supported by devising automatic systems whose efficiency is a critical constraint. In this paper, we focus on improving deduplication system throughput by speeding up the stack trace comparison. In contrast to the state-of-the-art techniques, we propose *FaST* , a novel sequence alignment method that computes the similarity score between two stack traces in linear time. Our method independently aligns identical frames in two stack traces by means of a simple alignment heuristic. We evaluate *FaST* and five competing methods on four datasets from open-source projects using ranking and binary metrics. Despite its simplicity, *FaST* consistently achieves state-of-the-art performance regarding all metrics considered. Moreover, our experiments confirm that *FaST* is substantially more efficient than methods based on optimal sequence alignment.

Keywords. Duplicate Crash Report, Crash Report Deduplication, Duplicate Crash Report Detection, Automatic Crash Reporting, Stack Trace

6.1 Introduction

To reduce user dependence in bug reporting and collect more data about errors, many software projects use *automated crash reporting systems* to monitor application executions. When target systems crash, such tools are invoked to gather relevant information about the failures and send it to backend systems [22].

The submitted information about a software error is grouped in a document called the *crash report*. A shortcoming of automated crash reporting systems is that they tend to rapidly increase the number of duplicate crash reports, that is, reports associated with the same failure.

[1]Available at [130]

Therefore, it becomes vital to automate deduplication when such tools are employed [131]. In the literature, such a task is denoted *crash report deduplication*, being also referred to as *duplicate crash report detection* or *crash report bucketing* [10].

In Figure 6.1, we depict an example of a crash report. Such documents may include the failure descriptions provided by users (**lines 1 and 13**) and environment information (**lines 5–12**). Additionally, crash reports contain stack traces (**lines 15–22**), one of the most valuable information source for bug fixing [8]. A stack trace is a sequence of *frames* in which the first frame corresponds to the topmost element in the application's call stack at the moment a crash occurs. The subsequent frames represent subroutines waiting for the execution of the previous frames near to the top. As shown in Figure 6.1, stack traces can contain multiple information about the frames (e.g., the source file name). Inspired by previous works [21, 22, 94, 118], this paper focuses on the positions and subroutine names of the frames. Moreover, to compare whether two frames are identical or not, we consider subroutine names as frame identifiers (shortly, *frame ids*).

```
 1  Crashed when I opened a website and clicked on 'flash version'.
 2
 3  Architecture: amd64
 4  Date: Tue Jun 19 20:16:04 2007
 5  DistroRelease: Ubuntu 7.10
 6  ExecutablePath: /usr/bin/gnash
 7  NonfreeKernelModules: vmnet vmmon cdrom
 8  Package: gnash 0.8.0~cvs20070611.1016-1ubuntu2
 9  PackageArchitecture: amd64
10  ProcCwd: /home/martin
11  SourcePackage: gnash
12  UserGroups: adm admin audio cdrom dialout dip floppy video
13  Title: gnash crashed with SIGSEGV in std::_Rb_tree::erase()
14
15  #0  0x027540aea in std::_Rb_tree::erase() at /usr/include/c++/4.1/bits/stl_tree.h:692
16  #1  0x027540b78 in std::_Rb_tree::erase() at /usr/include/c++/4.1/bits/stl_tree.h:1215
17  #2  0x02753f0c5 in movie_root::remove_key_listener() at /usr/include/c++/4.1/bits/stl_set.h:387
18  #3  0x02758100d in ~button_character_instance() at button_character_instance.cpp:280
19  #4  0x02753f0d3 in movie_root::remove_key_listener() at /usr/include/boost/intrusive_ptr.hpp:83
20  #5  0x02758100d in ~button_character_instance() at button_character_instance.cpp:280
21  #6  0x02753f0d3 in movie_root::remove_key_listener() at /usr/include/boost/intrusive_ptr.hpp:83
22  #7  0x02758100d in ~button_character_instance() at button_character_instance.cpp:280
```

Figure 6.1 Crash report example.

In the literature, a prevalent assumption is that crash reports are more likely to be duplicate when their stack traces are similar. Thus, the majority of techniques address crash report deduplication by comparing stack traces. For instance, *TraceSim* [118], one of the current state-of-the-art (SOTA) methods in crash report deduplication, measures the similarity of two stack traces by computing a weighted version of the optimal global alignment score [119] between them. In real environments, hundreds or even thousands of crash reports are submitted every day [10]. Each one of them must be analyzed by deduplication systems to identify whether they are duplicate in very large repositories of submitted reports. Hence, due to this high volume of data, deduplication systems must be implemented observing feasible throughput. A simple strategy to improve deduplication system performances is to speed up the similarity measurement of stack traces. In the literature of crash report deduplication, some methods can efficiently compare two stack traces in $O(m + n)$, where m and n are their lengths. However, such methods are significantly less effective than TraceSim, that computes the stack trace similarity score in $O(nm)$.

The inefficiency of methods based on optimal sequence alignment, including TraceSim, are mainly caused by their search for *optimal* alignments. In order to guarantee optimality, these methods iteratively compute a dynamic programming matrix using recursive functions. Furthermore, the found alignment must preserve the sequence order which makes challenging to independently compare subsets of subroutines in stack traces. Leveraging the removal of the optimality requirement and the order constraint, one can develop efficient heuristic algorithms that find near-optimal alignments. It is worthy to mention here that the final task objective does not consist in finding the optimal sequence alignment, but rather computing similarity scores that are effective to group duplicate reports.

Inspired by this idea, we propose *FaST*, a Fast Stack Trace alignment method for crash report deduplication. In *FaST*, the sequence alignment is produced by individually aligning the frames of each unique identifier in the stack traces. Since stack traces of duplicate reports are expected to contain subroutines in similar *absolute* positions [118], we argue that similarity scores can be fairly captured by directly comparing overlaps or missing frames of each individual subroutine. Instead of optimally aligning frames, we employ a simple alignment heuristic: given the frames of each distinct indentifier, *FaST* iteratively matches the two closest ones to the top positions. Such heuristic is based on the rationale that frames near the topmost position should be prioritized for alignment over those in the bottom, since they are usually more relevant for the deduplication [8, 22, 118]. In cases where frames of an identifier are only available in one of the stack traces, *FaST* aligns such remaining frames to special structures, called *gaps*. After finding the alignment, the similarity score is computed considering two important pieces of information regarding a subroutine: its position and its global frequency [118]. Due to its simplified alignment algorithm, *FaST* can compare stack traces in $O(n + m)$, i.e., linear time on the length of the two sequences.

We experimentally evaluate the efficiency and effectiveness of *FaST* by means of the methodology proposed by Rodrigues et al. [118]. We compare *FaST* with SOTA systems and strong baselines on four different datasets from the following open-source projects: Ubuntu, Eclipse, Netbeans, and Gnome. In our experiments, *FaST* consistently achieves similar or significantly superior performance in terms of effectiveness when compared with its competing methods. Moreover, as expected, we observe that *FaST* is considerable faster than optimal sequence alignment methods. It is important to highlight that we provide the source code of the evaluation framework and methods online[2]. The main contributions of this paper are summarized as follows:

1. We propose a novel stack trace alignment method with a linear time complexity for crash stack deduplication.
2. We show that a simple alignment heuristic can be as effective for deduplication as techniques that find optimal global alignments.
3. Our proposed method achieves state-of-the-art performances on all the considered datasets despite its simplicity.

6.2 Fast stack trace alignment method for crash report deduplication

As mentioned in the introduction, studies in the literature have addressed crash report deduplication by measuring stack trace similarity based on *optimal global alignment*. In the context of such problem, the term *global* means that frames are lined along the entire lengths of the compared stack traces.

In Figure 6.2, we depict an example of global alignment between two toy stack traces `adca` and `daccb`. As shown in the figure, there are three types of possible alignments: *match*, *mismatch*, and *gap*. A match occurs when two *identical* frames are aligned (e.g., the alignment between two frames `a`). Conversely, a mismatch arises when two *distinct* frames are aligned (e.g., frames `a` and `b` highlighted in red). The third type of alignment corresponds to lining up a frame to a *gap* (e.g., frames `d` and the cubes with dashed border). A gap represents an insertion/deletion operation performed in a sequence. A global alignment is only valid if the original sequence can be restored by removing the gaps, i.e., the sequence order cannot be altered.

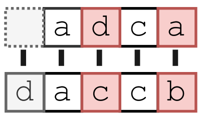


Figure 6.2 Example of a global alignment between the stack traces `adca` and `daccb`. Matches, mismatches and gaps are represented by white, red, and gray, respectively.

There are many distinct ways to align two stack traces end-to-end. In order to find the best global alignment, a scoring scheme is defined to assign a value to each element alignment.

[2]`https://github.com/irving-muller/FaST`

Following such scheme, the score of the entire sequence alignment is equal to the sum of matches values subtracted by the values of mismatch and gap alignments. The *optimal global alignment* consists in finding an alignment between two sequences for which the score alignment is maximum. Given two sequences with lengths n and m, the optimal alignment can be found in $O(nm)$ time with the Needleman–Wunsch (NW) algorithm [119].

In this section, we present *FaST*, a novel sequence alignment method that effectively aligns two stack traces in $O(n + m)$ time. To achieve such complexity, *FaST* relaxes the optimal global alignment problem, allowing the computation of sub-optimal and non-ordered alignments.

6.2.1 Similarity Algorithm

Similar to the optimal global alignment problem, *FaST* compares two stack traces based on the overlaps (represented by matches) and differences (captured by mismatches and gaps) between them. First, matches are performed by successively lining up the topmost unaligned frames with the same id. Each match increases the similarity score based on two factors: frame importance and the position discrepancy between the matched frames. The former depends on position and subroutine global frequency (two key frame features) and the latter alleviates the impact of poor matches performed by the heuristic alignment. After performing match alignments, all unmatched frames are aligned to gaps due to empirical evidences that gaps are more adequate than mismatches for crash report deduplication [118]. Each gap alignment penalizes the similarity score also based on the frame importance. Following previous works [22, 95, 118], the similarity score is normalized to be in range [-1.0, 1.0].

In Algorithm 1, we present the pseudo-code of *FaST* algorithm to compute the similarity score for two stack traces. The functions match($\cdot$) and gap($\cdot$) compute the values of a match and gap alignments, respectively, while w($\cdot$) returns a real number, called *frame weight*, that indicates the frame importance for the deduplication. Further details about such functions are provided in Section 6.2.2. In Figure 6.3, we depict an example of this algorithm execution on two toy stack traces.

As input, the similarity algorithm receives two lists sorted by frame id and position in ascending order. After sorting, each frame in Q and C are represented as f_p where f is its id and p is its position in the original stack trace. Basically, the sorting operation is analogous to split the frames in the stack trace and group them by their identifiers. It is worthy mentioning that such sort is executed only once right after stack trace creation. Thus, we consider that its time complexity is amortized given: (i) the relative short lengths of stack traces[3]; and

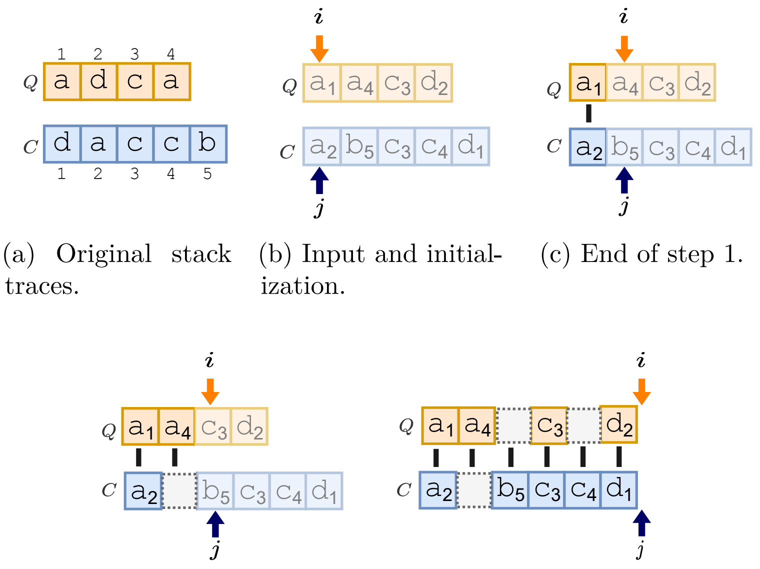
[3]In our datasets, 98% of the stack traces are shorter than 126 subroutines.

Algorithm 1: *FaST* pseudo-code

```
Input: $Q$ and $C$: lists of frames of two stack traces that are sorted by frame ids and
       positions.
Output: Normalized similarity between $Q$ and $C$.
1  $sim \leftarrow 0.0$
2  $i \leftarrow 1$
3  $j \leftarrow 1$
4  while $i <$ length($Q$) and $j <$ length($C$) do
       // $q$ and $u$ are the id and position of $Q[i]$
5      $q_u \leftarrow Q[i]$
       // $c$ and $v$ are the id and position of $C[j]$
6      $c_v \leftarrow C[j]$
7      if $q == c$ then
           // Match alignment.
8          $sim$ += match($q_u, c_v$)
9          $i$ += 1
10         $j$ += 1
11     else if $q < c$ then
           // $Q[i]$ is aligned to a gap.
12         $sim$ −= gap($q_u$)
13         $i$ += 1
14     else
           // $C[j]$ is aligned to a gap.
15         $sim$ −= gap($c_v$)
16         $j$ += 1
   // Align remaining frames in $Q$ or $C$ to gaps
17 while $i <$ length($Q$) do
18     $sim$ −= gap($Q[i]$)
19     $i$ += 1
20 while $j <$ length($C$) do
21     $sim$ −= gap($C[j]$)
22     $j$ += 1
   // Normalize the similarity score
23 return $\frac{sim}{\sum_{q_u \in Q} \mathrm{w}(q_u) + \sum_{c_v \in C} \mathrm{w}(c_v)}$
```

(ii) the amount of comparisons performed to a given stack trace in real applications of crash report deduplication is much larger than $O(n \log n)$.

During initialization, the algorithm creates two pointers that refer to the beginning of each list. Each pointer represents the next available frame in a list for the alignment. In the example presented in Figure 6.3b, the pointers are illustrated by i and j and they point to



(a) Original stack traces. (b) Input and initialization. (c) End of step 1. (d) End of step 2. (e) Final alignment.

Figure 6.3 An example of *FaST*'s alignment algorithm.

the first frames of the sorted list Q and C, respectively. As first step, *FaST* compares the frame ids pointed by i and j, i.e., the first elements within Q and C. Since the identifiers are the same, a_1 and a_2 are matched. Then, i and j are moved to the next elements in the lists – a_4 and b_5, respectively. The result of this procedure is depicted in Figure 6.3c. In the next step, i and j refers to different subroutines. Since frames are sorted by the subroutine identifiers and a is smaller than b, this means there is no other available frame with id $= a$ in the list C. Therefore, as illustrated in Figure 6.3d, a_4 is aligned to a gap and i is jumped to the subsequent available frame in Q. The algorithm proceeds by sequentially comparing the frames pointed by i and j. Match alignment is performed when frames share the same ids, otherwise the frame with the smallest identifier is aligned to a gap. The pointers are moved to the next available element in the sorted lists when the pointed frames are aligned. If the algorithm reaches the end of a list, then the remaining frames within the other one are aligned to gaps. Regarding our example, we depict the final alignment between Q and C found by such algorithm in Figure 6.3e.

A limitation of *FaST*'s algorithm is that it does not directly penalize order inversions. As depicted in Figure 6.3e, even though the two first frames in Q and C are in inverse order, *FaST* still performs two matches between these frames. On the other hand, as shown in Figure 6.2, the NW algorithm penalizes this inversion by only matching the frames `a` and performing a mismatch and a gap alignment regarding the frames `d`. Nevertheless, in *FaST*, the higher is the position difference between two matched frames, the lower is their match value (further details in Section 6.2.2). Thus, if two frames have their relative order inverted, at least one of them will be in different positions in the two sequences, and this will be penalized by our method. Furthermore, order inversions are not usual in stack traces, since they imply indirect recursions.

As mentioned earlier, the score of the final alignment is computed based on the chosen scoring scheme – functions match(·), gap(·), and w(·) – and it is normalized to be in a fixed interval – **line 23** in Algorithm 1. These algorithm aspects are described in details in the remainder of this section.

6.2.2 Scoring Scheme

In *FaST*, values of each match and gap alignments are computed using a scoring scheme similar to the one proposed by Rodrigues et al. [118]. In this scoring scheme, weights are assigned to each frame in the stack traces. A weight captures the importance of a frame f_p for the deduplication and it is computed as follows:

$$\mathrm{w}(f_p) = \frac{1}{p^\alpha} \times e^{-\beta \frac{\mathrm{df}(f)}{|S|}}, \tag{6.1}$$

where $|S|$ is the total number of stack traces in a repository S, and $\mathrm{df}(\cdot)$ is the number of stack traces in S that contain at least a subroutine identifier equal to f. The first component of (6.1) assigns higher values to positions closer to the top since such positions tend to be more related to the failure. The second one depends on the rarity of an id among the stack traces in the dataset. Frequent subroutines are usually ordinary operations in a system (e.g, logging and error-handling) and, thus, they are likely unrelated to the crash cause. Therefore, the more frequent the id of a frame is, the lower its weight should be. In Equation 6.1, similar to a logical *AND*, these two components are multiplied to consider a frame irrelevant for the deduplication when either its position is close to the bottom or its subroutine is frequent. Finally, $\alpha \in \mathbb{R}_{>0}$ and $\beta \in \mathbb{R}_{>0}$ are parameters that control the impact of the frame position and subroutine rarity on the weight value, respectively.

Following the original scheme, the gap alignment value is equal to the weight of a frame f_p aligned to a gap:

$$\mathrm{gap}(f_p) = \mathrm{w}(f_p). \tag{6.2}$$

However, unlike Rodrigues et al. [118] that employ the maximum weight between two matched frames q_u and c_v, we calculate the match value by means of the sum of these weights:

$$\mathrm{match}(q_u, c_v) = (\mathrm{w}(q_u) + \mathrm{w}(c_v)) \times \mathrm{diff}(u, v), \tag{6.3}$$

where the function diff$(\cdot)$ is defined as:

$$\text{diff}(u, v) = e^{-\gamma|u-v|}. \qquad (6.4)$$

The parameter $\gamma \in \mathbb{R}_{>0}$ regulates the impact of the position difference on the function output. Since a common assumption is that same subroutines appear in closer positions in stack traces of duplicate crash reports, diff$(\cdot)$ reduces the match value based on the position discrepancy of the matched frames.

6.2.3 Normalization

After aligning the stack traces, the alignment score is computed as the sum of the match values minus the sum of each gap alignment value. However, such score is not directly used for the deduplication since it can degrade the method effectiveness [118]. For instance, three stack traces $ST1$, $ST2$, and $ST3$ are depicted in Figure 6.4. Frame weights are represented by the real numbers below each subroutine identifier. In this example, the alignment scores are -1.7 and -2.2 when $ST1$ is compared with $ST2$ and $ST3$, respectively. However, this is unreasonable because $ST1$ is completely different of $ST2$ while the two topmost frames of $ST1$ and $ST3$ are overlapped. Such contradictory scores occurs because the alignment score is highly dependent on the frame weight values. Therefore, in order to mitigate such issue, the similarity scores are normalized based on the frame weights [118, 22].

Considering the definitions of gap and match (Equations 6.2 and 6.3), we can normalize the similarity score to be within the interval $[-1.0, 1.0]$ by simply dividing the alignment score by the sum of frame weights in the two stack traces (**line 23** in Algorithm 1). For instance, the sum of weights are 1.3, 0.4, and 3.7 in $ST1$, $ST2$, and $ST3$, respectively. Thus, the similarity score by comparing $S1$ with $S2$ and $S3$ is $\frac{-1.7}{1.3+0.4} = -1.0$ and $\frac{-2.2}{1.3+3.7} = -0.44$, respectively.

$ST1$	a	b	a	a	c
	0.4	0.3	0.2	0.1	0.3

$ST2$	d	d	e
	0.2	0.1	0.1

$ST3$	a	b	f	f	g	h
	0.4	0.3	0.9	0.8	0.8	0.5

Figure 6.4 Normalization example.

6.3 Related works

In this section, we focus on studies that address crash report deduplication by means of stack trace similarity.

Modani et al. [95] proposed a *prefix match algorithm* in which the similarity is proportional to the length of the longest common prefix between two stack traces.

Methods based on the popular TF-IDF approach (Term Frequency – Inverse Document Frequency) [10, 93, 94] have also been applied to crash report deduplication. Lerch and Mezini [93] and Campbell et al. [10] employed the TF-IDF-based score function from Lucene library[4] to measure the stack trace similarity. Sabor et al. [94] proposed the *DURFEX* technique, which uses only the package name of the subroutines, to compare two stack traces using the cosine similarity of their vector representations. One important drawback of these techniques is that they ignore a valuable piece of information: the position of the frames within the stack trace.

Some studies [21, 22, 118] proposed variantions of the NW algorithm to measure the similarity between two stack traces. In the technique designed by Brodie et al. [21], while mismatch and gap alignment values are constant, the match values are computed based on the rarity and position of the matched subroutines. On the other hand, Dang et al. [22] proposed a method, called PDM, in which match values depend only on the frame positions and the alignment score is not penalized by mismatches and gap alignments. Moreover, PDM contains parameters that regulate the impact of position information on the optimal solution. More recently, Rodrigues et al. [118] proposed TraceSim, a method for crash report deduplication that have outperformed previous methods from the literature. TraceSim computes match, mismatch, and gap values based on both the position and the global frequency (considering a large database) of a subroutine. To improve method flexibility over different data distributions, parameters control the weight of these two factors on the final similarity.

In order to improve efficiency without degrading the effectiveness of methods based on NW algorithm, Moroo et al. [102] proposed a reraking model, called PartyCrasher, that combines information retrieval techniques and PDM. First, PartyCrasher selects the top-k most similar candidates to a query by means of the score function designed by Lucene. Then, PDM is employed to compute the similarity between the selected candidates and the query. Finally, the final similarity score is a weighted harmonic mean of the similarities measured by Lucene's score function and PDM.

Edit distance is equivalent to optimal global alignment [124] and have been employed by two

[4]`https://lucene.apache.org/`

studies for crash report deduplication. Bartz et al. [96] proposed a logistic regression model based on the edit distance between two stack traces and some categorical features within the crash reports. To compute the edit distance, Bartz et al. [96] use the following information regarding a frame in the stack trace: the subroutine, its offset, and its module. Dhaliwal et al. [13] proposed to group the crash reports by the subroutine in the topmost position. Each group is then reorganized in subgroups based on the edit distance between its stack traces. These two techniques have the same efficiency issues that are present in techniques based on global alignment.

Khvorov et al. [131] proposed a siamese deep learning model, called S3M, for comparing two stack traces. A Long Short-Term Memory (LSTM) independently encodes the stack traces as fixed-length vectors. Then, a multilayer perceptron (MLP) computes the similarity between two stack traces based on their vector representations.

In Table 6.1, we present the time complexities of techniques to compute the similarity of two stack traces. Prefix match and methods based on TF-IDF run in linear time. However, they are less effective than more computationally expensive methods. Prefix match is highly affected by negligible differences in subroutine positions, while TF-IDF techniques ignore positional information altogether. Regarding S3M, considering that representations of stack traces are computed once and stored in a database, the amortized time complexity of such model is $O(dh)$, where d and h are the sizes of the input and hidden layer, respectively. In practice, d and h are comparable or even larger than the stack trace lengths, e.g., $i = 600$ and $d = 300$ in S3M while we found that, in our experimental setup, 98% of the stack trace contains less than 130 frames. Besides its quadratic complexity, other limitation of S3M is that it requires a considerable volume of labeled data for training. However, such type of data is not always available in industry projects. Unlike S3M, the parameters of *FaST* can be manually set by a specialist.

In order to effectively address crash report deduplication, *FaST* leverages the empirical findings observed in TraceSim's study [118], e.g., frame position, subroutine global frequency, normalization and function diff$(\cdot)$ are crucial for this task. On the other hand, *FaST* finds sub-optimal alignments in linear time complexity that are as effective as the optimal ones found by TraceSim in quadratic time (more details in Section 6.5). Moreover, our method uses a different function match$(\cdot)$ that is based on the sum of the frame weights instead of the maximum value between them. Such function allows to simplify the normalization: whereas TraceSim's ones is inspired by the weighted Jaccard index, *FaST* 's normalization divides the

[5]This corresponds to the multilayer perceptron complexity time. Such component contains one hidden layer of size h and receives an input of size d.

Table 6.1 Time complexity of crash report deduplication methods. The lengths of two stack traces is denoted n and m.

Method	**Time complexity**
Prefix Match [95]	$O(max(n, m))$
Lerch and Mezini [93]	$O(n + m)$
Campbell et al. [10]	$O(n + m)$
DURFEX [94]	$O(n + m)$
Brodie et al. [21]	$O(nm)$
PDM [22]	$O(nm)$
TraceSim [118]	$O(nm)$
Bartz et al. [96]	$O(nm)$
S3M [131][5]	$O(dh)$

alignment score by the sum of all frame weights in the stack traces.

Regarding the literature of sequence alignment works, the majority of them come from bioinformatics field. Thus, several heuristics for this problem make use of specific aspects of this domain to speed up algorithms [132]. Although, few optimal sequence alignment techniques were proposed besides the NW algorithm, they still run in $O(nm)$, being their superiority restricted to bioinformatics instances [133]. Overall, due to the particular characteristics of bioinformatics field, the proposed methods and their findings are not applied to the crash report deduplication task.

6.4 Experimental setup

In this section, we present the main components of our experimental setup: datasets, evaluation methodology, evaluation metrics, and competing methods. The developed code – including the evaluation framework and implementation of *FaST* and competing methods – are available online[6].

6.4.1 Datasets

The datasets published by Rodrigues et al. [118] are employed in our experiments[7]. Due to scarcity of publicly labeled data, a common practice in the literature is to generate crash reports by extracting stack traces from bug reports. Thus, such datasets were created by parsing bug reports from bug tracking systems (BTS) of four open source projects:

[6]`https://github.com/irving-muller/FaST`

[7]The dataset is available on `https://zenodo.org/record/5746044#.YeDFCNtyZH5`

Ubuntu [134], Eclipse [135], Netbeans [136], and Gnome [137]. Ubuntu's and Gnome's repositories are composed of issues from different applications for Ubuntu Linux distribution and Gnome desktop environment, respectively. Most of these applications are developed in C/C++. Eclipse and Netbeans are two popular Integrated Development Environments (IDEs) implemented in Java. Statistics of these datasets are presented in Table 6.2.

Table 6.2 Statistics of datasets.

Dataset	**Period**	**# Duplicates**	**# Reports**	**# Buckets**
Ubuntu	25/05/07 - 18/10/15	11,468	15,293	3,825
Eclipse	11/10/01 - 31/12/18	8,332	55,968	47,636
Netbeans	25/09/98 - 31/12/16	13,703	65,417	51,714
Gnome	02/01/98 - 31/12/11	117,216	218,160	100,944

We perform two extra preprocessing steps in addition to the ones applied in [118]. In the provided datasets, a crash report can contain multiple stack traces. Rodrigues et al. [118] decided to include all identified stack traces found in the description and attached files of the original bug reports due to different reasons, e.g., the difficulty to determine which subroutine caused the failure, parsing limitations, among others. However, we observed that a significant portion of the stack traces in crash reports are, in fact, the top-k frames of other stack traces in the same reports. In order to improve the readability, testers and developers may only provide the first frames of a stack trace in the description of the bug report. The full content is attached to the report as a file. Thus, to remove this duplicate data, we identify the longest stack trace ST^l in a crash report r and, then, the remaining stack traces in r are filtered when they are a prefix of ST^l. Moreover, specifically for Gnome, we applied the same procedure used in the BTS to identify the "interesting stack traces" of multi-thread systems[8]. In a nutshell, such procedure consists in keeping or removing stack traces based on a list of relevant subroutine names (e.g., `signal` and `segv`). In Table 6.3, we present the number of crash reports with more than one stack trace found in the datasets before and after our preprocessing.

6.4.2 Evaluation Methodology

In this work, in order to assess different methods, we employ the comprehensive evaluation methodology proposed in [118]. This methodology uses a dataset D composed of crash reports

[8]The original code can be found in the function `interesting_threads` in the following file: `https://bazaar.launchpad.net/~bgo-maintainers/bugzilla-traceparser/3.4/view/head:/lib/TraceParser/Trace.pm`

Table 6.3 Percentage of reports with multiple stack traces in each dataset before and after the preprocessing.

BTS	**Before**	**After**
Ubuntu	0.03%	0.03%
Eclipse	24.37%	23.75%
Netbeans	61.37%	28.93%
Gnome	80.14%	9.52%

sorted by their creation date. Then, a *query set* Q is generated by randomly selecting a sequence of consecutive reports in D. In order to assess a similarity-based deduplication method, each query report $q \in Q$ is considered as a newly submitted report, and the method is used to compute the similarity between q and older reports in D (reports submitted before q).

As mentioned before, crash reports in the dataset are grouped into buckets. A bucket is the set of all reports associated to the same software bug and is denoted as B_r, where r is the first submitted (oldest) report in B_r. In the used evaluation methodology, when a query report q is considered, buckets for reports submitted before q are known. In that way, the evaluation is based on similarities between the query report q and the buckets, instead of individual reports. The similarity between q and a bucket B is defined as:

$$\text{sim}'(q, B) = \max_{c \in B} \text{sim}(q, c),$$

where $\text{sim}(q, c)$ is the similarity between the query report q and a candidate report c of B calculated by the system being evaluated.

Since crash report datasets can be very large, in order to improve system's efficiency, the deduplication of a query report q is restricted to a subset of *candidate buckets* denoted as $C^B(q)$. This set comprises only buckets that include at least one report submitted within a time window of two years before q. All reports in such selected buckets are considered as candidates, including those submitted outside of the time window. Therefore, when deduplicating q, buckets are *unreachable* if all theirs reports are submitted more than two years before q. Reports created after q are always ignored, as mentioned above.

For a more detailed and comprehensive explanation of the methodology, we refer the reader to Rodrigues et al. [118].

6.4.3 Evaluation Metrics

Considering a query set Q, the methodology evaluates a method by means of three metrics: Mean of Average Precision (MAP), Recall Rate@k (RR@k), and Area Under the ROC Curve (AUC). MAP and RR@k are ranking metrics, i.e., they assess the quality of ranked lists generated for each query based on the similarity technique. In this methodology, a ranked list, denoted as $L(q)$, consists of the candidate buckets for a query report q sorted by their similarity to q in ascending order. Moreover, the ranking metrics are not measured for queries related to crashes that have never been reported before. Such non-duplicate reports, called singletons, are ignored in this ranking evaluation since their respective ranked lists do not contain their correct bucket (the relevant candidate). We denote $Q^d \subset Q$ the subset of non-singleton queries.

Considering $p_{L(q)}$ as the position of the correct bucket of a query q within a ranked list $L(q)$, RR@kk is computed as follows:

$$\text{RR@}k = \frac{\sum_{q \in Q^d} \mathbb{1}[p_{L(q)} \leq k]}{|Q^d|}, \tag{6.5}$$

where $k \in \mathbb{N}^+$ and $\mathbb{1}[p_L \leq k]$ returns 1 if the position of the correct bucket is in the top-k positions of a ranked list, and 0 otherwise. In summary, RR@kk is the fraction of queries in Q^d whose correct buckets appear in the first k positions of the ranked lists. Particularly, in realistic scenarios where reports are automatically assigned to the most similar buckets, RR@k1 represents the system accuracy for assigning duplicate reports to buckets.

Unlike RR@kk, MAP can summarize the ranked list quality by means of a single real value. In this methodology setting, since there is one relevant item within a ranked list, MAP can be simplified as:

$$\text{MAP} = \frac{1}{|Q^d|} \sum_{q \in Q^d} \frac{1}{p_{L(q)}}. \tag{6.6}$$

MAP values range in the interval $[0, 1]$ where MAP $= 1$ when the correct bucket is among the first elements in all ranked lists.

In real systems, it is important to distinguish a duplicate report from a singleton since a significant portion of the queries are non-duplicate reports. Thus, in order to consider such aspect of the deduplication task, this methodology also cast this task as a binary classification. In this case, a query is classified as duplicate when the highest similarity score between the query and its candidate buckets is greater than a given threshold t. For an evaluation that is independent of the threshold value, the classification performance is measured by the well-known area under the ROC curve (AUC) metric [138]. The ROC (receiving operating characteristic) curve is the plot of true positive rate versus the false positive rate for all possible values of t. The AUC metric derives a single real number from the ROC curve.

6.4.4 Parameter Tuning and Model Validation

Following [118], evaluation is performed using two subsequent, but disjoint, query sets: a tuning set T and a validation set V. The query set V is composed of reports submitted during a (randomly selected) period of one year, and the query set T comprises the last 250 reports submitted immediately before V. Additionally, P is denoted as the set of reports submitted earlier than T. In the experiments, the parameters are first tuned on T by means of a Tree-structured Parzen Estimator (TPE) [122]. Given a maximum number of iterations[9], such optimizer tries to search for parameter values that maximize the sum of MAP and AUC scores on the tuning set. Finally, using the best parameters found, we evaluate the method effectiveness on the corresponding validation set V.

Since data distribution tends to significantly change during the repository lifetime, the performance of the same method can highly vary depending on the data period used in the evaluation [111]. In order to better capture the method effectiveness along the whole repository, 50 validation sets (periods of one year) are randomly selected in each dataset. Thus, the tuning sets are generated based on each sampled validation set. In Figure 6.5, we illustrate an example where three random validation sets are sampled.

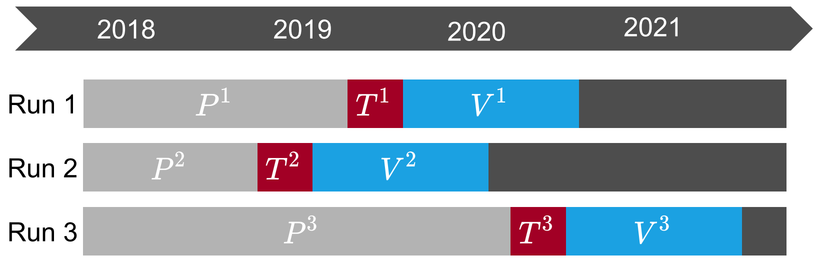


Figure 6.5 Three validation sets (along with the corresponding tuning sets) sampled from a dataset. The validation set, tuning set, and P in run k is represented as V^k, T^k, and P^k, respectively.

6.4.5 Competing Methods

In order to empirically demonstrate the effectiveness and efficiency of the proposed alignment heuristic, we compare *FaST* to two optimal sequence alignment methods: *TraceSim* and *PDM*.

[9]TPE is run in 100 iterations.

TraceSim significantly outperformed sequence matching and information retrieval methods for the majority of metrics (AUC, MAP, and RR@k) and datasets. On the other hand, PDM was the only method to surpass TraceSim's performance in one specific scenario and it can be better optimized than others NW algorithm variants [21, 118]. Finally, our method is also compared to a modified version of TraceSim, called *TSM*, whose match($\cdot$) and normalization are equivalent to *FaST*'s ones. Our objective is to investigate whether the proposed match($\cdot$) and normalization significantly impact the method effectiveness.

Additionally, *FaST* is compared to Prefix Match, TF-IDF, and DURFEX due to their relatively low time complexity. For simplicity, the first method is abbreviated to *PrefixM*. To guarantee a fair comparison in the experiments, we implement TF-IDF in our evaluation framework following Lucene's implementation and we only consider the subroutine names and positions within stack traces for the crash report deduplication. The subroutine names are the fully qualified method names in Java's stack traces while function names in C++'s ones. Finally, we only evaluate DURFEX on Eclipse and Netbeans datasets, since it was designed for the Java language.

6.5 Experimental Results

In this section, we study the effectiveness and efficiency of *FaST* in comparison with four competing methods on Ubuntu, Eclipse, Netbeans, and Gnome datasets. For each method, we report throughput values (queries/second) and their distributions over the 50 validation sets by means of box plots combined with scatter plots. Moreover, we measure the speedup between *FaST* and a competing technique in terms of throughput on each validation set. Then, we depict the distribution of the obtained speedup values over the validation sets using box plots. It is worthy to mention here that speedup between two methods in dataset depends only how fast each method compares two stack traces. Thus, other variables related to the dataset (e.g., the number of reports) do not affect such measurement.

Furthermore, violin plots [127] are used to present the distribution values of AUC, MAP, and RR@1 achieved by each method over the validation sets. Such plots are generated by means of seaborn library and they contains three dashed lines to represent the 25th, the 50th, and the 75th percentiles. Additionally, we calculate the performance differences between *FaST* and each competitor regarding AUC, MAP, and RR@1 in each validation set. The differences of such metrics are denoted ΔAUC, ΔMAP, and ΔRR@1, respectively. These differences are positive whenever *FaST* outperforms its competitor. We report the distributions of ΔAUC, ΔMAP, and ΔRR@1 in the 50 validation sets using box plots. Throughout this section, the mean values are depicted as white circles in the plots.

In order to assess whether a method superiority is statistically significant, we apply the Wilcoxon signed-rank test to ΔAUC, ΔMAP, and ΔRR@1 as follows:

H_0: The two methods yield the same performance.

H_1: The two methods yield different performances.

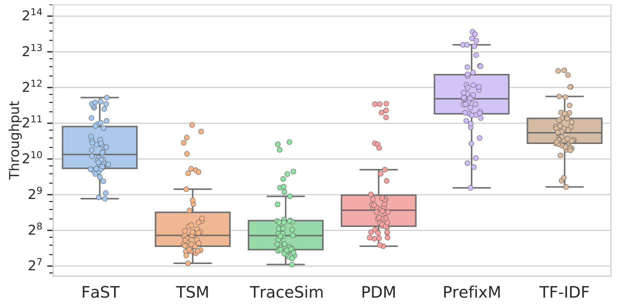
We accept the alternative hypothesis (H_1), consequently rejecting the null hypothesis (H_0), when $p < 0.01$. In the plots, the symbol ★ next to the name of a method indicates that the performance difference to *FaST* is statistically significant.

In Figure 6.6 (left), we depict the throughput of *FaST* and competitive methods on Ubuntu, Eclipse, Netbeans, and Gnome. At the right of this figure, we report the distribution of speedups between *FaST* and its competing methods on the tested datasets. Additionally, in Figure 6.7, we depict the differences ΔAUC, ΔMAP, and ΔRR@1 between FaST and competitors on Ubuntu, Eclipse, Netbeans, and Gnome. Complementary, for each dataset, the distributions of the absolute metric values are depicted in Figure 6.8.

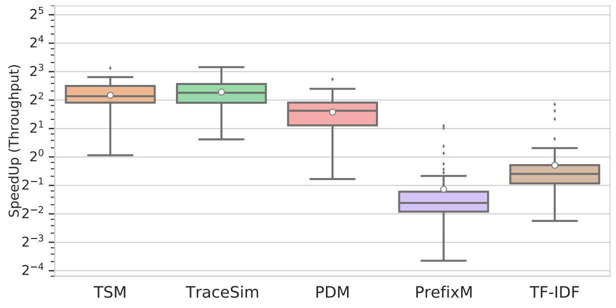
Overall, *FaST* is not only more efficient than the optimal sequence alignment methods, but it is also at least as effective as them. As shown in Figure 6.6, considering the speedup average, *FaST* is two to four times faster than PDM while its speedup ranges between 4x - 8x regarding TraceSim. In addition to this superior efficiency, our method significantly surpasses PDM and TraceSim in nine and six of the twelve possible evaluation scenarios, respectively. In the remaining ones, the performance of *FaST* and such techniques are considered as comparable since the differences in their results are not statistical significant.

As expected, the efficiency superiority of *FaST* over TSM is similar to the one observed in the previous TraceSim's analysis. However, in terms of effectiveness, we do not find statistical significance in their performances in the evaluation scenarios, except on Ubuntu regarding ΔMAP which *FaST* is superior. Despite this finding, we cannot conclude whether the normalization and function match(·) used in *FaST* are more effective than the ones proposed in TraceSim. In additional significance tests, we found that TSM's and TraceSim's performances are comparable in all scenario with the exception to Netbeans regarding RR@1.

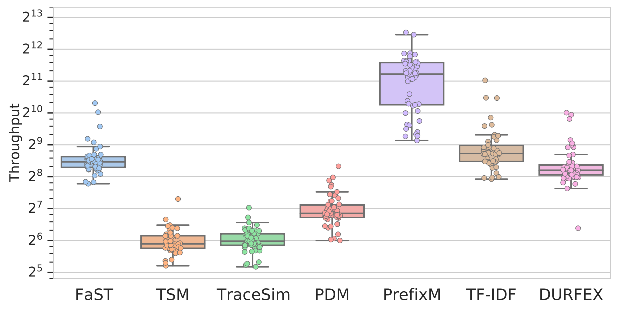
In our experiments, PrefixM is the most efficient technique. In comparison to our method, on average, it is 2.19, 4.30, 6.35, and 2.53 times faster than *FaST* in Ubuntu, Eclipse, Netbeans, and Gnome, respectively. Such high efficiency is due to the fact that the similarity is computed only considering the topmost shared frames between two stack traces, i.e., it can easily filter frames that do not affect the comparison. However, this negatively affects the method effectiveness. As reported in Figure 6.7, PrefixM is significantly outperformed by



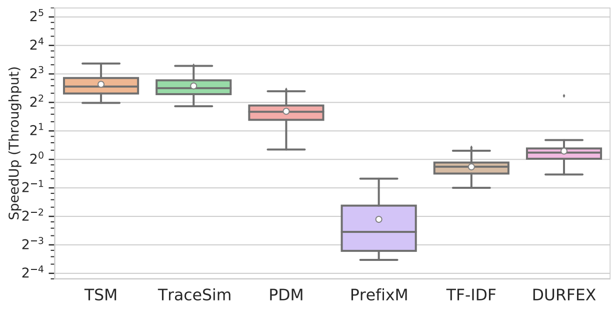
(a) Ubuntu - Throughput



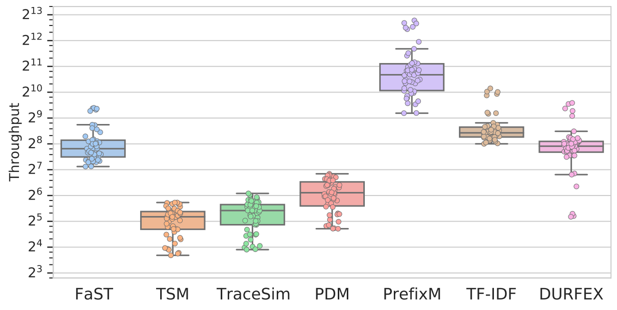
(b) Ubuntu - Speedup



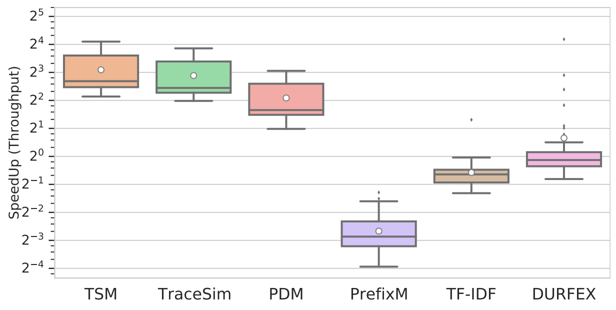
(c) Eclipse - Throughput



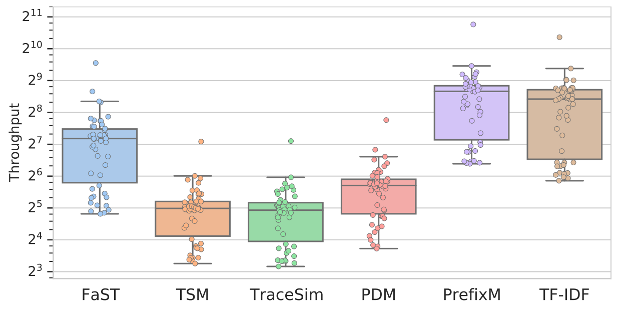
(d) Eclipse - Speedup



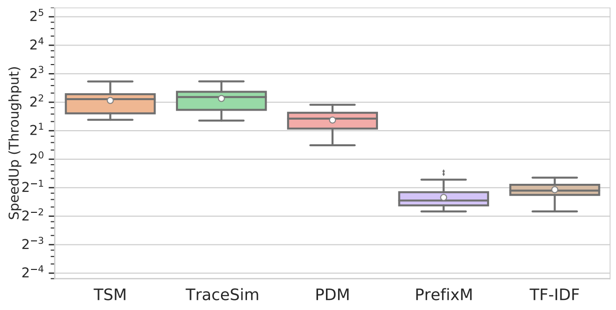
(e) Netbeans - Throughput



(f) Netbeans - Speedup

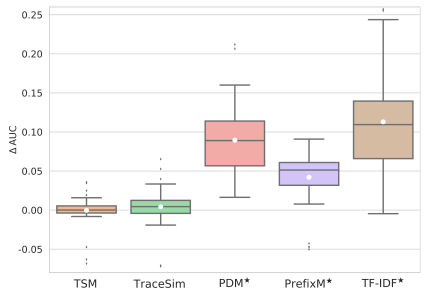


(g) Gnome - Throughput

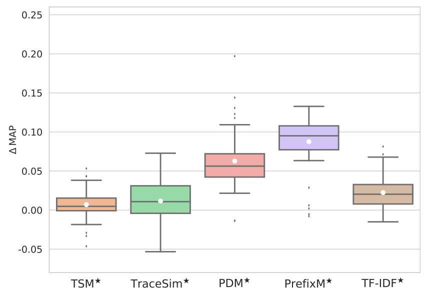


(h) Gnome - Speedup

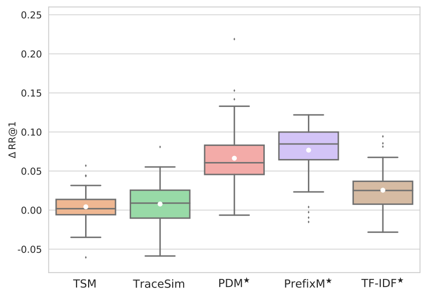
Figure 6.6 At the left, throughput (queries per second) of methods in all validation sets of Ubuntu, Eclipse, Netbeans, and Gnome. At the right, the speedup between *FaST* and the competing methods regarding throughput.



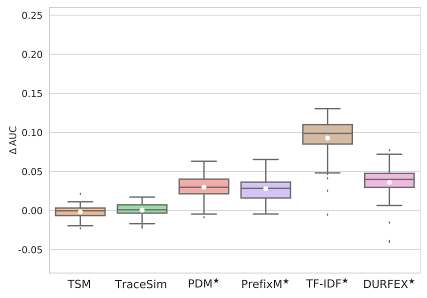
(a) Ubuntu - ΔAUC



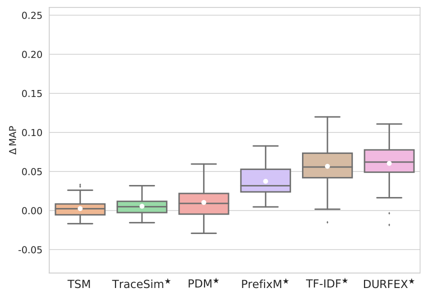
(b) Ubuntu - ΔMAP



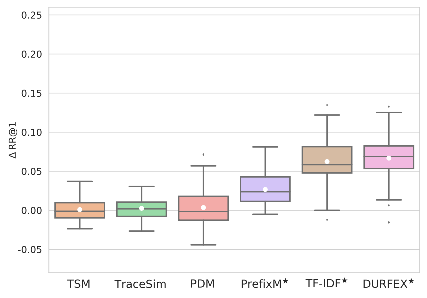
(c) Ubuntu - ΔRR@1



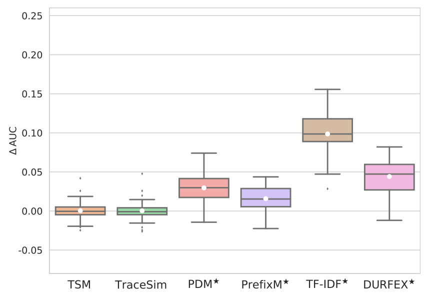
(d) Eclipse - ΔAUC



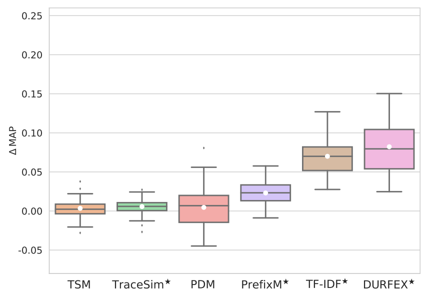
(e) Eclipse - ΔMAP



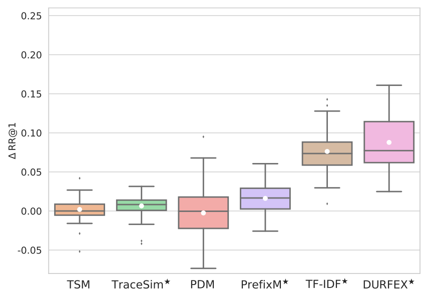
(f) Eclipse - ΔRR@1



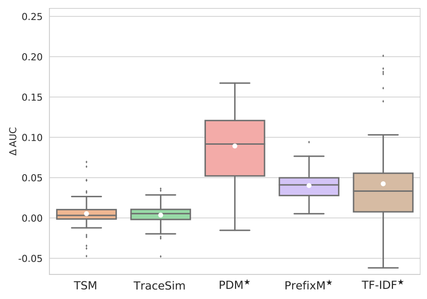
(g) Netbeans - ΔAUC



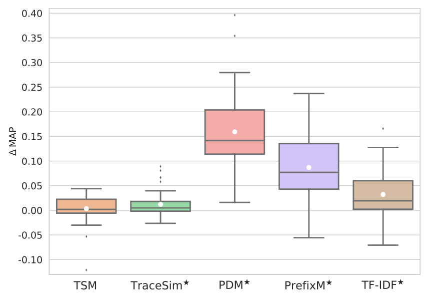
(h) Netbeans - ΔMAP



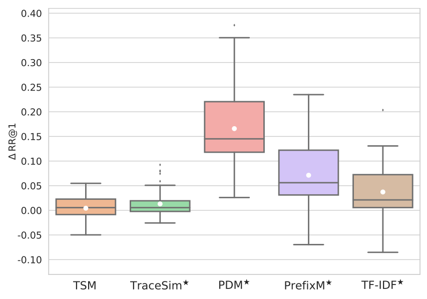
(i) Netbeans - ΔRR@1



(j) Gnome - ΔAUC



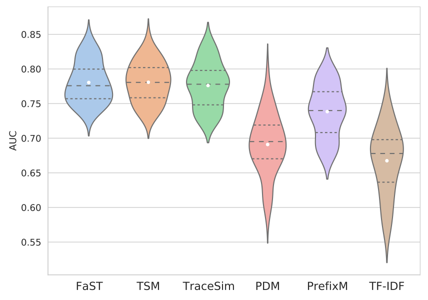
(k) Gnome - ΔMAP



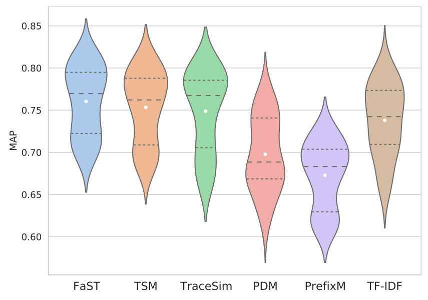
(l) Gnome - ΔRR@1

★ $p < 0.01$

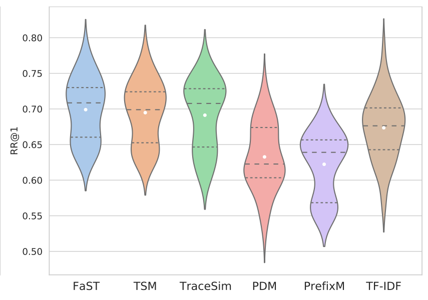
Figure 6.7 The distribution of ΔAUC, ΔMAP, and ΔRR@1 between *FaST* and each competing method in all validation sets of each dataset.



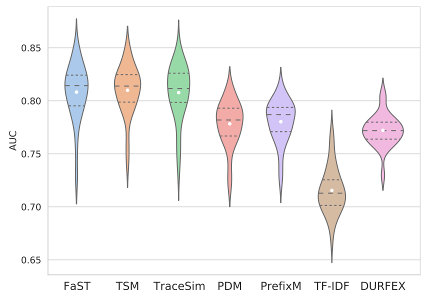
(a) Ubuntu - AUC



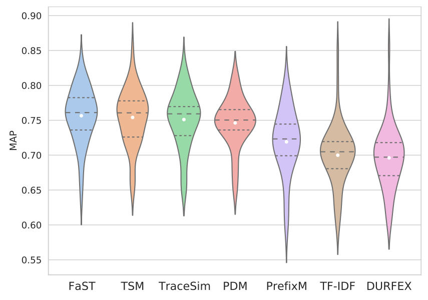
(b) Ubuntu - MAP



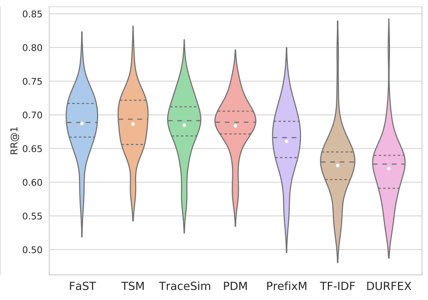
(c) Ubuntu - RR@1



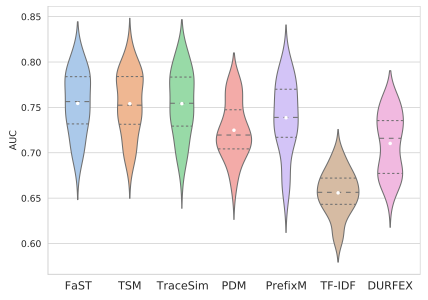
(d) Eclipse - AUC



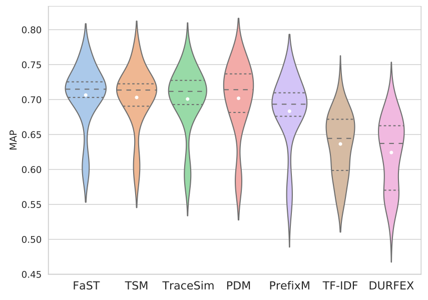
(e) Eclipse - MAP



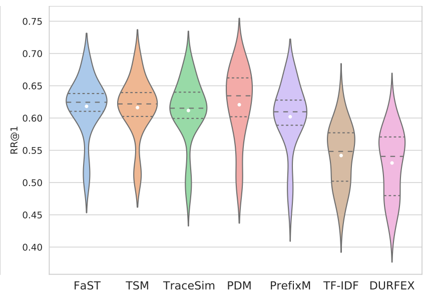
(f) Eclipse - RR@1



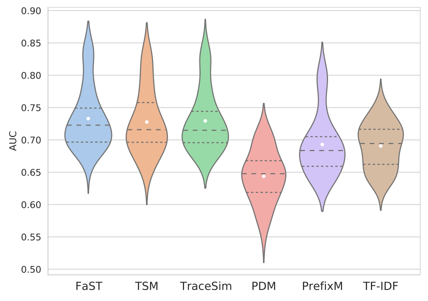
(g) Netbeans - AUC



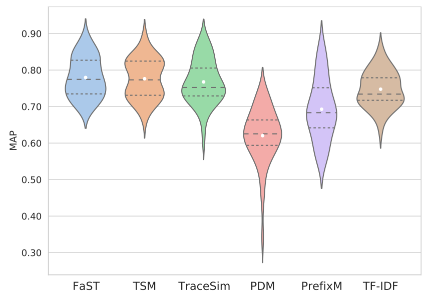
(h) Netbeans - MAP



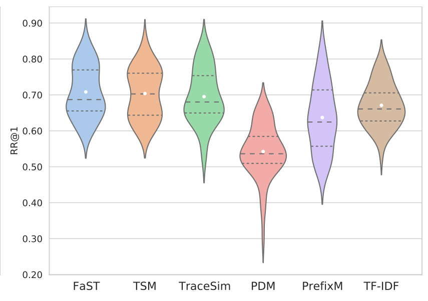
(i) Netbeans - RR@1



(j) Gnome - AUC



(k) Gnome - MAP



(l) Gnome - RR@1

Figure 6.8 The distributions of AUC, MAP, and RR@1 achieved by *FaST* and competitors in all validation sets of each dataset.

FaST regarding AUC, MAP, and RR@1 on all datasets. For instance, the lowest average of ΔAUC, ΔMAP, and ΔRR@1 between these two methods are +1.57%, +2.31%, and +1.61% in Netbeans. However, we observe higher performance differences in datasets with C++ stack traces, e.g., *FaST* largely outperforms PrefixM by 4.20%, 8.76%, and 7.67% regarding ΔAUC, ΔMAP, and ΔRR@1 in Ubuntu.

As shown in Figure 6.6, the second most efficient technique is TF-IDF. Regarding *FaST*, such method speeds the experiments, on average, by 1.22x, 1.19x, 1.49x, and 2.09x in Ubuntu, Eclipse, Netbeans, and Gnome, respectively. Such speedups are considerable lower than the ones found in PrefixM. But, similar to PrefixM, *FaST* significantly outperforms TF-IDF in all evaluation scenarios. For example, the lowest average value of ΔAUC, ΔMAP, and ΔRR@1 between FaST and TF-IDF are +4.23%, +2.23%, and +2.54%, respectively. However, such performances occurs in datasets that contain stack traces from C++ applications. Considering only Netbeans and Eclipse, those values increase to 9.27%, 5.66%, and 6.23%, respectively. Finally, although we were not able to observe a conclusive efficiency difference between *FaST* and DURFEX, the results show that our method is statistically more effective than the latter regarding ΔAUC, ΔMAP, and ΔRR@1 in all datasets.

6.6 Threats to Validity

In this section, the threats to validity of our study are presented as follows.

Data quality In this study, the experimental evaluation is based on manual labeled data provided in BTSs. However, due to the complexity associated to the deduplication task, reports might be assigned to incorrect buckets or considered as singletons by triagers. Moreover, stack traces are mostly extracted from textual data, i.e., from files and report descriptions. However, due to this unstructured data nature, data extraction is not trivial and, therefore, portion of text might be incorrectly identified as a stack trace, and vice-versa. To mitigate these problems, we employ datasets that have been used in previous studies. Additionally, regarding mislabeled data, we used data from popular open source projects that contain mature triage processes. Finally, Rodrigues et al. [118] mitigate the problem related to stack trace extraction by using parsers already employed in real environments or well-known studies.

Subject selection bias. In this paper, we perform an empirical study to compare the effectiveness and efficiency of distinct methods. Thus, due to different domain characteristics, our findings might not be observed in other software projects. To mitigate such threat, our experimental setup includes data from different projects and two distinct programming languages. Moreover, the experimental methodology and framework are developed to replicate real environments as much as possible.

6.7 Conclusions

In this study, we proposed *FaST*, a novel alignment method heuristic for crash report deduplication. In contrast to previous methods based on optimal sequence alignment, *FaST* heuristically computes the similarity of stack traces in linear time. We experimentally evaluated *FaST* and its competing methods by means of the methodology proposed in [118]. Our results revealed that *FaST* is consistently faster than previous SOTA methods while being at least as effective – it was more effective in many of the considered scenarios. In fact, our experimental results indicate that sub-optimal alignments can be as effective as optimal ones for crash report deduplication.

The proposed modifications on TraceSim's scoring scheme allows our method to compute the similarity score by exclusively considering the shared frames between stack traces. For that, the assumption is that the sums of frame weights of the stack traces are known before the algorithm execution. This method capability combined to frame independence makes our method more appropriate for effectively speeding up the deduplication by means of inverted index data structure [139] or MapReduce [140]. Thus, as a possible avenue for future works, we intend to evaluate the method speedup achieved when inverted index data structure is used and the method is implemented based on MapReduce. Moreover, analogous to prefix match, we intend to investigate different strategies to compare stack traces only considering a small portion of the frames.

CHAPTER 7 GENERAL DISCUSSION

The scope of this thesis was to investigate two relevant tasks for software maintenance: bug deduplication and crash report deduplication. In Chapter 4, we addressed the limited feature interaction between bug reports in previous deep learning methods. Leveraging attention mechanisms, we introduced SABD, a deep learning model that can dynamically focus on different portions of a report that are similar to a particular segment of another report. The proposed architecture provides a superior model capability to capture more relevant features for a particular deduplication case and, consequently, it potentially mitigates the information loss associated to the representation generation. In our experiments, in comparison with previous techniques, SABD generates superior ranked lists, i.e., a significant number of correct candidates in such lists are closer to the top positions. Thus, in practice, SABD's deployment could significantly reduce the time spent by triagers to identify duplicate reports. Regarding its efficiency, in scenarios for which SABD is a bottleneck, such issue could be mitigated by adjusting time window length and employing re-ranking techniques.

In chapters 5 and 6, we introduced sequence alignment methods for crash report deduplication (TraceSim and *FaST*) based on a novel scoring scheme in which the value of each alignment depends on the frame positions and global frequencies. Moreover, the impact of these features is controlled by parameters that are adjusted through machine learning techniques. The proposed scoring scheme provides superior robustness and effectiveness to our methods in comparison with previous techniques. Additionally, leveraging particular task characteristics, we developed an alignment heuristic that computes the similarity of two stack traces in linear time without performance degradation. In summary, besides this efficiency improvement, our contributions allow deduplication systems to: (i) separate duplicate crash reports from non-duplicate ones more effectively; (ii) add reports to their correct buckets more accurately; and (iii) generates better ranked lists in scenarios where human assistance is needed.

During this Ph.D. research, TraceSim has been employed in our partner project for deduplicating Python's stack traces. Due to the lack of labeled data, we were not able to tune the parameters as presented in Chapter 5. Thus, we separately plotted $\mathrm{lw}(\cdot)$, $\mathrm{gw}(\cdot)$, $\mathrm{diff}(\cdot)$ – equations 5.1, 5.3, 5.10, respectively – for different parameter values. Based on the provided experimental data, we chose values in which the function smoothness seemed more adequate to the domain. Then, we ran TraceSim on the available stack traces to investigate whether the computed similarity scores were satisfactory. A specialist provided feedback regarding the method behaviour which helped us to find better parameter values. TraceSim was shown to be effective in such preliminary data and, currently, its deployment on the production environment is in progress. In the near future, we also plan to test and deploy *FaST* in such project.

In both deduplication tasks, most of works have evaluated their method using their own datasets. Many of them implemented crawlers to retrieve reports from open source projects, but, frequently, such retrieved data was not publicly available. Recently, in bug deduplication, few works, including ours, employed the data published by Lazar et al. [63]. Nevertheless, the train-test split was done using distinct strategies in such studies. As shown by our results and in Rakha et al. [111], the performances achieved on different portions of the data can substantially vary. Thus, even though the experiments are performed using the same set of reports, it may be misleading to directly compare methods based on the performance values reported on distinct test and training sets.

In addition to datasets, the majority of the studies did not make available their developed code. Thus, especially in bug deduplication where techniques are more complex and literature is more vast, it may be difficult to correctly implement previous methods since results often cannot be reproduced due to the lack of data availability and crucial components of the methods may not have been accurately reported. For instance, in our first article, we spent months working on the implementation of the neural network proposed by Deshmukh et al. [68] until we were able to achieve adequate performance.

Finally, in the literature of bug deduplication and crash report deduplication, there is no widely adopted evaluation methodology. Moreover, even for specific approaches, the performed evaluations in the works can contain some differences that may significantly impact the achieved results. Therefore, it is unfeasible to compare several proposed methods in the literature.

This lack of consensus in the field regarding datasets and methodologies added to non-availability of data and source code made more challenging the study and investigation of both tasks. In this thesis, we mitigated such issue by choosing and combining available methodologies to make them more adherent to real environments. Furthermore, we tried to extensively compare our methods with previous techniques related to the article objectives. For the majority of the previous techniques, it was required to carefully implement them from scratch by following their descriptions in the studies.

Finally, we would like to highlight that the data used in our articles and the developed code are all available on popular repositories.

CHAPTER 8 CONCLUSION

In this thesis, we studied and proposed techniques for addressing bug deduplication and crash report deduplication. Our main contributions are summarized in Section 8.1. Finally, in Section 8.2, we discuss limitations of the proposed methods and suggest potential research avenues for future investigation.

8.1 Thesis Contributions

In Chapter 4, we introduced *SABD*, a novel deep learning model for bug deduplication whose core component is a soft alignment comparison layer. In contrast to previous deep learning works, such layer can dynamically extract features on distinct parts of a report that are related to a given word in the other report. Based on such word representation comparisons, our model generates joint representations of the textual data in reports. As demonstrated in a ablation study, the proposed architecture is more effective than previous deep learning methods. Moreover, our model achieved SOTA performances in all evaluated scenarios.

In Chapter 5, we presented a novel optimal sequence alignment method for crash report deduplication, called *TraceSim*. Such method finds the optimal global alignment score between stack traces by means of a scoring scheme whose alignment values are computed based on the frame position and global frequency. The impact of such pieces of information in the alignment values are controlled by parameters learned through ML algorithms. We extensively compared TraceSim with the previous techniques by means of a new methodology for crash report deduplication. Our experiments revealed that, in contrast to other methods, TraceSim consistently performs well regarding distinct metrics and datasets. Moreover, in most of the evaluation scenarios, TraceSim was significantly superior to competitive techniques. Finally, an ablation study demonstrated the effectiveness of TraceSim's elements and the proposed scheme for computing mismatches and gap alignments.

Finally, in Chapter 6, we investigated how to improve the efficiency of sequence alignment methods. To reduce time complexity, we proposed to remove the following constraints of optimal global alignment problem: the algorithm must find an solution that is optimal and preserves sequence order. Based on that, we introduced a novel method, called *FaST*, that measures stack trace similarity in linear time. *FaST* matches frames of an specific identifier from the top-most positions until the bottom ones. After exhausting all possible matches, the remaining frames are aligned to gaps. Even though *FaST* finds sub-optimal solutions that may not maintain the original sequence order, we demonstrated, through a series of experiments, that our method is not only significantly faster than TraceSim, but it also achieves SOTA performances.

8.2 Limitations and Future Research

The majority of methods in bug deduplication literature, including SABD, are heavily based on textual data similarity. However, other information sources might be as much, or even more, important in some deduplication cases, e.g, images and videos of errors are usually appropriate to effectively describe user interface bugs. Thus, a potential research avenue is to investigate systems that can dynamically select and combine multiple different information types (e.g., image, texts, source code, traces, logs . . .) for bug deduplication.

Moreover, in crash report deduplication, sequence matching algorithms consider two frames with different identifiers as dissimilar even when subroutines in such frames present comparable behaviours. Thus, minor differences in identifiers might considerable affect the similarity measurement. To mitigate such problem, future researchers could investigate how to improve frame comparison in such techniques. Similar to deep learning models [97], unsupervised techniques could be applied to learn distributed representation of frames. Thus, two frames could be compared based on their representation similarity. However, in such approach, it would be challenging to adequately represent out-of-vocabulary identifiers which are frequent once new functions are added during system lifetime. Hence, we argue that a more robust approach could be based on textual similarity regarding token or even character level. The rationale behind this is that two subroutines with similar qualified names are expected to present comparable behaviours.

In Chapter 6, we accelerated deduplication by improving the efficiency of the similarity measurement computation. However, another complementary and effective approach consists in reducing the number of candidates for a query. In both deduplication tasks, the use of time windows is a popular strategy to filter candidates. However, such strategy is limited since the number of candidates may still be large and dissimilar reports to the query might be considered for the comparison. Thus, a promising research avenue is to study techniques to speed up deduplication by selecting a subset of submitted reports without a substantial effectiveness degradation.

Even though few evaluation methodologies were proposed for deduplication of bug and crash reports, to the best of our knowledge, none of the studies in the literature have compared such methodologies and presented their limitations and strengths. Moreover, based on such comparison, a future work could establish an evaluation methodology that would be adequate for different techniques and adherent to real environments. Hence, in addition to the establishment of standard datasets, this would help to compare new methods and keep a track of the literature evolution in both tasks.

REFERENCES

[1] A. Hindle, A. Alipour, and E. Stroulia, “A contextual approach towards more accurate duplicate bug report detection and ranking,” *Empirical Software Engineering*, vol. 21, no. 2, pp. 368–410, 2016.

[2] J. Anvik, L. Hiew, and G. C. Murphy, “Who should fix this bug?” in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE ’06. New York, NY, USA: ACM, 2006, pp. 361–370. [Online]. Available: http://doi.acm.org/10.1145/1134285.1134336

[3] C. R. Reis and R. P. de Mattos Fortes, “An overview of the software engineering process and tools in the mozilla project,” in *Proceedings of the Open Source Software Development Workshop*, 2002, pp. 155–175.

[4] S. Banerjee, B. Cukic, and D. Adjeroh, “Automated duplicate bug report classification using subsequence matching,” in *2012 IEEE 14th International Symposium on High-Assurance Systems Engineering*, Oct 2012, pp. 74–81.

[5] J. Anvik, “Assisting bug report triage through recommendation,” Ph.D. dissertation, University of British Columbia, 2007. [Online]. Available: https://open.library.ubc.ca/collections/ubctheses/24/items/1.0051337

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

[7] I. Ahmed, N. Mohan, and C. Jensen, “The impact of automatic crash reports on bug triaging and development in mozilla,” in *Proceedings of The International Symposium on Open Collaboration*, ser. OpenSym ’14. New York, NY, USA: Association for Computing Machinery, 2014, p. 1–8. [Online]. Available: https://doi.org/10.1145/2641580.2641585

[8] A. Schroter, A. Schröter, N. Bettenburg, and R. Premraj, “Do stack traces help developers fix bugs?” in *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. IEEE, 2010, pp. 118–121.

[9] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, “An approach to detecting duplicate bug reports using natural language and execution information,” in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE

'08. New York, NY, USA: ACM, 2008, pp. 461–470. [Online]. Available: http://doi.acm.org/10.1145/1368088.1368151

[10] J. C. Campbell, E. A. Santos, and A. Hindle, "The unreasonable effectiveness of traditional information retrieval in crash report deduplication," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: ACM, 2016, pp. 269–280. [Online]. Available: http://doi.acm.org/10.1145/2901739.2901766

[11] N. Bettenburg, R. Premraj, T. Zimmermann, and . Sunghun Kim, "Duplicate bug reports considered harmful ... really?" in *2008 IEEE International Conference on Software Maintenance*, Sep. 2008, pp. 337–345.

[12] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt, "Debugging in the (very) large: Ten years of implementation and experience," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 103–116. [Online]. Available: https://doi.org/10.1145/1629575.1629586

[13] T. Dhaliwal, F. Khomh, and Y. Zou, "Classifying field crash reports for fixing bugs: A case study of mozilla firefox," in *Proceedings of the 2011 27th IEEE International Conference on Software Maintenance*, ser. ICSM '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 333–342. [Online]. Available: https://doi.org/10.1109/ICSM.2011.6080800

[14] C. Sun, D. Lo, S.-C. Khoo, and J. Jiang, "Towards more accurate retrieval of duplicate bug reports," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 253–262. [Online]. Available: http://dx.doi.org/10.1109/ASE.2011.6100061

[15] S. Banerjee, Z. Syed, J. Helmick, M. Culp, K. Ryan, and B. Cukic, "Automated triaging of very large bug repositories," *Information and Software Technology*, vol. 89, pp. 1–13, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584916301653

[16] S. Chopra, R. Hadsell, and Y. LeCun, "Learning a similarity metric discriminatively, with application to face verification," in *Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05) - Volume*

1 - Volume 01, ser. CVPR '05. USA: IEEE Computer Society, 2005, p. 539–546. [Online]. Available: https://doi.org/10.1109/CVPR.2005.202

[17] T. M. Lai, T. Bui, and S. Li, "A review on deep learning techniques applied to answer selection," in *Proceedings of the 27th International Conference on Computational Linguistics*. Santa Fe, New Mexico, USA: Association for Computational Linguistics, Aug. 2018, pp. 2132–2144.

[18] M. Tan, C. dos Santos, B. Xiang, and B. Zhou, "Improved representation learning for question answer matching," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Berlin, Germany: Association for Computational Linguistics, August 2016, pp. 464–473. [Online]. Available: http://www.aclweb.org/anthology/P16-1044

[19] L. Poddar, L. Neves, W. Brendel, L. Marujo, S. Tulyakov, and P. Karuturi, "Train one get one free: Partially supervised neural network for bug report duplicate detection and clustering," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Industry Papers)*. Minneapolis - Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 157–165. [Online]. Available: https://www.aclweb.org/anthology/N19-2020

[20] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[21] M. Brodie, S. Ma, G. Lohman, L. Mignet, N. Modani, M. Wilding, J. Champlin, and P. Sohn, "Quickly finding known software problems via automated symptom matching," in *Second International Conference on Autonomic Computing (ICAC'05)*, June 2005, pp. 101–110.

[22] 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 of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 1084–1093. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337364

[23] C. D. Manning and H. Schütze, *Foundations of Statistical Natural Language Processing*. Cambridge, MA, USA: MIT Press, 1999.

[24] D. Jurafsky and J. H. Martin. (2019) Speech and language processing. Draft of the 3rd edition book. [Online]. Available: https://web.stanford.edu/~jurafsky/slp3/

[25] A. Singhal, C. Buckley, and M. Mitra, "Pivoted document length normalization," in *Proceedings of the 19th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, ser. SIGIR '96. New York, NY, USA: ACM, 1996, pp. 21–29. [Online]. Available: http://doi.acm.org/10.1145/243199.243206

[26] S. Robertson, H. Zaragoza, and M. Taylor, "Simple bm25 extension to multiple weighted fields," in *Proceedings of the Thirteenth ACM International Conference on Information and Knowledge Management*, ser. CIKM '04. New York, NY, USA: ACM, 2004, pp. 42–49. [Online]. Available: http://doi.acm.org/10.1145/1031171.1031181

[27] H. Zaragoza, N. Craswell, M. Taylor, S. Saria, and S. Robertson, "Microsoft cambridge at trec-13: Web and hard tracks," in *IN PROCEEDINGS OF TREC 2004*, 2004.

[28] S. Robertson and H. Zaragoza, "The probabilistic relevance framework: Bm25 and beyond," *Found. Trends Inf. Retr.*, vol. 3, no. 4, pp. 333–389, Apr. 2009. [Online]. Available: http://dx.doi.org/10.1561/1500000019

[29] Y. Goldberg, "A primer on neural network models for natural language processing," *J. Artif. Int. Res.*, vol. 57, no. 1, pp. 345–420, Sep. 2016. [Online]. Available: http://dl.acm.org/citation.cfm?id=3176748.3176757

[30] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, p. 436, 2015.

[31] Y. Bengio, A. Courville, and P. Vincent, "Representation learning: A review and new perspectives," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 35, no. 8, pp. 1798–1828, Aug 2013.

[32] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2009.

[33] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[34] O. Delalleau and Y. Bengio, "Shallow vs. deep sum-product networks," in *Advances in Neural Information Processing Systems 24*, J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, Eds. Curran

Associates, Inc., 2011, pp. 666–674. [Online]. Available: http://papers.nips.cc/paper/4350-shallow-vs-deep-sum-product-networks.pdf

[35] M. A. Nielsen, *Neural Networks and Deep Learning.* Determination Press, 2015. [Online]. Available: http://neuralnetworksanddeeplearning.com/

[36] A. Graves, *Supervised sequence labelling with recurrent neural networks.* Springer, 2012.

[37] M. Schuster and K. Paliwal, “Bidirectional recurrent neural networks,” *Trans. Sig. Proc.*, vol. 45, no. 11, pp. 2673–2681, Nov. 1997. [Online]. Available: http://dx.doi.org/10.1109/78.650093

[38] R. Jozefowicz, W. Zaremba, and I. Sutskever, “An empirical exploration of recurrent network architectures,” in *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ser. ICML’15. JMLR.org, 2015, pp. 2342–2350. [Online]. Available: http://dl.acm.org/citation.cfm?id=3045118.3045367

[39] Y. Bengio, P. Simard, and P. Frasconi, “Learning long-term dependencies with gradient descent is difficult,” *Trans. Neur. Netw.*, vol. 5, no. 2, pp. 157–166, Mar. 1994. [Online]. Available: http://dx.doi.org/10.1109/72.279181

[40] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997. [Online]. Available: http://dx.doi.org/10.1162/neco.1997.9.8.1735

[41] F. A. Gers, J. Schmidhuber, and F. Cummins, “Learning to forget: continual prediction with lstm,” in *1999 Ninth International Conference on Artificial Neural Networks ICANN 99. (Conf. Publ. No. 470)*, vol. 2, Sep. 1999, pp. 850–855 vol.2.

[42] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, “TensorFlow: Large-scale machine learning on heterogeneous systems,” 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[43] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, “Automatic differentiation in pytorch,” in *NIPS-W*, 2017.

[44] Z. Lin, M. Feng, C. N. dos Santos, M. Yu, B. Xiang, B. Zhou, and Y. Bengio, “A structured self-attentive sentence embedding,” in *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, 2017. [Online]. Available: https://openreview.net/forum?id=BJC_jUqxe

[45] I. Sutskever, O. Vinyals, and Q. V. Le, “Sequence to sequence learning with neural networks,” in *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2014, pp. 3104–3112. [Online]. Available: http://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks.pdf

[46] S. Venugopalan, M. Rohrbach, J. Donahue, R. Mooney, T. Darrell, and K. Saenko, “Sequence to sequence - video to text,” in *The IEEE International Conference on Computer Vision (ICCV)*, December 2015.

[47] R. Nallapati, B. Zhou, C. dos Santos, C. Gulcehre, and B. Xiang, “Abstractive text summarization using sequence-to-sequence rnns and beyond,” in *Proceedings of The 20th SIGNLL Conference on Computational Natural Language Learning.* Association for Computational Linguistics, 2016, pp. 280–290. [Online]. Available: http://aclweb.org/anthology/K16-1028

[48] Z. Xu, S. Wang, F. Zhu, and J. Huang, “Seq2seq fingerprint: An unsupervised deep molecular embedding for drug discovery,” in *Proceedings of the 8th ACM International Conference on Bioinformatics, Computational Biology,and Health Informatics*, ser. ACM-BCB ’17. New York, NY, USA: ACM, 2017, pp. 285–294. [Online]. Available: http://doi.acm.org/10.1145/3107411.3107424

[49] T. Luong, H. Pham, and C. D. Manning, “Effective approaches to attention-based neural machine translation,” in *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing.* Association for Computational Linguistics, 2015, pp. 1412–1421. [Online]. Available: http://aclweb.org/anthology/D15-1166

[50] L. Qiu, Y. Cao, Z. Nie, Y. Yu, and Y. Rui, “Learning word representation considering proximity and ambiguity,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 28, no. 1, 2014.

[51] Y. Bengio, "Neural net language models," *Scholarpedia*, vol. 3, no. 1, p. 3881, 2008.

[52] J. Turian, L. Ratinov, and Y. Bengio, "Word representations: A simple and general method for semi-supervised learning," in *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, ser. ACL '10. Stroudsburg, PA, USA: Association for Computational Linguistics, 2010, pp. 384–394. [Online]. Available: http://dl.acm.org/citation.cfm?id=1858681.1858721

[53] C. D. Santos and B. Zadrozny, "Learning character-level representations for part-of-speech tagging," in *Proceedings of the 31st International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, E. P. Xing and T. Jebara, Eds., vol. 32, no. 2. Bejing, China: PMLR, 22–24 Jun 2014, pp. 1818–1826. [Online]. Available: http://proceedings.mlr.press/v32/santos14.html

[54] R. Wang, W. Liu, and C. McDonald, "Corpus-independent generic keyphrase extraction using word embedding vectors," in *Software Engineering Research Conference*, vol. 39, 2014.

[55] Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin, "A neural probabilistic language model," *JOURNAL OF MACHINE LEARNING RESEARCH*, vol. 3, pp. 1137–1155, 2003.

[56] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in Neural Information Processing Systems 26*, C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2013, pp. 3111–3119. [Online]. Available: http://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality.pdf

[57] G. E. Hinton *et al.*, "Learning distributed representations of concepts," in *Proceedings of the eighth annual conference of the cognitive science society*, vol. 1. Amherst, MA, 1986, p. 12.

[58] J. Bromley, I. Guyon, Y. LeCun, E. Säckinger, and R. Shah, "Signature verification using a "siamese" time delay neural network," in *Advances in Neural Information Processing Systems 6*, J. D. Cowan, G. Tesauro, and J. Alspector, Eds. Morgan-Kaufmann, 1994, pp. 737–744. [Online]. Available: http://papers.nips.cc/paper/769-signature-verification-using-a-siamese-time-delay-neural-network.pdf

[59] W. Liao, M. Y. Yang, N. Zhan, and B. Rosenhahn, "Triplet-based Deep Similarity Learning for Person Re-Identification," *arXiv e-prints*, p. arXiv:1802.03254, Feb 2018.

[60] P. Neculoiu, M. Versteegh, and M. Rotaru, "Learning text similarity with siamese recurrent networks," in *Proceedings of the 1st Workshop on Representation Learning for NLP.* Association for Computational Linguistics, 2016, pp. 148–157. [Online]. Available: http://aclweb.org/anthology/W16-1617

[61] H. He, K. Gimpel, and J. Lin, "Multi-perspective sentence similarity modeling with convolutional neural networks," in *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing.* Association for Computational Linguistics, 2015, pp. 1576–1586. [Online]. Available: http://aclweb.org/anthology/D15-1181

[62] 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*, vol. 121, pp. 223–233, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121216000546

[63] A. Lazar, S. Ritchey, and B. Sharif, "Improving the accuracy of duplicate bug report detection using textual similarity measures," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: ACM, 2014, pp. 308–311. [Online]. Available: http://doi.acm.org/10.1145/2597073.2597088

[64] Y. Tian, C. Sun, and D. Lo, "Improved duplicate bug report identification," in *Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering*, ser. CSMR '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 385–390. [Online]. Available: https://doi.org/10.1109/CSMR.2012.48

[65] L. Feng, L. Song, C. Sha, and X. Gong, "Practical duplicate bug reports detection in a large web-based development community," in *Web Technologies and Applications*, Y. Ishikawa, J. Li, W. Wang, R. Zhang, and W. Zhang, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 709–720.

[66] K. Aggarwal, F. Timbers, T. Rutgers, A. Hindle, E. Stroulia, and R. Greiner, "Detecting duplicate bug reports with software engineering domain knowledge: Detecting duplicate bug reports with software-engineering domain knowledge," *Journal of Software: Evolution and Process*, vol. 29, 10 2016.

[67] T. M. Rocha and A. L. D. C. Carvalho, "Siameseqat: A semantic context-based duplicate bug report detection using replicated cluster information," *IEEE Access*, vol. 9, pp. 44 610–44 630, 2021.

[68] J. Deshmukh, A. K. M, 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 (ICSME)*, Sep. 2017, pp. 115–124.

[69] G. Xiao, X. Du, Y. Sui, and T. Yue, “Hindbr: Heterogeneous information network based duplicate bug report prediction,” in *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, 2020, pp. 195–206.

[70] L. Feng, L. Song, C. Sha, and X. Gong, “Practical duplicate bug reports detection in a large web-based development community,” in *Asia-Pacific Web Conference*. Springer, 2013, pp. 709–720.

[71] A. Hindle, N. A. Ernst, M. W. Godfrey, and J. Mylopoulos, “Automated topic naming to support cross-project analysis of software maintenance activities,” in *Proceedings of the 8th Working Conference on Mining Software Repositories*, ser. MSR ’11. New York, NY, USA: Association for Computing Machinery, 2011, p. 163–172. [Online]. Available: https://doi.org/10.1145/1985441.1985466

[72] N. Cooper, C. Bernal-Cárdenas, O. Chaparro, K. Moran, and D. Poshyvanyk, “It takes two to tango: Combining visual and textual information for detecting duplicate video-based bug reports,” in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 957–969.

[73] T. Chen, S. Kornblith, M. Norouzi, and G. Hinton, “A simple framework for contrastive learning of visual representations,” in *International conference on machine learning*. PMLR, 2020, pp. 1597–1607.

[74] N. Jalbert and W. Weimer, “Automated duplicate detection for bug tracking systems,” in *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, June 2008, pp. 52–61.

[75] T. Prifti, S. Banerjee, and B. Cukic, “Detecting bug duplicate reports through local references,” in *Proceedings of the 7th International Conference on Predictive Models in Software Engineering*, ser. Promise ’11. New York, NY, USA: ACM, 2011, pp. 8:1–8:9. [Online]. Available: http://doi.acm.org/10.1145/2020390.2020398

[76] P. Runeson, M. Alexandersson, and O. Nyholm, “Detection of duplicate defect reports using natural language processing,” in *Proceedings of the 29th International Conference on Software Engineering*, ser. ICSE ’07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 499–510. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2007.32

[77] J. Zhou and H. Zhang, “Learning to rank duplicate bug reports,” in *Proceedings of the 21st ACM International Conference on Information and Knowledge Management*, ser. CIKM ’12. New York, NY, USA: ACM, 2012, pp. 852–861. [Online]. Available: http://doi.acm.org/10.1145/2396761.2396869

[78] M.-J. Lin and C.-Z. Yang, “An improved discriminative model for duplication detection on bug reports with cluster weighting,” in *Proceedings of the 2014 IEEE 38th Annual Computer Software and Applications Conference*, ser. COMPSAC ’14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 117–122. [Online]. Available: https://doi.org/10.1109/COMPSAC.2014.18

[79] X. Yang, D. Lo, X. Xia, L. Bao, and J. Sun, “Combining word embedding with information retrieval to recommend similar bug reports,” in *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, Oct 2016, pp. 127–137.

[80] C. Yang, H. Du, S. Wu, and I. Chen, “Duplication detection for software bug reports based on bm25 term weighting,” in *2012 Conference on Technologies and Applications of Artificial Intelligence*, Nov 2012, pp. 33–38.

[81] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo, “A discriminative model approach for accurate duplicate bug report retrieval,” in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE ’10. New York, NY, USA: ACM, 2010, pp. 45–54. [Online]. Available: http://doi.acm.org/10.1145/1806799.1806811

[82] A. T. Nguyen, T. T. Nguyen, 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*, ser. ASE 2012. New York, NY, USA: ACM, 2012, pp. 70–79. [Online]. Available: http://doi.acm.org/10.1145/2351676.2351687

[83] A. Sureka and P. Jalote, “Detecting duplicate bug report using character n-gram-based features,” in *Proceedings of the 2010 Asia Pacific Software Engineering Conference*, ser. APSEC ’10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 366–374. [Online]. Available: http://dx.doi.org/10.1109/APSEC.2010.49

[84] F. Šarić, G. Glavaš, M. Karan, J. Šnajder, and B. Dalbelo Bašić, “TakeLab: Systems for measuring semantic text similarity,” in **SEM 2012: The First Joint Conference on Lexical and Computational Semantics – Volume 1: Proceedings of the main conference and the shared task, and Volume 2: Proceedings of the Sixth International*

Workshop on Semantic Evaluation (SemEval 2012). Montréal, Canada: Association for Computational Linguistics, 7-8 Jun. 2012, pp. 441–448.

[85] S. Banerjee, Z. Syed, J. Helmick, and B. Cukic, "A fusion approach for classifying duplicate problem reports," in *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2013, pp. 208–217.

[86] J. Zou, L. Xu, M. Yang, X. Zhang, J. Zeng, and S. Hirokawa, "Automated duplicate bug report detection using multi-factor analysis," *IEICE TRANSACTIONS on Information and Systems*, vol. 99, no. 7, pp. 1762–1775, 2016.

[87] A. Budhiraja, R. Reddy, and M. Shrivastava, "Lwe: Lda refined word embeddings for duplicate bug report detection," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 165–166. [Online]. Available: http://doi.acm.org/10.1145/3183440.3195078

[88] A. Budhiraja, K. Dutta, M. Shrivastava, and R. Reddy, "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*, ser. ICTIR '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 167–170. [Online]. Available: https://doi.org/10.1145/3234944.3234949

[89] A. Budhiraja, K. Dutta, R. Reddy, and M. Shrivastava, "Poster: Dwen: Deep word embedding network for duplicate bug report detection in software repositories," in *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, May 2018, pp. 193–194.

[90] Q. Xie, Z. Wen, J. Zhu, C. Gao, and Z. Zheng, "Detecting duplicate bug reports with convolutional neural networks," in *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, Nara, Japan, Dec 2018, pp. 416–425.

[91] A. Kukkar, R. Mohana, Y. Kumar, A. Nayyar, M. Bilal, and K.-S. Kwak, "Duplicate bug report detection and classification system based on deep learning technique," *IEEE Access*, vol. 8, pp. 200 749–200 763, 2020.

[92] J. He, L. Xu, M. Yan, X. Xia, and Y. Lei, *Duplicate Bug Report Detection Using Dual-Channel Convolutional Neural Networks*. New York, NY, USA: Association for Computing Machinery, 2020, p. 117–127. [Online]. Available: https://doi.org/10.1145/3387904.3389263

[93] J. Lerch and M. Mezini, "Finding duplicates of your yet unwritten bug report," in *Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering*, ser. CSMR '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 69–78. [Online]. Available: http://dx.doi.org/10.1109/CSMR.2013.17

[94] K. K. 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, QRS 2017, Prague, Czech Republic, July 25-29, 2017*. IEEE, 2017, pp. 240–250.

[95] N. Modani, R. Gupta, G. Lohman, T. Syeda-Mahmood, and L. Mignet, "Automatically identifying known software problems," in *Proceedings of the 2007 IEEE 23rd International Conference on Data Engineering Workshop*, ser. ICDEW '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 433–441. [Online]. Available: http://dx.doi.org/10.1109/ICDEW.2007.4401026

[96] K. Bartz, J. W. Stokes, J. C. Platt, R. Kivett, D. Grant, S. Calinoiu, and G. Loihle, "Finding similar failures using callstack similarity," in *Proceedings of the Third Conference on Tackling Computer Systems Problems with Machine Learning Techniques*, ser. SysML'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 1–1. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855895.1855896

[97] A. Khvorov, R. Vasiliev, G. Chernishev, I. M. R. Rodrigues, D. Koznov, and N. Povarov, "S3m: Siamese stack (trace) similarity measure," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 2021, pp. 266–270.

[98] S. Kim, T. Zimmermann, and N. Nagappan, "Crash graphs: An aggregated view of multiple crashes to improve crash triage," in *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*. IEEE, 2011, pp. 486–493.

[99] N. E. Koopaei and A. 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*, ser. CASCON '15. USA: IBM Corp., 2015, p. 201–210.

[100] 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*, vol. 113, pp. 98–109, 2019.

[101] G. Jiang, H. Chen, C. Ungureanu, and K. Yoshihira, “Trace analysis for fault detection in application servers,” in *Autonomic Computing.* CRC Press, 2018, pp. 495–516.

[102] A. Moroo, A. Aizawa, and T. Hamamoto, “Reranking-based crash report deduplication,” in *SEKE.* KSI Research Inc. and Knowledge Systems Institute Graduate School, 2017, pp. 507–510.

[103] I. M. Rodrigues, D. Aloise, E. R. Fernandes, and M. Dagenais, “A soft alignment model for bug deduplication,” in *Proceedings of the 17th International Conference on Mining Software Repositories.* New York, NY, USA: Association for Computing Machinery, 2020, p. 43–53. [Online]. Available: https://doi.org/10.1145/3379597.3387470

[104] A. Parikh, O. Täckström, D. Das, and J. Uszkoreit, “A decomposable attention model for natural language inference,” in *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing.* Austin, Texas: Association for Computational Linguistics, Nov. 2016, pp. 2249–2255.

[105] R. Nallapati, “Discriminative models for information retrieval,” in *Proceedings of the 27th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, ser. SIGIR ’04. New York, NY, USA: ACM, 2004, pp. 64–71. [Online]. Available: http://doi.acm.org/10.1145/1008992.1009006

[106] D. M. Blei, “Probabilistic topic models,” *Commun. ACM*, vol. 55, no. 4, p. 77–84, Apr. 2012. [Online]. Available: https://doi.org/10.1145/2133806.2133826

[107] Y. Tay, L. A. Tuan, and S. C. Hui, “Multi-cast attention networks,” in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery &#38; Data Mining*, ser. KDD ’18. New York, NY, USA: ACM, 2018, pp. 2299–2308. [Online]. Available: http://doi.acm.org/10.1145/3219819.3220048

[108] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, u. Kaiser, and I. Polosukhin, “Attention is all you need,” in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS’17. Red Hook, NY, USA: Curran Associates Inc., 2017, p. 6000–6010.

[109] S. Wang and J. Jiang, “A compare-aggregate model for matching text sequences,” *CoRR*, vol. abs/1611.01747, 2016.

[110] A. Lazar, S. Ritchey, and B. Sharif, “Generating duplicate bug datasets,” in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser.

MSR 2014. New York, NY, USA: ACM, 2014, pp. 392–395. [Online]. Available: http://doi.acm.org/10.1145/2597073.2597128

[111] M. S. Rakha, C. 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. 44, no. 12, pp. 1245–1268, 2018.

[112] J. L. Davidson, N. Mohan, and C. Jensen, “Coping with duplicate bug reports in free/open source software projects,” in *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Sep. 2011, pp. 101–108.

[113] J. Pennington, R. Socher, and C. D. Manning, “Glove: Global vectors for word representation,” in *Empirical Methods in Natural Language Processing (EMNLP)*, 2014, pp. 1532–1543. [Online]. Available: http://www.aclweb.org/anthology/D14-1162

[114] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *CoRR*, vol. abs/1412.6980, 2014.

[115] M. Feng, B. Xiang, M. R. Glass, L. Wang, and B. Zhou, “Applying deep learning to answer selection: A study and an open task,” in *2015 IEEE Workshop on Automatic Speech Recognition and Understanding (ASRU)*, Dec 2015, pp. 813–820.

[116] J. Rao, H. He, and J. Lin, “Noise-contrastive estimation for answer selection with deep neural networks,” in *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, ser. CIKM ’16. New York, NY, USA: ACM, 2016, pp. 1913–1916. [Online]. Available: http://doi.acm.org/10.1145/2983323.2983872

[117] F. Schroff, D. Kalenichenko, and J. Philbin, “Facenet: A unified embedding for face recognition and clustering,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 815–823.

[118] I. M. Rodrigues, A. Khvorov, D. Aloise, R. Vasiliev, D. Koznov, E. R. Fernandes, G. Chernishev, D. Luciv, and N. Povarov, “Tracesim: An alignment method for computing stack trace similarity,” *Empirical Software Engineering*, vol. 27, no. 2, p. 53, Mar 2022. [Online]. Available: https://doi.org/10.1007/s10664-021-10070-w

[119] S. Needleman and C. Wunsch, “A general method applicable to the search for similarities in the amino acid sequence of two proteins,” *Journal of Molecular Biology*, vol. 48, no. 3, pp. 443–453, 1970.

[120] F. Chierichetti, R. Kumar, S. Pandey, and S. Vassilvitskii, "Finding the jaccard median," in *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*. SIAM, 2010, pp. 293–311.

[121] M. M. Deza and E. Deza, *Encyclopedia of Distances*, 4th ed. Springer Berlin Heidelberg, 2016.

[122] J. Bergstra, D. Yamins, and D. D. Cox, "Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures," in *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ser. ICML'13. JMLR.org, 2013, p. I–115–I–123.

[123] F. P. Miller, A. F. Vandome, and J. McBrewster, *Levenshtein Distance: Information Theory, Computer Science, String (Computer Science), String Metric, Damerau?Levenshtein Distance, Spell Checker, Hamming Distance*. Alpha Press, 2009.

[124] P. H. Sellers, "On the theory and computation of evolutionary distances," *SIAM Journal on Applied Mathematics*, vol. 26, no. 4, pp. 787–793, 1974.

[125] J. Bergstra, D. Yamins, and D. D. Cox, "Hyperopt: A python library for optimizing the hyperparameters of machine learning algorithms," in *Proceedings of the 12th Python in Science Conference*. Citeseer, 2013, pp. 13–20.

[126] S. Putatunda and K. Rama, "A comparative analysis of hyperopt as against other approaches for hyper-parameter optimization of xgboost," in *Proceedings of the 2018 International Conference on Signal Processing and Machine Learning*, ser. SPML '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 6–10. [Online]. Available: https://doi.org/10.1145/3297067.3297080

[127] P. Kampstra, "Beanplot: A boxplot alternative for visual comparison of distributions," *Journal of Statistical Software, Code Snippets*, vol. 28, no. 1, pp. 1–9, 2008. [Online]. Available: https://www.jstatsoft.org/v028/c01

[128] M. Waskom, "mwaskom/seaborn," Sep. 2020. [Online]. Available: https://doi.org/10.5281/zenodo.592845

[129] E. A. Gehan, "A generalized wilcoxon test for comparing arbitrarily singly-censored samples," *Biometrika*, vol. 52, no. 1/2, pp. 203–223, 1965. [Online]. Available: http://www.jstor.org/stable/2333825

[130] I. M. Rodrigues, D. Aloise, and E. R. Fernandes, “Fast: A linear time stack trace alignment heuristic for crash report deduplication,” in *Proceedings of the 19th International Conference on Mining Software Repositories*, in press.

[131] A. Khvorov, R. Vasiliev, G. Chernishev, I. M. Rodrigues, D. Koznov, and N. Povarov, “S3m: Siamese stack (trace) similarity measure,” in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, 2021, pp. 266–270.

[132] S. Batzoglou, “The many faces of sequence alignment,” *Briefings in bioinformatics*, vol. 6, no. 1, pp. 6–22, 2005.

[133] A. Chakraborty and S. Bandyopadhyay, “Fogsaa: Fast optimal global sequence alignment algorithm,” *Scientific reports*, vol. 3, no. 1, pp. 1–9, 2013.

[134] C. Ltd. (2021) Ubuntu bts. [Online]. Available: https://bugs.launchpad.net/

[135] E. Foundation. (2021) Eclipse bts. [Online]. Available: https://bugs.eclipse.org/bugs/

[136] T. A. S. Foundation. (2016) Netbeans bts. [Online]. Available: https://bz.apache.org/netbeans/

[137] A. S. Foundation. (2013) Gnome bts. [Online]. Available: https://bugzilla.gnome.org/

[138] J. A. Hanley and B. J. McNeil, “The meaning and use of the area under a receiver operating characteristic (roc) curve.” *Radiology*, vol. 143, no. 1, pp. 29–36, 1982.

[139] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval.* New York, NY, USA: Cambridge University Press, 2008.

[140] J. Dean and S. Ghemawat, “Mapreduce: Simplified data processing on large clusters,” *Commun. ACM*, vol. 51, no. 1, p. 107–113, jan 2008. [Online]. Available: https://doi.org/10.1145/1327452.1327492

APPENDIX A SUMMARY OF STUDIES ON BUG DEDUPLICATION

In this appendix, we summarize the proposed studies in the literature of bug deduplication in Tables A.1–A.7. On the following, we describe the columns in these tables:

- **Study**.
- **Methodology**. This column contains the methodology approaches employed for method evaluation.
- **Categorical**. It presents how categorical data was used.
- **Textual**. This column describes how studies represented and compared textual data.
- **Additional**. Besides textual and categorical data, it describes which additional data was used for the deduplication.
- **Deduplication**. It provides the final technique to deduplicate reports.
- **Novelty**. This column presents main study novelty.

Study	Methodology	Categorical	Textual	Additional	Deduplication	Novelty
Runeson et al. [76]	Ranking	Product information included as textual data	VSM and log of term frequency	None	Cosine similarity	First work to propose NLP
Jalbert and Weimer [74]	Binary classification	None	VSM and log of term frequency	None	Cosine Similarity	First work to propose a method for binary classification
Wang et al. [9]	Ranking	None	TF-IDF and cosine similarity	TF-IDF and cosine similarity to compare execution traces	Linear Combination	Use of execution traces
Sureka and Jalote [83]	Ranking	None	Character-level of n-grams	None	Number of shared n-grams between the reports	Character-level of n-grams
Sun et al. [81]	Ranking	None	Similarity score is the sum of IDFs of shared terms	None	SVM	SVM combined with 24 features derived by comparing different textual sources.
Prifti et al. [75]	Ranking	None	Centroid vector representing of Buckets	None	Cosine Similarity	Bucket representation
Sun et al. [14]	Ranking	Boolean features of product and component	$BM25F_{EXT}$	None	Linear Combintation	$BM25F_{EXT}$

Table A.1 Summary of bug deduplication literature (Part I).

Study	**Methodology**	**Categorical**	**Textual**	**Additional**	**Deduplication**	**Novelty**
Nguyen et al. [82]	Ranking	None	T-model and Jensen-Shannon divergence; BM25F	None	Linear combination	T-Model
Zhou and Zhang [77]	Ranking	None	Cosine Similarity of TF-IDF; Sum of the IDF and term frequency of each shared term	None	Linear combination	Pairwise ranking loss function for training linear combination weights
Banerjee et al. [4]	Ranking	Boolean features of component and product	LCS	None	Linear combination	LCS
Tian et al. [64]	Binary Classification	Boolean features of product and component comparisons	$BM25F_{EXT}$	None	SVM	They outperforms Jalbert and Weimer [74]
Feng et al. [65]	Binary Classification and Ranking	Boolean features of product and component	TF-IDF; LDA	Comment and reporter profile features are used for the deduplication	Off-the-shelf models	The use of comments and reporter information. They outperformed Tian et al. [64].

Table A.2 Summary of bug deduplication literature (Part II).

Study	Methodology	Categorical	Textual	Additional	Deduplication	Novelty
Banerjee et al. [85]	Ranking	Boolean features of component and product	Centroid vector representing of Buckets; LCS	None	Multi-label classifier that predicts if a similarity score computed by a method is adequate or not for a deduplication	Multi-label classifier; Strategy to merge distinct ranking lists
Lin and Yang [78]	Ranking	None	CFC weighting scheme	None	SVM	CFC weighting scheme
Lazar et al. [63]	Decision-Making	Boolean features of component and product	TakeLab	None	Off-the-shelf models	Features from TakeLab
Aggarwal et al. [66]	Decision-making	Boolean features of component and product	BM25F	BM25F compares report with word lists produced from textbooks and documentations	Off-the-shelf models	Contextual features generated from additional content outside of bug reports.
Zou et al. [86]	Binary classification	One hot encoding of product and component	LDA; Cosine Similarity	None	Similarity score is computed by a linear combination	LDA and classification based on threshold

Table A.3 Summary of bug deduplication literature (Part III).

Study	**Methodology**	**Categorical**	**Textual**	**Additional**	**Deduplication**	**Novelty**
Yang et al. [79]	Ranking	Boolean features of product and component	TF-IDF; Average vector of word embeddings	None	The cosine similarities of TF-IDF representantion and average vector of word embeddings are normalized by categorical data similarity.	Word embeddings
Lin et al. [62]	Ranking	None	CFC weighting scheme; BM25 weighting scheme; Average vector of word embeddings	None	SVM	Features derived from two different weighting schemes and average vector of word embeddings
Hindle et al. [1]	Decision-making; Ranking	Boolean features of component and product	BM25F	BM25F compares report with word lists produced from non-functional requirement terms and set of architecture words	Off-the-shelf models and Cosine Similarity	Contextual features generated from additional content outside of bug reports.

Table A.4 Summary of bug deduplication literature (Part IV).

Study	Methodology	Categorical	Textual	Additional	Deduplication	Novelty
Banerjee et al. [15]	Binary classification and Ranking	One-hot encoding	LCS	Reporter profile	Random forest classifier	Automated framework for bug deduplication in industrial-scale
Budhiraja et al. [87]	Ranking	None	LDA; Average vector of word embeddings.	None	Cosine similarity	Re-ranking using topic distribution similarity and the cosine similarity of embedding average vectors
Budhiraja et al. [88]	Ranking	None	Average vector of word embeddings	None	Cosine Similarity	Investigation of impact of different techniques to learn embeddings on the model performance
Budhiraja et al. [89]	Ranking	None	Average vector of word embeddings	None	MLP	Average vector of word embeddings combined with MLP

Table A.5 Summary of bug deduplication literature (Part V).

Study	Methodology	Categorical	Textual	Additional	Deduplication	Novelty
Deshmukh et al. [68]	Decision-making and Ranking	Distribution representation of the categorical values followed by MLP	LSTM and CNN extract features from summary and description, respectively	None	Cosine similarity and MLP	First work to propose siamese neural networks for bug deduplication
Xie et al. [90]	Decision-making	One-hot encoding of Component	CNN	None	Logistic regression classifier that receives the cosine similarity of CNN outputs and component similarity	Novel architecture
Poddar et al. [19]	Decision-making	None	Attentive Pooling with LSTM	None	MLP	Simultaneously train a model for topic based clustering and bug deduplication
Kukkar et al. [91]	Decision-making and Ranking	None	CNN with holistic and predefined filters.	None	A MLP receives a matrix produced by the cosine similarity of the sentence embeddings.	CNN with holistic and predefined filters and similarity between sentence embeddings

Table A.6 Summary of bug deduplication literature (Part VI).

Study	Methodology	Categorical	Textual	Additional	Deduplication	Novelty
Xiao et al. [69]	Decision-making	Distribution representation of the categorical values followed by MLP	LSTM	None	Manhattan distance between representations generated by a MLP that receives the embedding of their categorical and textual data	Deep learning model for bug deduplication in the after just-in-time retrieval
He et al. [92]	Decision-making	Product and component as considered as textual data	Dual-Channel CNN in which each channel is the textual data of a report	None	MLP	Dual-Channel CNN that jointly extracts features from reports
Cooper et al. [72]	Ranking	None	TF-IDF and Lucene's scoring function	Feature extraction from videos employing SimCLR and LCS algorithms	Linear combination	The first to propose a deep learning method to address bug deduplication through videos.
Rocha and Carvalho [67]	Decision-making and Ranking	Distribution representation of the categorical values followed by MLP	BERTs independently encode summary and description	None	MLP for classification and cosine similarity for similarity measurement	Quintet loss

Table A.7 Summary of bug deduplication literature (Part VII).

APPENDIX B SUMMARY OF STUDIES ON CRASH REPORT DEDUPLICATION

In this appendix, we summarize the proposed studies in the literature of crash report deduplication in Tables B.1 and B.2. On the following, we describe the columns in these tables:

- **Study**.
- **Methodology**. This column presents the evalution methodology approaches employed in the studies.
- **Stack trace similarity**. This column introduces how studies represented and compared stack traces.
- **Deduplication**. It describes how the deduplication was performed.

Study	**Methodology**	**Stack Trace Similarity**	**Deduplication**
Brodie et al. [21]	Method is not evaluated	Variant of NW algorithm in which match(·) depends on frame position and global frequency information	Stack trace similarity
Modani et al. [95]	Binary Classification	Prefix match	Stack trace similarity
Bartz et al. [96]	Binary Classification	Edit distance in which operations depends on module name, sobroutine name, and offset	Logistic regression that receives edit distance score and the features related to categorial data
Dhaliwal et al. [13]	Method is not evaluated	Edit distance	The reports are filtered by the similarity of topmost frame and, then, stack trace similarity is employed for the deduplication
Kim et al. [98]	Binary Classification	Stack traces and buckets are represented as graphs; and the similarity of two graphs is based on the number of shared edges.	Graph similarity
Dang et al. [22]	Clustering	Variant of NW algorithm in which match(·) depends on position information and contains parameters	Stack trace similarity
Lerch and Mezini [93]	Ranking	TF-IDF and Lucene's score function	Stack trace similarity
Koopaei and Hamou-Lhadj [99]	Binary Classification	Stack traces are compared to automatas that are generated for each bucket	Automata

Table B.1 Summary of crash report deduplication literature (Part I).

Study	**Methodology**	**Stack Trace Similarity**	**Deduplication**
Campbell et al. [10]	Clustering	TF-IDF and Lucene's score function	Textual similarity based on stack trace content and environment information; the textual data related to execution environment is tokenized by splitting words that are written in CamelCase
Sabor et al. [94]	Ranking	TF-IDF and Cosine Similarity; only package names are employed	Stack trace similarity.
Moroo et al. [102]	Clustering	Methods proposed by [10] and [22]	Re-ranking using the harmonic mean of two similarity scores
Ebrahimi et al. [100]	Ranking	HMM is trained for each bucket	A probability of stack trace being in a bucket is computed by means of HMM
Khvorov et al. [97]	Ranking	Siamese neural network	Stack trace similarity

Table B.2 Summary of crash report deduplication literature (Part II).

APPENDIX C ADDITIONAL ABLATION STUDY RESULTS REGARDING ARTICLE 2

In this appendix, we expand the ablation study presented in Chapter 5 in which Global Weight, Local Weight, the diff(·) function, and normalization are removed. We depict ΔAUC, ΔMAP, and ΔRR@1 between the original TraceSim and each possible configuration that has not more than two components enabled in Figures C.1–C.8.

The following configurations are not reported:

1. *TraceSim without Global Weight and Local Weight.* In this case, frame weights are always equal to 1. Since the normalization was designed based on variable frame weights, the normalization loses its effectiveness.

2. *TraceSim without Global Weight, Local Weight, and the* diff(·) *Function.* Similarly to the previous configuration, the normalization is not effective because the frame weights are constants.

3. *TraceSim without Global Weight, Local Weight, normalization and the* diff(·) *Function.* This configuration is equivalent to NW algorithm in which the match, mismatch and gap values are set to 1.0, 2.0, and 1.0, respectively.

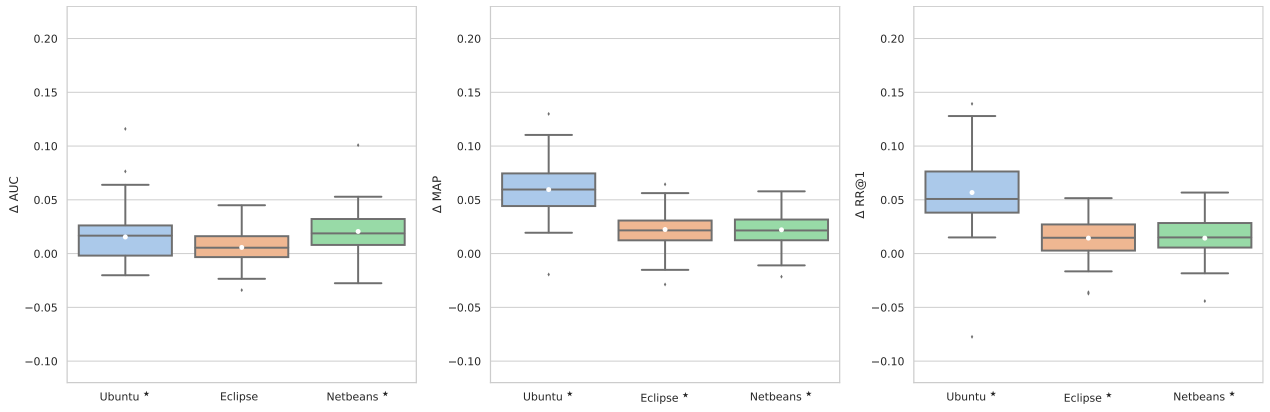


Figure C.1 Distributions of ΔAUC (left), ΔMAP (middle) and ΔRR@1 (right) between full TraceSim and TraceSim *without the* diff(·) *Function and Normalization.*

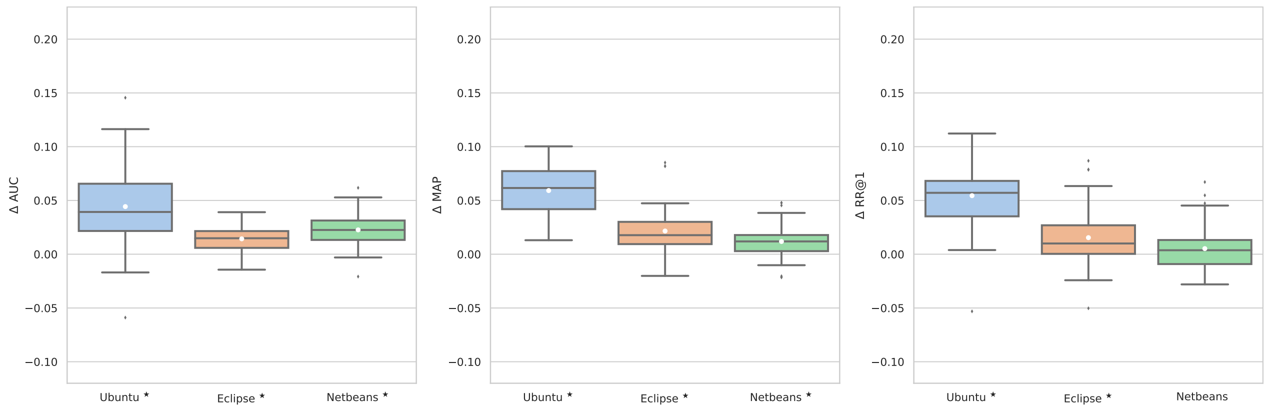


Figure C.2 Distributions of ΔAUC (left), ΔMAP (middle) and ΔRR@1 (right) between full TraceSim and TraceSim *without Global Weight and the* diff(·) Function.

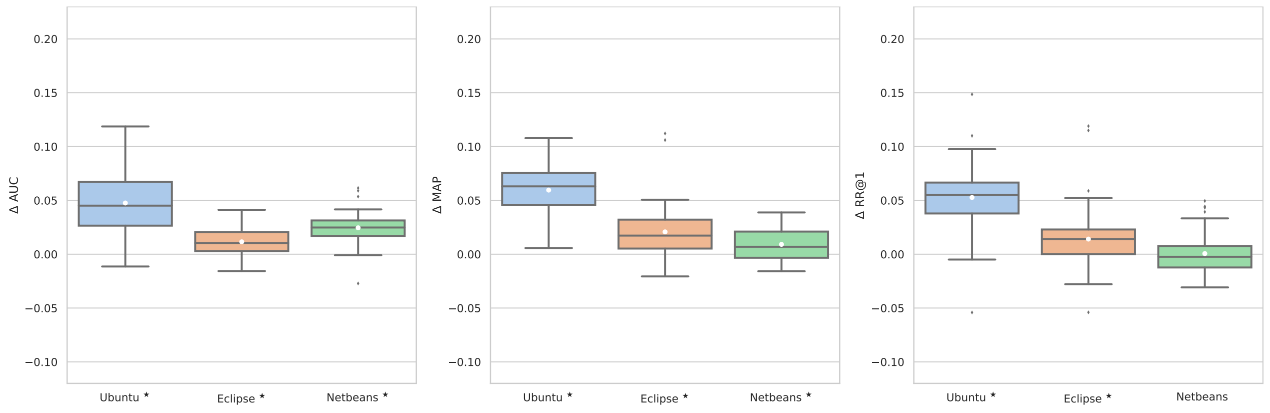


Figure C.3 Distributions of ΔAUC (left), ΔMAP (middle) and ΔRR@1 (right) between full TraceSim and TraceSim *without Global Weight and Normalization.*

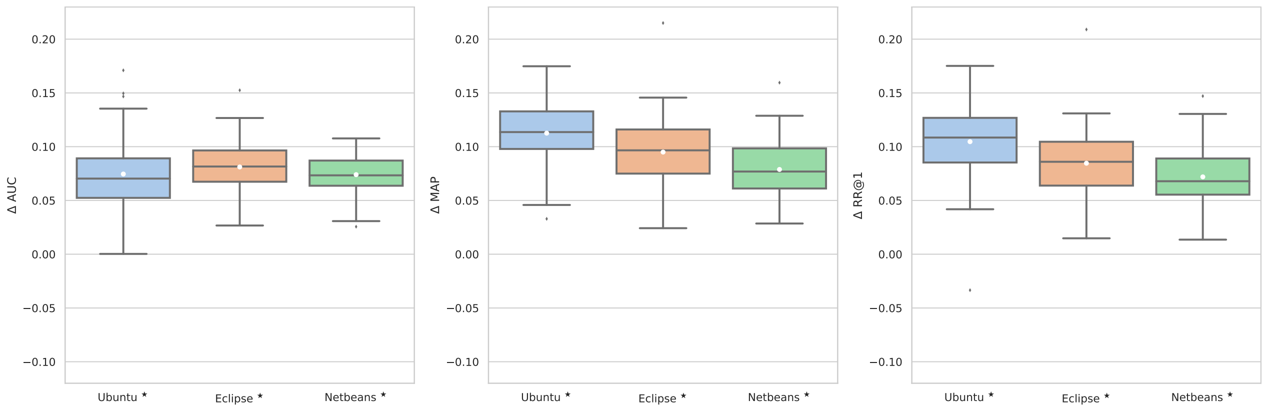


Figure C.4 Distributions of ΔAUC (left), ΔMAP (middle) and ΔRR@1 (right) between full TraceSim and TraceSim *without Global Weight, Local Weight and Normalization.*

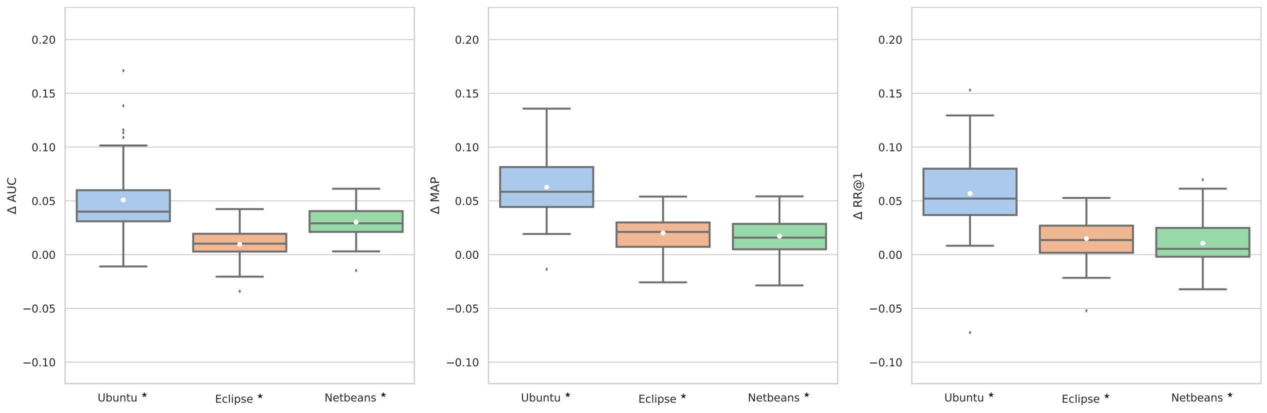


Figure C.5 Distributions of ΔAUC (left), ΔMAP (middle) and ΔRR@1 (right) between full TraceSim and TraceSim *without Global Weight, the* diff(·) *Function, and Normalization.*

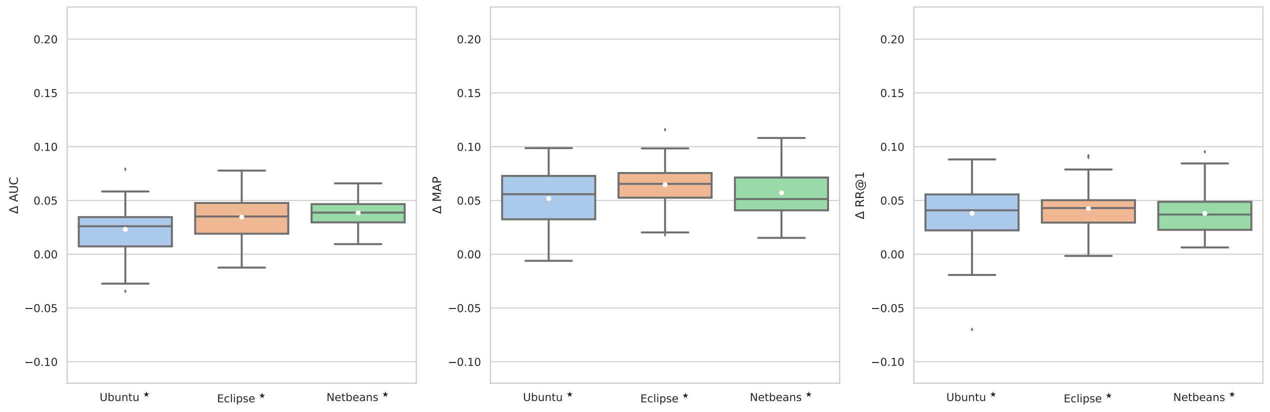


Figure C.6 Distributions of ΔAUC (left), ΔMAP (middle) and ΔRR@1 (right) between full TraceSim and TraceSim *without Local Weight and Normalization*.

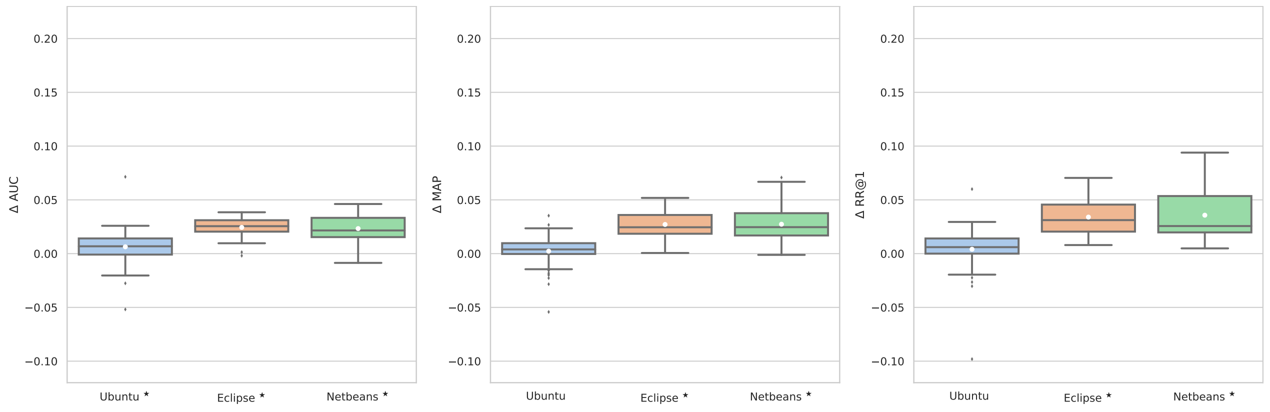


Figure C.7 Distributions of ΔAUC (left), ΔMAP (middle) and ΔRR@1 (right) between full TraceSim and TraceSim *without Local Weight and the* diff(·) Function.

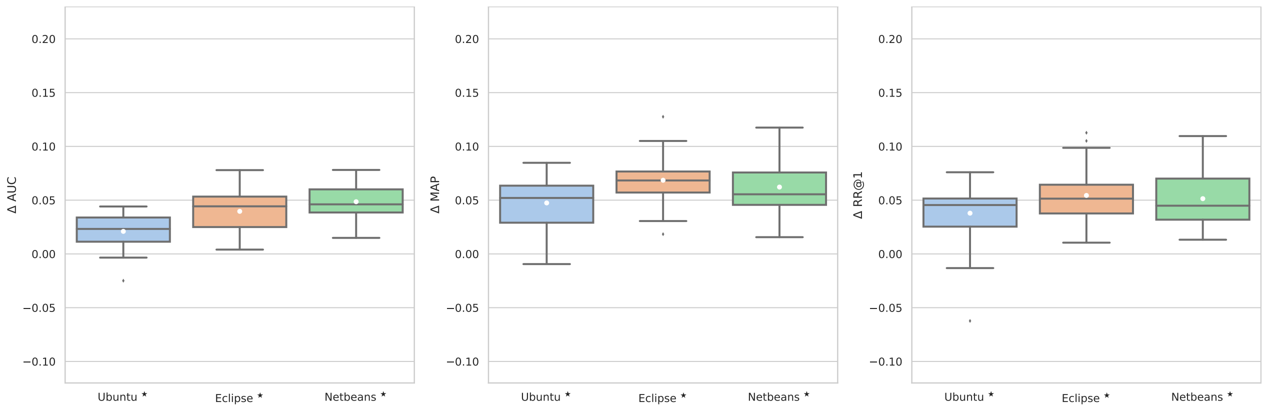


Figure C.8 Distributions of ΔAUC (left), ΔMAP (middle) and ΔRR@1 (right) between full TraceSim and TraceSim *without Local Weight, the* diff(·) *Function and Normalization.*