University of Alberta

A Contextual Approach towards More Accurate Duplicate Bug Report Detection

by

Anahita Alipour

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

©Anahita Alipour
Fall 2013
Edmonton, Alberta

Permission is hereby granted to the University of Alberta Libraries to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only. Where the thesis is converted to, or otherwise made available in digital form, the University of Alberta will advise potential users of the thesis of these terms.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as herein before provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

Abstract

The issue-tracking systems used by software projects contain issues or bugs written by a wide variety of bug reporters, with different levels of knowledge about the system under development. Typically, reporters lack the skills and/or time to search the issue-tracking system for similar issues already reported. Hence, many reports end up referring to the same issue, which effectively makes the bug-report triaging process time consuming and error prone.

Many researchers have approached the bug-deduplication problem using off-the-shelf information-retrieval tools. In this thesis, we extend the state of the art by investigating how contextual information about software-quality attributes, software-architecture terms, and system-development topics can be exploited to improve bug-deduplication. We demonstrate the effectiveness of our contextual bug-deduplication method on the bug repository of Android, Eclipse, Mozilla, and OpenOffice Software Systems. Based on this experience, we conclude that researchers should not ignore the context of the software engineering domain for deduplication.

Acknowledgements

I would like to thank my advisers Dr. Hindle and Dr. Stroulia for their great supervision and helpful advice. I also would like to thank Chengnian Sun, David Lo, Siau-Cheng Khoo, and Jing Jiang for sharing the data-sets related to their work [52] with us.

This work is partially supported by Natural Sciences and Engineering Research Council (NSERC), Alberta Innovates Technology Futures (AITF), and International Business Machines (IBM) corporation.

Table of Contents

1 Introduction **1**
1.1 Bug Deduplication . . . 2
1.2 Contributions . . . 3
1.3 Outline . . . 5

2 Related Work **6**
2.1 Information Retrieval (IR) Techniques . . . 6
2.1.1 IR in Software Engineering . . . 7
2.2 Bug Report Deduplication . . . 8
2.2.1 Approaches Applying IR Techniques Exclusively . . . 8
2.2.2 Stack Traces based Approaches . . . 12
2.2.3 Textual and Categorical Similarity Based Approaches . . . 13
2.2.4 Topic Model Based Approaches . . . 14
2.3 Contextual Bug Report Deduplication . . . 15
2.4 Other Bug Report Related Studies . . . 16

3 The Data Set **19**
3.1 The Lifecycle of Bug Reports . . . 21
3.2 Software-engineering Context in Bug Descriptions . . . 23

4 Methodology **26**
4.1 Preprocessing . . . 26
4.2 Textual Similarity Measurement . . . 28
4.3 Categorical Similarity Measurement . . . 30
4.4 Contextual Similarity Measurement . . . 32
4.5 Combining the Measurements . . . 33
4.6 Prediction . . . 34
4.6.1 Classification . . . 34
4.6.2 Retrieving the List of the Most Similar Candidates . . . 37

5 Case Studies **48**
5.1 Evaluating the Classification-based Retrieval Method . . . 48
5.1.1 Discussion of Findings . . . 55
5.2 Effectiveness of Number of Features . . . 58
5.2.1 Discussion of Findings . . . 58
5.3 Evaluating the List of Candidates . . . 63
5.3.1 Discussion of Findings . . . 65
5.4 Context Matters . . . 65
5.5 Threats to Validity . . . 66

6 Conclusions and Future Work **68**
6.1 Contributions . . . 70
6.2 Future Work . . . 70

Bibliography **72**

List of Tables

2.1 Related Literature on Detecting Duplicate Bug Reports 9

3.1 Details of Datasets 20
3.2 Fields of Interest in Our Research 21
3.3 Examples of duplicate bug reports from Android bug-tracking system. . . . 21

4.1 Examples of Android bug reports before and after preprocessing 28
4.2 Some examples of pairs of the bug reports from Android bug repository with categorical and textual similarity measurements ("textual_categorical" table). 31
4.3 Examples of the NFR contextual features for some of Android bug reports ("table of contextual measures") 33
4.4 Examples of the records in the data-set containing categorical, textual, and contextual measurements for the pairs of Android bug reports. 34
4.5 Examples of pairs of bug reports from Mozilla bug repository with their REP comparisons result and their class (the "similarity_criteria" table) . . . 39
4.6 Examples of pairs of bug report from Mozilla repository with their REP and cosine_sim comparisons for different contexts and their class 40
4.7 Examples of pairs of bug reports from Mozilla repository with their REP and contextual_distance comparisons for different contexts and their class . 41

5.1 Statistical measures resulted by the experiments on Android bug repository including textual, categorical, and contextual data 50
5.2 Statistical measures resulted by the experiments on Eclipse bug repository including textual, categorical, and contextual 51
5.3 Statistical measures resulted by the experiments on Mozilla bug repository including textual, categorical, and contextual data 52
5.4 Statistical measures resulted by the experiments on OpenOffice bug repository including textual, categorical, and contextual data 53
5.5 Examples of predictions made by K-NN algorithm for Android bug repository including textual, categorical, and Labeled-LDA context's data 53
5.6 MAP results for the list of candidates of Android bug repository 63
5.7 MAP results for the list of candidates of Eclipse bug repository 64
5.8 MAP results for the list of candidates of Mozilla bug repository 65
5.9 MAP results for the list of candidates of OpenOffice bug repository 66

List of Figures

3.1 Distribution of Android, Eclipse, Mozilla, and OpenOffice duplicate bug reports into buckets. 20
3.2 Bug lifecycle in Bugzilla [57]. Rounded corner rectangles are the states and the notes represent the resolutions of the bug reports. 22
3.3 Android bug lifecycle. Rounded corner rectangles are the states and the notes represent the resolutions of the bug reports. 23

4.1 Workflow of our methodology. The typical rectangles represent the datasets and the rounded corner rectangles represent the activities. The arrows emerging from the typical rectangles represent the data flows. And, the arrows emerging from the rounded corner rectangles represent the control flows. 27
4.2 Categorical and textual measurements to compare a pair of bug reports [52]. 30
4.3 Overal workflow to retrieve duplicate bug reports 38
4.4 Buckets of the bug reports for a sample repository. 44
4.5 An example of bug report retrieval scenario for a duplicate bug report with ID 7 and evaluating the retrieval method using MAP measure. 45
4.6 An example of bug report retrieval scenario for a duplicate bug report with ID 2 and evaluating the retrieval method using MAP measure. 46

5.1 ROC curves resulted by applying K-NN algorithm on Android reports. . . . 54
5.2 ROC curves resulted by applying C4.5 algorithm on Android reports. 54
5.3 ROC curves resulted by applying K-NN algorithm on Eclipse reports. . . . 54
5.4 ROC curves resulted by applying logistic regression algorithm on Eclipse reports. 55
5.5 ROC curves resulted by applying C4.5 algorithm on Mozilla reports. 55
5.6 ROC curves resulted by applying K-NN algorithm on Mozilla reports. . . . 56
5.7 ROC curves resulted by applying C4.5 algorithm on OpenOffice reports. . . 56
5.8 ROC curves resulted by applying logistic regression algorithm on OpenOffice reports. 57
5.9 Kappa versus number of added features for Android bug repository. The x axis shows the number of features each context adds to the bug reports (which is equal to the number of word lists of the contextual data). The contexts from left to right are no context, architecture, NFR, Random words, LDA, and Labeled-LDA. 59
5.10 AUC versus number of added features for Android bug repository. The x axis shows the number of features each context adds to the bug reports. The contexts from left to right are no context, architecture, NFR, Random words, LDA, and Labeled-LDA. 59
5.11 Kappa versus number of added features for Eclipse bug repository. The x axis shows the number of features each context adds to the bug reports (which is equal to the number of word lists of the contextual data). The contexts from left to right are no context, architecture, NFR, Random words, and LDA. 60

5.12 AUC versus number of added features for Eclipse bug repository. The x axis shows the number of the features each context adds to the bug reports. The contexts from left to right are no context, architecture, NFR, Random words, and LDA. 60
5.13 Kappa versus number of added features for Mozilla bug repository. The x axis shows the number of features each context adds to the bug reports (which is equal to the number of word lists of the contextual data). The contexts from left to right are no context, architecture, NFR, Random words, and LDA. 61
5.14 AUC versus number of added features for Mozilla bug repository. The x axis shows the number of the features each context adds to the bug reports. The contexts from left to right are no context, architecture, NFR, Random words, and LDA. 61
5.15 Kappa versus number of added features for OpenOffice bug repository. The x axis shows the number of features each context adds to the bug reports (which is equal to the number of word lists of the contextual data). The contexts from left to right are no context, architecture, NFR, Random words, and LDA. 62
5.16 AUC versus number of added features for OpenOffice bug repository. The x axis shows the number of the features each context adds to the bug reports. The contexts from left to right are no context, architecture, NFR, Random words, and LDA. 62

List of Symbols

$BM25F$	a textual comparison criterion to compare documents
$TF_Q(q,t)$	a weighted frequency of term t in query q
W_Q	an intermediary variable to measure the similarity of two documents by $BM25F$
$TF_D(d,t)$	a weighted frequency of term t in document d
k_1	a constant specifying the contribution of TF_D in comparing two documents
k_3	a constant specifying the contribution of TF_Q in comparing two documents
b_f	a constant specifying the contribution of the textual field f in computing the similarity of two documents
w_f	a constant specifying the contribution of field f in computing the similarity of two documents
$Pr(a)$	observed correlation among the raters when classifying
$Pr(e)$	the probability of chance agreement among the raters when classifying
Ci_j	j_{th} contextual features of the i_{th} bug report in comparison
B_i	i_{th} bug report in comparison
Q	number of duplicate bug reports under study
$AvgP$	average precision
$p(k)$	precision at the cut-off k

Chapter 1

Introduction

As new software systems are getting larger and more complex every day, software bugs are an inevitable phenomenon. Software development is an evolutionary process where after the first release, bug report submissions by the users and testers come through. Bugs arise during different phases of software development, from inception to transition. They occur for a variety of reasons, ranging from ill-defined specifications, to carelessness, to a programmers misunderstanding of the problem, technical issues, non-functional qualities, corner cases, etc. [37, 21]. Also, software bugs are considerably expensive. Existing research indicates that software bugs cost United States, billions of dollars per year [55, 56].

Recognizing bugs as a "fact of life", many software projects provide methods for users to report bugs, and to store these bug/issue reports in a bug-tracker (or issue-tracking) system. The issue-tracking systems like Bugzilla [50] and Google's issue-tracker [5] enable the users and testers to report their findings in a unified environment. These systems enable the reporters to specify a set of features for the bug reports, such as the type of the bug report (defect or feature request), the component in the system the report belongs to, the product the report is about, etc. Then, the developers will select the reported bugs considering some of their features. The selected bug reports are handled with respect to their priority and eventually closed. The issue-tracking systems also provide users with the facility of tracking the status of bug reports.

Addressing bug reports frequently accounts for the majority of effort spent in the maintenance phase of a software project's life-cycle. This is why, researchers have been trying to enhance the bug-tracking systems to facilitate the bug-fixing process [27, 8].

On the other hand, these advantages come with a notable drawback. According to the large number of users of the software systems, defect reporting process is not very systematic. As a result, the users may report defects that already exist in the bug-tracking

system. These bug reports are called "duplicates". The word duplicate may also represent the bug reports referring to different bugs in the system that are caused by the same software defect. Researchers have addressed several reasons for duplicate bug reports [8]: inexperienced users, poor search feature of bug-trackers, and intentional/accidental re-submissions for reporting duplicate bugs, etc.

Hence, there is always need for inspection to detect whether a newly reported defect has been reported before. If the incoming report is a new bug, then it should be assigned to the responsible developer and if the bug report is a duplicate, the report will be classified as a duplicate and attached to the original "master" report. This process is referred to as triaging.

1.1 Bug Deduplication

Identifying duplicate bug reports is of great importance since it can save time and effort of developers. Recently, many researchers like Bettenburg *et al.* [8] have focused on this problem. Here are some of the important motivations for detecting duplicate bug reports:

- Duplicate bug reports may be assigned to different developers by mistake which results in wasting developers' time and effort.
- In addition, when a bug report gets fixed, addressing the duplicates as independent defects is a waste of time.
- Finally, identifying duplicate bug reports can also be helpful in fixing the bugs, since some of the bug reports may provide more useful descriptions than their duplicates [8]

Currently, detecting duplicate bug reports is usually done manually by the triagers. When the number of daily reported bugs for a popular software is taken into consideration, manually triaging takes a significant amount of time and the results are unlikely to be complete. For instance, in Eclipse, two person-hours are daily being spent on bug triaging [3]. Also, Mozilla reported in 2005 that "everyday, almost 300 bugs appear that need triaging" [2].

A number of studies have attempted to address this issue by automating bug-report deduplication. To that end, various bug-report similarity measurements have been proposed, concentrating primarily on the textual features of the bug reports, and utilizing natural-language processing (NLP) techniques to do textual comparison [53, 46, 22, 36, 54]. Some

of these studies also exploit categorical features extracted from the basic properties of the bug reports (i.e. *component, version, priority, etc.*) [52, 26].

Some of these studies result in a method that automatically filters duplicate reports from reaching triagers [26]. While, some other techniques provide a list of similar bug reports to each incoming report. Accordingly, rather than checking against the entire collection of bug reports the triager could first inspect the top-k most similar bug reports returned by this method [59, 54, 52].

1.2 Contributions

In this work, we introduce a new approach for improving the accuracy of detecting duplicate bug reports of a software system. For the purpose of bug report similarity measurement, we make use of textual and categorical features of the bug reports as well as their contextual characteristics. In terms of automating the triaging process, our approach provides the triagers with a list of the most similar bug reports (sorted based on a similarity measurement method) to every incoming report. So, the triagers can make the final decision about the actual duplicates.

Our approach exploits domain knowledge, about the software-engineering process in general and the system specifically, to improve bug-report deduplication. Essentially, rather than naively and exclusively applying information-retrieval (IR) tools, we propose to take advantage of our knowledge of the software process and product. Intuitively, we hypothesize that bug reports are likely to refer to software qualities, i.e., non-functional requirements (possibly being desired but not met), or software functionalities (linked to architectural components responsible for implementing them). Thus, we utilize a few software dictionaries and word lists representing software functional and non-functional requirements contexts, exploited by prior research, to extract the context implicit in each bug report. To that end, we compare the bug reports to the contextual word lists and we record the comparison results as new features for the bug reports, in addition to the primitive textual and categorical features of the bug reports such as *description, component, type, priority, etc.* Then, we utilize this extended set of bug-report features to compare the bug reports and detect duplicates. Through our experiments, we demonstrate that the use of contextual features improves bug-deduplication performance. Also, we investigate the effect of the number of added features on bug-deduplication. Finally, we propose a set of most similar reports for each incoming bug report to assist the tiragers in identifying the duplicates.

We apply our approach on four large bug repositories from the Android, Eclipse, Mozilla, and OpenOffice projects. In this research, we are taking advantage of five different contextual word lists to study the effect of various software engineering contexts on the accuracy of duplicate bug-report detection. These word lists include: architectural words [19], software Non-Functional Requirements words [24], topic words extracted applying Latent Dirichlet Allocation (LDA) method [20], topic words extracted by Labeled-LDA method [20], and random English dictionary words (as a control).

To evaluate our approach, several well-known machine-learning classifiers are applied. To validate the retrieval approach we employed 10-fold cross validation. We indicate that our method results in up to 11.5% and 41/% relative improvements in accuracy and Kappa measures respectively (over the Sun et al.'s work [52]). Additionally, we take advantage of another evaluation method called Mean Average Precision (MAP) measure to assess the quality of the list of the most similar bug reports returned by our approach.

This work makes the following contributions:

1. We propose the use of domain knowledge about the software process and products to improve the bug-deduplication performance. The previous studies in this area either focus on the textual [46], categorical, stack trace data [59], or a combination of them [26] but not the context of the bug reports. We systematically investigate the effect of considering different contextual features on the accuracy of bug-report deduplication (by exploiting variant sets of contextual and non-contextual features in our experiments).

2. We posit a new evaluation methodology for bug-report deduplication (by applying the machine learning classifiers), that improves the methodology of Sun et al. [52] by considering true-negative duplicate cases as well.

3. We demonstrate that our contextual similarity measurement approach was able to improve the accuracy of duplicate bug-report detection by up to 11.5%, the Kappa measure by up to 41%, and the AUC measure by up to 16.8% over the Sun *et al.*'s method [52] that only makes use of textual and categorical features when comparing the bug reports.

4. Finally, we propose some new bug report similarity criteria based on the REP function introduced by Sun *et al.* [52] and the factors generated by the logistic regression classification algorithm. As a result, we suggest a list of the most similar bug reports

for each incoming report. We show that this criteria succeeded to improve the quality of the list of candidate duplicates and consequently the MAP measure by 7.8-9.5% over Sun *et al.*'s approach [52].

1.3 Outline

The rest of this thesis is organized as follows. Chapter 2 presents an overview of the related work. We discuss the frequently used IR techniques in the area of software engineering. We outline some of the software engineering challenges in which IR techniques are commonly applied as of writing this thesis. Moreover, we address the existing research concentrating on the issue of detecting duplicate bug reports. Finally, we briefly discuss some other studies on the bug reports that could potentially result in facilitating the bug report triaging and duplicate report detection processes.

Chapter 3 provides detailed information about the data-sets exploited in our experiments. These data-sets include the bug reports from the bug repositories utilized in this study as well as the contextual word collections adapted from other existing studies.

In Chapter 4 we discuss our approach for detecting duplicate bug reports. In this Chapter, firstly, we explain our data preprocessing method. Secondly, we describe the textual and categorical similarity measurements we benefit from in this study. Thirdly, we explain our contextual similarity measurement method. Then, we present two different duplicate bug report retrieval techniques applied in our work. Finally, we discuss the evaluation techniques exploited in this thesis.

In Chapter 5, we report the results of our experiments on four different real world bug repositories including Android, Eclipse, Mozilla, and OpenOffice bug reports. Then, we analyze and discuss these results as well as the treats to validity of our approach.

Finally, we conclude in Chapter 6, summarizing the substantial points and contributions made in this thesis and propose some potential future work.

Chapter 2

Related Work

In this Chapter we outline the research literature related to this thesis which is organized in two sections. First, we review the Information Retrieval (IR) techniques and the cases of application of these techniques in software engineering challenges. Then, we discuss the existing work on the automation of the bug report deduplication process and some other bug report related studies which could help in this process.

2.1 Information Retrieval (IR) Techniques

Information retrieval is the activity of obtaining the needed information from a collection of information resources. IR techniques are applied on a broad spectrum of different scopes from image retrieval to web search. Here, we indicate some of the most frequently used IR techniques.

Vector Space Model (VSM) is one of the tools exploited repeatedly in information retrieval. This model is a mathematical representation of text documents introduced by Salton *et al.* [48]. This model is commonly utilized for the purpose of comparing textual queries or documents. One of the outstanding methods of forming a weight-vector out of a text is the Term Frequency-Inverse Document Frequency (TF-IDF) [47]. TF-IDF is a weighting factor which denotes how important a word is to a document in a repository of documents. The basic formulas for the TF-IDF are as follows:

$$tf(t,d) = 0.5 + \frac{0.5 * f(t.d)}{max\{f(w,d) : w \in d\}} \tag{2.1}$$

$$idf(t,D) = log\frac{|D|}{|\{d : D : t \in d\}|} \tag{2.2}$$

$$tf - ifd(t,d,D) = tf(t,d) * idf(t,D) \tag{2.3}$$

Where $f(t, d)$ is the frequency of the term t in the document d. While, $idf(t, D)$ shows if the term t is common across the documents. The $idf(t, D)$ divides the total number of the documents by the number of documents containing term t.

To compare the resulting weighted vectors, several methods are proposed such as Jaccard and cosine similarity [47]. There are also plenty of other information retrieval techniques. We explain some of the commonly-used ones as follows.

Robertson *et al.* [45] have introduced a probabilistic retrieval model called BM25 including the following variables: within-document term frequency, document length, and within-query term frequency. This approach has shown remarkable improvement in performance. Later, Robertson *et al.* [44] have extended this approach by adding the calculation of combination of term frequencies prior to the weighting phase. This extension has made the BM25 retrieval method more simple and interpretable with more computation speed and higher performance.

Ganter *et al.* [17] have proposed the Formal Concept Analysis (FCA) technique which is a method of deriving a concept hierarchy from a set of objects and their characteristics. Additionally, Dumais *et al.* [15] have proposed a generative probabilistic model for sets of discrete data called Latent Semantic Indexing (LSI). This method aims to identify patterns in the relationship between the words and concepts included in a collection of text documents. Moreover, Blei *et al.* [11] have presented Latent Dirichlet Allocation (LDA) which is a generative model for documents in which each document is related to a group of topics. The authors presented a convexity-based variational approach for inference and demonstrated that it is a fast algorithm with reasonable performance.

2.1.1 IR in Software Engineering

Information retrieval techniques are frequently applied to resolve the software engineering problems. These techniques pertain to the maintenance and evolution phases of the software life-cycle. These techniques are exploited for variant issues including feature/concept location, fault prediction, developer identification, comprehension, impact analysis, traceability links, and refactoring [9]. Here we outline some of the existing research in this area.

Aversano *et al.* [4] have proposed a method to predict bug-introducing changes using machine learners. In this approach, the software changes are represented as elements of an n-dimensional space that can be used as a feature vector to train a classifier. Also, Zhao *et al.* [60] have presented a static and non-interactive approach to locate features. Their method combines the vector space information retrieval model and static program analysis.

Maletic *et al.* [32] introduce a system called PROCSSI using LSI to identify semantic similarities between pieces of source code. The result is employed to cluster the software components. The authors present a model encompassing structural information to assist in the code comprehension task. Regarding the problem of comprehension, Kuhn *et al.* [28] have applied LSI to calculate the similarity among the software artifacts and then clustered them. These clusters assist the developers to get familiar with the system at hand within a reasonable amount of time. In addition, Marcus *et al.* [33] have used LSI to map the concepts expressed by the programmers (in queries) to the relevant parts in the source code. Their method is built upon finding semantic similarities between the queries and modules of the software.

Maskeri *et al.* [34] have applied LDA in the context of comprehension and extracted topics from the source code. Besides, Hindle *et al.* [23] have proposed and implemented a labeled topic extraction method based on labeling the extracted topics (from commit log repositories) using non-functional requirement concepts. Hindle *et al.*'s method is based on LDA topic extraction technique. They have selected the non-functional requirements concept as they believe these concepts apply across many software systems. Additionally, Poshyvanyk *et al.* [40] have applied the FCA, LSI, and LDA techniques in order to locate the concepts in the source code. They have also defined some novel IR based metrics (exploiting LSI) to measure the conceptual coupling of the classes in the object oriented programs [41]. This method is based on the textual information shared between the modules of the source code.

2.2 Bug Report Deduplication

According to the necessity of automating the duplicate bug report detection process, several researchers have studied this issue. Almost, all of the existing studies in this scope benefit from IR techniques; and each one is trying to improve the state-of-the-art. The bug report deduplication approaches reviewed in this section could be divided into four groups. These groups are illustrated in Table 2.1.

2.2.1 Approaches Applying IR Techniques Exclusively

Runeson *et al.* [46] have presented a method in which only the natural language processing techniques are utilized to identify duplicate bug reports. In this approach, after processing the textual features of bug reports (tokenizing, stemming, and stop words removal), the bug reports are converted into weight vectors using the following weighting formula for each

Table 2.1: Related Literature on Detecting Duplicate Bug Reports

	Author	**Comparison technique**	**Retrieval Technique**	**Evaluation**
Approaches Applying IR Techniques Exclusively	Runeson *et al.* [46]	applying vector space model and cosine similarity metric. similarity metric considering the time frames	list of candidate duplicates	recall rate
	Sun *et al.* [53]	applying SVM to predict duplicates based on textual comparison metrics	list of candidate duplicates	recall rate
	Nagwani *et al.* [36]	applying vector space model and cosine similarity metric to specify duplicates based on a specific threshold	automatic filtering	recall and precision
	Sureka *et al.* [54]	constructing the character n-grams of description and title of the reports and comparing them based on the number of shared character n-grams	list of candidate duplicates	recall rate
	Hiew [22]	applying vector space model, cosine similarity metric, and clustering to identify duplicates based on a specific threshold	list of candidate duplicates	recall and precision
Stack Traces based Approaches	Wang *et al.* [59]	comparing bug reports textually using TF-IDF and cosine similarity metrics as well as execution information and combining these metrics	list of candidate duplicates	recall rate
Textual and Categorical Similarity based Approaches.	Jalbert *et al.* [26]	applying vector space model, cosine similarity metric, using surface features, and clustering the bug reports	list of candidate duplicates	recall rate and Area Under the ROC Curve (AUC)
	Sun *et al.* [52]	applying a set of 7 comparisons including BM25F and categorical similarity metrics	list of candidate duplicates	recall rate and Mean Reciprocal Rank (MRR)
Topic Model [10] based Approaches	Nguyen *et al.* [38]	applying BM25F, and LDA based topic extraction similarity metric and combining the metrics using Ensembled Averaging	list of candidate duplicates	recall rate

term: $weight = 1 + log(frequency)$ in which $frequeny$ is the frequency of the term in a document. For comparing two bug reports, the cosine similarity metric is applied. Furthermore, the authors have considered the time frames when comparing the bug reports.

To retrieve the duplicates, a few top similar reports to any incoming bug report are provided to the triager to make the final decision about the actual duplicates. The authors have performed their experiments on defects from Sony Ericsson software project. Although this method was able to identify only 40% of duplicate reports, they have ended up with the conclusion that $2/3$ of the duplicates can possibly be found using the NLP techniques. Also, the authors have interviewed a tester and a team of analysts that have utilized this technique as an implemented tool. As reported by the authors, all of the interviewees had found this tool helpful and time-saving.

Nagwani *et al.* [36] have proposed an object oriented similarity measurement method to identify duplicate and similar bug reports. The authors call the two bug reports "similar" when the same implementation resolves both of them. In contrast, they call two bug reports "duplicate" when they report the same problem in different sentences. In this approach, each bug report object includes 3 main features (summary, description, and comments). They suggest the weight of these properties for a given bug report as $W = X * S_{summary} + Y * S_{description} + Z * S_{comments}$ where W represents the weight of the bug report; and $S_{summary}$, $S_{description}$, and $S_{comments}$ denote the similarity measure for the summary, description, and comments. X, Y, and Z are the weights for the preceding features respectively. After converting a bug report to an object, these weights are calculated and the textual weighted similarity functions are applied on these objects. Consequently, based on some predefined thresholds, the similar and duplicate bug reports are identified. If the similarity thresholds for all the features (description, summary, and comments) are met for two particular bug reports, the authors call them duplicates. If some of the thresholds are satisfied, the bug reports are classified as similar.

Hiew *et al.* [22] have proposed a model of existing bug reports in the repository and a method in which incoming bug reports are compared to the existing ones textually. Applying this method, some of the incoming bug reports are recognized as duplicates and sent to triager who should make the final decision about them. In this approach, any incoming report is converted to a weight vector in which the terms are weighted exploiting TF-IDF technique. Then, the weight vectors are compared to the centroid in the above mentioned model, utilizing the cosine similarity metric. If the result of these comparisons exceed a specific threshold, the incoming report is classified as a duplicate. Finally, the existing clusters and centroids are updated when the incoming bug reports are added to the repository.

The experiments are performed on a subset of the bug reports from Firefox, Eclipse, Apache, and Fedora software projects. This approach has achieved the best results for the

Firefox bug repository for which 29% precision and 50% recall is acquired. The authors have also conducted a study, taking advantage of human participants as triagers applying the above-mentioned approach. This experiment resulted in detecting duplicate bug reports more accurately within less amount of time in comparison to the case of absence of this method.

Sureka *et al.* [54] have proposed a method exclusively utilizing textual features of the bug reports. Like some of the above mentioned approaches, this method provides the triager with the top-N similar existing bug reports to a specific report. Then, the triager makes the final decision. The main novelty in this approach is exploiting the character-level representations versus word-level ones. The authors count several advantages for using n-grams[1] [16] over the word-level text mining as follows: low-level text mining is language independent so is more portable across languages; utilizing n-grams is more useful for analyzing noisy text; n-grams are able to match concepts from system messages; n-grams are able to extract super-word features; This method can handle misspelled words; this method is able to match short-forms with their expanded form; the n-gram-based approach is able to match term variations to a common root; this approach is able to match hyphenated phrases. In this approach, first, the character n-grams of size 4 to 10 of the description and title of the two bug reports under investigation are extracted. Then, the overall similarity score between the two bug reports is calculated based on the following parameter: number of shared character n-grams between the two bug reports; number of the character n-grams extracted from the title of one bug report present in the description of the other one. This technique is applied on some Mozilla and Eclipse bug reports. According to what the authors reported, this method could achieve 34% recall for the top-50 results for 1100 randomly selected test cases.

Sun *et al.* [53] have proposed a novel text-based similarity measurement method to detect duplicate bug reports. In their approach, first, the bug reports are processed using IR techniques including tokenizing, stemming, and stop words removal. Then, the duplicate bug reports are organized in *buckets*. A *bucket* is a data structure including one master bug report and zero or more duplicate bug reports. In other words, in each *bucket* all the reports are duplicates of each other and the master is the one reported earlier.

Afterward, some examples of duplicate and non-duplicate pairs of bug reports are extracted from the repository to train a Support Vector Machine (SVM) learner. In the train set, for each pair a rich set of 54 textual features are extracted exploiting description, title,

[1] n-gram is a continuous sequence of n characters from a given text

a combination of them, bigrams of them, and the idf metric. The authors utilize libSVM to train their discriminative model to produce the probability of two bug reports being duplicates of each other. To retrieve the duplicate reports, any incoming bug report is compared to the existing bug reports in the buckets. And, based on the duplicate probability values, a list of candidate duplicates are provided. This approach is applied on three repositories from OpenOffice, Firefox, and Eclipse open-source projects. As the authors expressed, their approach achieved 17-31%, 22-26%, and 34-43% improvement over the state-of-the-art on recall measure for OpenOffice, Firefox, and Eclipse bug repositories respectively.

2.2.2 Stack Traces based Approaches

Wang *et al.* [59] have suggested a technique in which both the textual and execution information of the arriving bug reports are compared against the existing bug reports' textual and execution information features. In this approach, a small list of the most similar bug reports are proposed to the tirager to make the final decision and recognize if the new report is a duplicate. As expressed by the authors, the execution information has the following advantages over the natural language information. Execution information reflects the exact situation of the defect and is not affected by the variety of natural language. Besides, execution information demonstrates the internal abnormal behavior which is not noticed by the reporter.

In this approach, for each incoming bug report, two different similarities are calculated between this report and all the existing ones. The first similarity metric is the Natural-Language-based Similarity (NL-S) in which the summary and description of the bug reports are converted to weight vectors using TF-IDF and compared with each other using cosine similarity metric. The second one is called Execution-information-based Similarities (E-S) in which a vector space model is used to calculate the similarity of the bug reports, based on the execution information. However, in this similarity measurement, only the methods that are invoked during the run are studied without considering how many times each method has been invoked. Also, the canonical signature of each method is counted as one index term. Thus, the weight vectors for the execution information are created using TF-IDF and the similarities are measured by the cosine similarity metric. Finally, a combination of NL-S and E-S contribute in ranking the most similar reports to a particular incoming bug report. The experimental result indicates that this approach is able to detect 67%-93% of duplicate bug reports in the Firefox bug repository.

2.2.3 Textual and Categorical Similarity Based Approaches

Jalbert *et al.* [26] have proposed a technique that automatically classifies and filters arriving duplicate bug reports to save triagers' time. Their classifier combines the surface features of the bug reports (non-textual features such as severity, operating system, and number of associated patches), textual similarity measurements, and graph clustering algorithms to identify duplicate bug reports. This classifier applies a linear regression over the features of the bug reports. Each document is represented by a vector in which each vector is weighted utilizing the following formula $w_i = 3 + 2log_2(freq)$ in which the w_i is the weight of word i in the document and $freq$ is the count of word i in the document. The textual similarity between every two documents is calculated by the cosine similarity metric. The result of this similarity measurement is the basis for inducing a similarity graph. And, a clustering algorithm is applied on the graph. Finally, the surface features are exploited to identify the duplicate reports. The experiments are performed on a subset of Mozilla bug reports. As the authors report, this approach can detect and filter 8% of duplicate reports automatically.

Furthermore, Sun *et al.* [52] introduced an approach in which both textual and categorical features (including product, component, type, priority, and version) are taken into consideration when comparing bug reports. They proposed an extended version of BM25 textual similarity metric [44], called BM25F, to compare long queries such as bug reports descriptions. This metric is based on TF-IDF weighting technique. Moreover, the authors present seven comparison metrics illustrated in Figure 4.2 to compare two bug reports in terms of their textual and categorical characteristics. To combine all these comparisons, the authors have proposed a linear function indicated bellow:

$$REP(d, q) = \sum_{i=1}^{7} \omega_i \times comparison_i$$

in which d and q are two bug reports being compared. $comparison_i$s are the comparisons indicated in Figure 4.2 and ω_i are the weights for each comparison.

Sun *et al.* exploited a subset of bug reports from the repositories at hand as the training set. Using the training set, they have tuned the free parameters in BM25F and REP functions. Additionally, they organized the duplicates of the train set into modules called buckets (explained earlier). In the test phase, every single incoming duplicate bug report is compared to all the existing buckets using REP function. Then, according to the values returned by REP, a sorted list of candidate masters is suggested. The index of the real master

in the list represents the accuracy of this method. To evaluate the preceding approach recall rate@k and a modified version of MAP metrics are exploited.

$$recallrate@k = \frac{N_{detected}}{N_{total}}$$

The above formula measures the ratio of duplicate reports that are successfully detected in the retrieved top-K masters ($N_{detected}$) over all the duplicate reports under investigation (N_{total}). For calculating the MAP measure, considering the fact that each bucket has only one relevant master report, the original MAP function is simplified by Sun *et al.* to the Mean Reciprocal Rank (MRR) [58] measure as follows:

$$MRR(Q) = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{index_i}$$

where $index_i$ is the index where the right master is recognized for the i_{th} duplicate report. And, Q is the number of duplicate reports.

As the authors expressed, they are the first to conduct the duplicate report detection experiments on a large number of bug reports from different software systems. They have utilized Eclipse, Mozilla, and OpenOffice software systems to accomplish their experiments. Finally, they have reported 10-27% improvement in recall rate@k ($1 \leq K \leq 20$) and 17-23% in MAP over the state-of-the-art.

2.2.4 Topic Model Based Approaches

Nguyen *et al.* [38] have proposed a novel technique called DBTM in which both IR-based techniques and topic extraction ones are applied to detect duplicate bug reports. To train the DBTM, the existing bug reports in the repository and their duplication information is utilized. For prediction, DBTM is applied to a new bug report and uses the train parameters to estimate the similarity between the bug report and existing reports in terms of textual features and topics. They have also proposed a novel LDA-based technique called T-Model to extract the topics from the bug reports. The T-Model is trained in train phase in a way that the words in bug reports and the duplication relation among them are used to estimate the topics, the topic properties, and the local topic properties. In the prediction phase, for any new bug report b_{new} the T-Model takes advantage of the trained parameters to find the groups of duplicates G that have the most similarity with b_{new} in terms of topics. This similarity is measured using the following formula:

$$topicsim(b_{new}, G) = \max_{b_i \in G}(topicsim(b_{new}, b_i))$$

in which $topicsim(b_{new}, b_i)$ is the topic proportion similarity between the bug reports b_{new} and b_i.

To measure the textual similarity between the bug reports, BM25F method [52] is exploited. To combine topic-based and textual metrics a machine learning technique called Ensemble Averaging is applied. Below, you can find the equation for calculating y which is the linear combination of the two above mentioned metrics:

$$y = \alpha_1 \times y_1 + \alpha_2 \times y_2$$

In the above function, y_1 and y_2 are textual and topic-based metrics. Also, α_1 and α_2 control the significance of these metrics in the duplicate bug report identification process. These factors satisfy $\alpha_1 + \alpha_2 = 1$. This approach provides a list of top-K similar bug reports for every new report. The authors have performed their experiments on OpenOffice, Eclipse, and Mozilla project bug repositories. And, reported 20% improvement in the accuracy over the state-of-the-art.

2.3 Contextual Bug Report Deduplication

In our previous work [1], we developed a method to identify duplicate bug reports based on their contextual features in addition to their textual and categorical fields. To implement this method, we exploited software contextual data-sets, each consisting of a set of contextual word lists. These contextual data-sets involve software architectural words, software non-functional requirement words, topic words extracted by LDA, topic words extracted by Labeled-LDA, and random English words (as a control). Given these contextual words, we proposed several new features for the bug reports by comparing each contextual word list to the textual features of the bug reports (description and title) using the BM25F metric proposed by Sun *et al.* [52].

To compare the bug reports textually and categorically, we have applied Sun *et al.*'s [52] comparison metrics illustrated in Figure 4.2. As a result, we could exploit all the textual, categorical, and contextual features of the bug reports when comparing them with each other. To retrieve the duplicate bug reports, we created a data-set including pairs of bug reports with their textual, categorical, and contextual features and provided this data-set to the machine learning classifiers to decide whether the two bug reports in each record are

duplicates or not. For the purpose of validation, the 10-fold cross validation technique was utilized.

We conducted our experiments on bug reports from Android bug repository and succeeded to improve the accuracy of duplicate bug report identification by 11.5% over the Sun *et al.*'s approach [52]. We also investigated the influence of the number of added features on the accuracy of the bug report deduplication by applying the random English words context which resulted in a poor performance. These results led us to the conclusion that it is context that improves the deduplication performance and not the number of added features to the bug reports.

In this thesis, we extended the work in the above mentioned paper by applying the machine learning based duplicate report retrieval on Eclipse, Mozilla, and OpenOffice bug repositories in addition to Android bug reports. As a result, we were able to improve the bug report deduplication for all these repositories by up to 0.7% in accuracy, 2% in Kappa and 0.5% in AUC which is not as significant as the improvement achieved for Android repository.

In addition to the work in our paper, we developed a new set of experiments that compare every single bug report to all the existing reports in the repository by the means of three different bug report similarity criteria, i.e. cosine similarity based, Euclidean distance based, and logistic regression based metrics. As a result, for every incoming bug report, a sorted list of candidate duplicates (based on a specific similarity criterion) is provided to the triager to make the final decision about the duplicates of the incoming report. This bug report retrieval method is evaluated by the Mean Average Precision (MAP) metric.

2.4 Other Bug Report Related Studies

Apart from detecting duplicate bug reports, a substantial number of studies have been conducted concentrating on software bug reports. Here we outline a few studies that end up with helpful results for enhancing issue-tracking systems and/or detecting duplicate reports.

Bettenburg *et al.* [7] believe that the current bug-tracking systems have defects causing IR processes be less precise. They have summarized the result of their survey as follows:

- Information related to reproducing steps and stack traces are the most helpful information for fixing the bug reports.
- Duplicate bug reports provide beneficial information to fix the bug reports.
- Bug reporters have difficulties to provide stack traces and reproducing steps.

- The bug reports do not provide the information required by developers very often.

Later, in another study, the authors have conducted an interview with 156 developers and 310 bug reporters from Apache, Mozilla, and Eclipse projects [27]. Regarding the feedback they received from these individuals, the authors have proposed a list of seven recommendations for enhancing the bug-tracking systems. Some of these recommendations are as follows: providing a powerful and simple search engine for bug-tracking systems, proving support for merging bug reports, integrating reputation into user profiles to identify experienced reporters.

Besides, Bettenburg *et al.* [8] believe that not only the duplicate bug reports are not harmful, but also they provide helpful information to fix the defects. To prove this hypothesis, they conduct an empirical study , by exploiting bug reports from Eclipse project, which indicates that duplicate bug reports contain information which is not present in master reports. Based on their experiments, they present the following suggestions to enhance the bug-tracking systems:

- Provide the possibility of merging bug reports.
- Check for resubmission of similar bug reports.
- provide the possibility of renewing not fixed old bug reports.
- The reporters should be encouraged to add more information to an already existing bug report.
- Enhance the search feature of bug-tracking systems.

There are also some other studies that try to facilitate understanding the bug reports. One of these studies is conducted by Lotufo *et al.* [31] who believe that bug reports are not easy to understand since they are constructed from communication between reporters and developers. The authors proposed an approach to summarize the bug reports to develop a better understanding of the information provided in the bug reports.

This summarizer is based upon the model of reading a bug report by a human. The authors believe that a reader would focus on the sentences tha are more important to him. Based on a grounded theory, the authors suppose that readers mostly concentrate on the sentences related to the topics in the title and description of the bug reports. This hypothesis is tested taking advantage of the bug repositories from Mozilla, Debian, Launchpad, and Chrome projects. The results illustrate 12% improvement in the state-of-the-art. For

the purpose of validation, the authors have also applied their summarizing method on a randomly selected reports and asked 58 developers to asses their results . These developers validated the usefulness of this approach.

Anvik *et al.* [3] have presented an approach for semi-automating the developer assignment in bug triaging process. They take advantage of machine learners to recommend a list of candidate developers to the triagers. The machine learning algorithms applied in this approach include Naive Bayes, SVM, and C4.5. To train the classifier, a set of reports labeled with the name of the developer who was either assigned to the report or resolved it. New, unconfirmed, and reopened reports are converted to feature vectors. After the training phase, for every incoming bug report, the machine learner recommends a list of developers who may be qualified based on the reports developers have resolved before. The authors have applied their method on Eclipse and Mozilla bug reports which resulted in 50-64% precision. They have also applied this approach on gcc bug repository which resulted in 6% precision.

Cubranic *et al.* [14] have proposed an approach to cut out the triager and automatically assign the incoming bug reports to developers. The author treat the problem of developer assignment as text classification problem. In the model they presented, each developer is related to a single class of bug reports; and each document is assigned to only one class. A proportion of the bug reports was used as train set that shows the correspondence of each developer to the bug reports he/she has been assigned to. This train set is used to train a Naive Bayes machine learner. In the test phase, the machine learner predicts the class for each bug report in the test set. This method is applied on the bug reports from Eclipse project achieving 30% classification accuracy. The authors proposed that this accuracy could significantly lighten the heavy triaging burden.

Anvik *et al.* [2] presented some statistical information to characterize the data in the software bug repositories of Firefox and Eclipse projects. As the authors reported, the proportions of the reports that can result in a change in the software system to all the reports for Eclipse and Mozilla projects are 58% and 44% respectively. Their work addresses two common challenges in software repositories: difficulty of detecting duplicate reports and assigning incoming bug reports to appropriate developers. According to the outcome of this study, the authors emphasize on the necessity of automating or at least semi-automating the above mentioned procedures. Besides, the authors have investigated the application of the machine learning approaches to assist this automation process.

Chapter 3

The Data Set

As mentioned earlier, four large bug repositories are used in this study. These include: Android, Eclipse, Mozilla, and OpenOffice bug repositories. Android is a Linux-based operating system with several sub-projects. The Android bug repository used in this study involves Android bug reports submitted from November 2007 to September 2012. After filtering unusable bug reports (the bug reports without necessary feature values such as Bug ID), the total number of bug reports is 37536 and 1361 of them are marked as duplicate. The Eclipse, Mozilla, and OpenOffice bug repositories utilized in this study, are adapted from Sun *et al.*'s paper [52]. Eclipse is a popular open source integrated development environment. It can be used to develop applications in Java and some other languages. Eclipse bug repository includes the bug reports reported in year 2008. After filtering unusable bug reports, the total number of reports is 43729 and 2834 of them are marked as duplicate. OpenOffice is a well-known open source rich text editor. OpenOffice contains several sub-projects including a word processor (Writer), a spreadsheet (Calc), a presentation application (Impress), a drawing application (Draw), a formula editor (Math), and a database management application. The OpenOffice bug repository includes 29455 bug reports in which there are 2779 bug reports marked as duplicate. Mozilla is a free software community best known for producing the Firefox web browser. In addition, Mozilla produces Thunderbird, Firefox Mobile, and Bugzilla. The Mozilla bug repository exploited in this study contains 71292 bug reports (after filtering junk bug reports) in which 6049 of them are marked as duplicate.

Table 3.1 shows the statistical details of these bug repositories. The last column in this table reports the number of buckets including duplicate reports in each bug repository. As described in Chapter 2, bucket is a data structure proposed by Sun *et al.* [52] in which all the reports are duplicates of each other and the one submitted earlier than others is called

Table 3.1: Details of Datasets

Dataset	#Bugs	#Duplicates	Period		#Duplicate Including Buckets
			From	To	
Android	37536	1361	2007-11	2012-09	737
Eclipse	43729	2834	2008-01	2008-12	2045
Mozilla	71292	6049	2010-01	2010-12	3790
OpenOffice	29455	2779	2008-01	2010-12	1642

the "master" report. Also, Figure 3.1 illustrates the distribution of duplicate bug reports in the buckets for Android, Eclipse, Mozilla, and OpenOffice repositories.

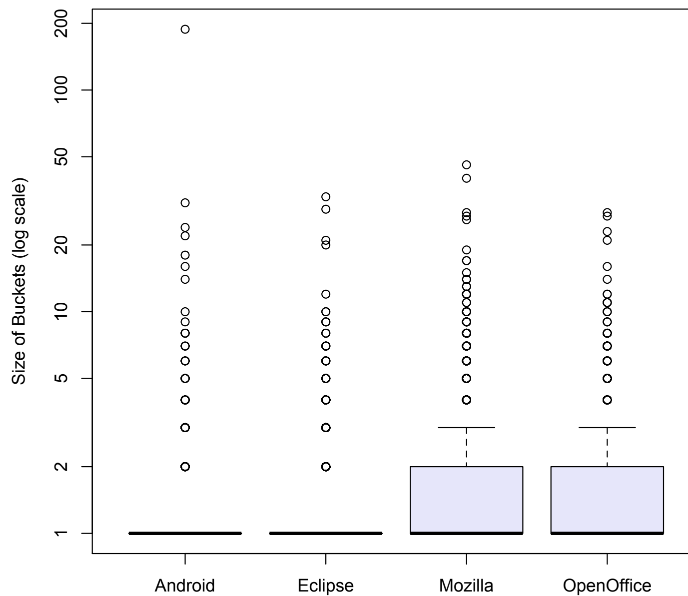

Figure 3.1: Distribution of Android, Eclipse, Mozilla, and OpenOffice duplicate bug reports into buckets.

Although according to the system, the features of the bug reports vary, these features are analogous in general. The fields of interest in our study are demonstrated in Table 3.2. These are the fields we take under consideration for every single bug report in our experiments.

As indicated in Table 3.2, the bug reports exploited in this research include the following features: *description, summary, status, component, priority, type, version, product* and *Merge_ID*. The status feature can have different values including "Duplicate" which means the bug report is recognized as a duplicate report by the triager. To explain the functionality

Table 3.2: Fields of Interest in Our Research

Feature	Feature Type	Description
Summary	Text	A brief description of the problem.
Description	Text	A detailed declaration of the problem which may include reproduction steps and stack traces.
Product	Enum (String)	The product the report is about.
Component	Enum (String)	The component the report is about.
Version	Enum (String)	The version of the product the bug report is about.
Priority	Enum (String)	The priority of the report to be fixed.
Type	Enum (String)	The type of the report: defect, enhancement, task, feature.
Status	Enum (String)	The current status of the bug report: Fixed, Closed, Resolved, Duplicate, etc.
Merge ID	Integer	If the report is a duplicate report, this field shows the ID of the report which the bug report is duplicating.

of Merge_ID we bring the following example. Assume the bug report A is recognized as a duplicate of bug report B by the triager, the Merge_ID feature of A refers to B's Bug ID. We call B the "immediate master" of A. Table 3.3 depicts some examples of duplicate bug reports with their immediate master reports in Android bug-tracking system.

Table 3.3: Examples of duplicate bug reports from Android bug-tracking system.

Pair	ID	Component	Product	Priority	Type	Version	Status	Merge_ID
1	13321	GfxMedia		Medium	Defect		New	
	13323	GfxMedia		Medium	Defect		Duplicate	13321
2	2282	Applications		Medium	Defect	1.5	Released	
	3462	Applications		Medium	Defect		Duplicate	2282
3	14516	Tools		Critical	Defect	4	Released	
	14518	Tools		Critical	Defect	4	Duplicate	14516

Table 3.3 shows examples of pairs of duplicate bug reports from Android and their categorical features. The *Product* field does not have any values in this table since Android bug reports do not have *Product* field. The *Summary* and *Description* fields are not shown in this table.

3.1 The Lifecycle of Bug Reports

As pointed out previously, Eclipse, Mozilla, and OpenOffice bug reports are extracted from the Bugzilla issue-tracker. Bugzilla is a web-based bug-tracking tool, originally developed by Mozilla. It is an open source and free software that has been utilized by numerous software development organizations. The lifetime of a bug report in Bugzilla is as follows [2]:

A newly submitted bug report has the status of either NEW or UNCONFIRMED. When the report is assigned to a developer, the status changes to ASSIGNED. When the report is closed, the status is set to RESOLVED. After the report is verified by the quality assurance team, its status will change to VERIFIED and then CLOSED. There are a few different ways for resolving a bug report. In Bugzilla, these are called resolution. If a bug report resulted in a change in code, its resolution will be FIXED. If it is recognized as a duplicate of an existing report, it will be resolved as DUPLICATE. If the bug is not reproducible, it will be resolved as WORKSFORME. If the report explains a problem that could not be fixed, the report will be resolved as WONTFIX. If the report was not an actual bug, it will be resolved as INVALID. If the report is related to another repository, its resolution status changes to MOVED. A resolved bug report may be opened later with the REOPENED resolution. Figure 3.2 indicates the life-cycle of a bug report in Bugzilla.

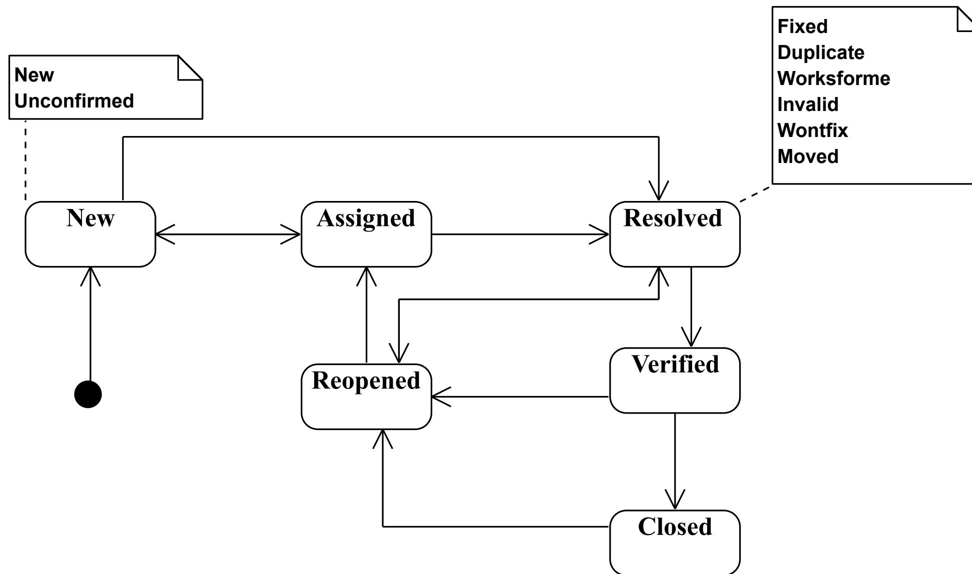

Figure 3.2: Bug lifecycle in Bugzilla [57]. Rounded corner rectangles are the states and the notes represent the resolutions of the bug reports.

Android bug reports are classified differently. They are divided into 4 main groups by the means of their status including New, Open, No-Action, and Resolved [13]. New issues include the bug reports that have not been triaged yet (New) and the bug reports that do not provide sufficient information (NeedsInfo). Open issues include the bug reports that are triaged but not assigned to any developer yet (Unassigned), the bug reports that are being tracked in a separate repository (Reviewed), and the reports that are currently assigned to a developer (Assigned). No-Action issues include spams, reports presenting a question (Question), the reports that are not producible by the developer (Unreproducible),

the reports that describe a behavior which in fact is not a bug (WorkingAsIntended), and the reports that in fact ask for a feature and are not bug reports (Declined). Resolved issues include the bugs that have been fixed but not released yet (FutureRelease), the fixed and released bugs (Released), and reports that are duplicates of existing reports (Duplicate). Figure 3.3 displays the life-cycle of an Android bug report. As Figures 3.2 and 3.3 illustrate, Bugzilla and Android bug reports are handled almost similarly. The only notable difference is that no verification process is addressed for Android bug reports.

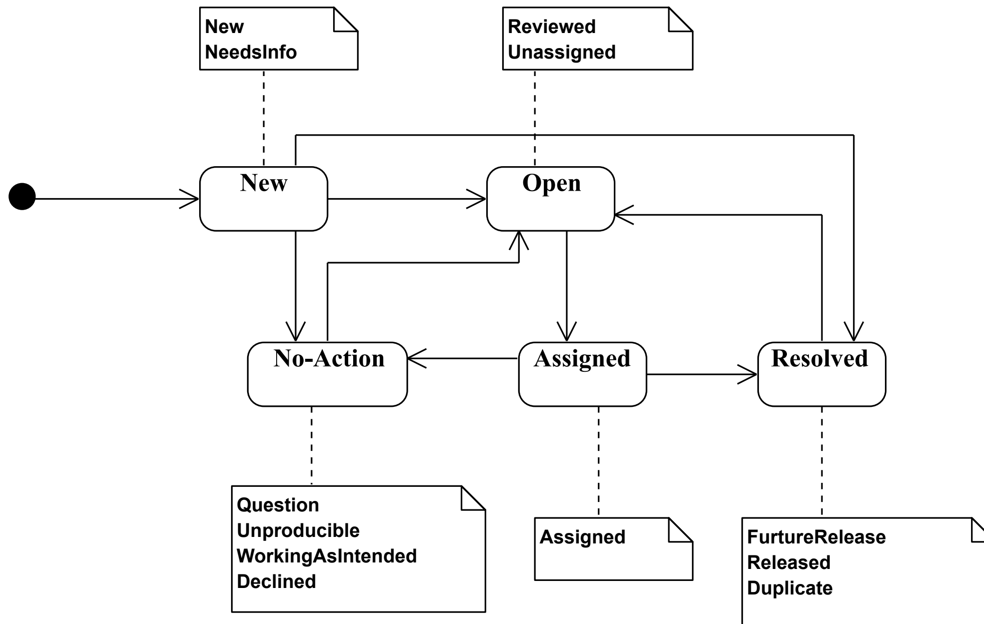

Figure 3.3: Android bug lifecycle. Rounded corner rectangles are the states and the notes represent the resolutions of the bug reports.

3.2 Software-engineering Context in Bug Descriptions

To study the effect of software-engineering contexts on detecting duplicate bug reports, we have taken advantage of different software related contextual data-sets presented as lists of contextual words. These contextual word lists are later exploited to be compared with the bug reports' textual features and specify the contextual characteristics of the bug reports. These contextual word lists elaborate the raw data in the primitive bug reports before being used for the bug report deduplication process. The contextual word lists are discussed below.

- **Architecture words:** For each of the bug repositories a set of architecture words is created. Each set is organized in a few word lists. Each word list represents an architectural layer.

  For Android bug repository, we utilized the word lists provided by Guana *et al.* [19]. They produced a set of Android architecture words to categorize Android bug reports based on architecture. These words are adapted from Android architecture documents and are organized in five word lists (one word list per Android architectural layer [12]) with the following labels: Applications, Framework, Libraries, Runtime, and Kernel.

  For Eclipse bug repository, we have created a set of architecture words that are organized in three word lists (one word list per Eclipse architectural layer) with the following labels: IDE, Plugins, and Workbench.

  For OpenOffice bug repository, the architectural words are organized in four word lists with the following labels: Abstract layer, Applications layer, Framework layer, and Infrastructure layer.

  The architectural words related to Mozilla software system are organized in five word lists. The word lists are labeled as follows: Extensions, UI, Script, XPCOM, and Gecko.

- **Non-Functional Requirement (NFR) words:** Hindle and Ernst *et al.* [24] have proposed a method to automate labeled topic extraction, built upon LDA, from commit-log comments in source control systems. They have labeled the topics from a generalizable cross-project taxonomy, consisting of non-functional requirements such as portability, maintainability, efficiency, etc. They have created a data-set of software NFR words organized in six word lists with the following labels: *Efficiency, Functionality, Maintainability, Portability, Reliability*, and *Usability*. These word lists are exploited as the NFR context words in this thesis.

- **LDA topic words:** LDA represents the topic structure and topic relation among the bug reports. Two duplicate bug reports must address the same technical topics. The topic selection of a bug report is affected by the buggy topics for which the report is intended.

  Han *et al.* [20] have applied both LDA and Labeled-LDA [43] topic analysis models to Android bug reports. We are using their Android HTC LDA topics, organized in 35 word-lists of Android topic words labeled as Topic*i* where *i* ranges from 0 to 34. We

also use their Android HTC topics extracted by Labeled-LDA, organized in 72 lists of words labeled as follows: *3G, alarm, android_market, app, audio, battery, Bluetooth, browser, calculator, calendar, calling, camera, car, compass, contact, CPU, date, dialing, display, download, email, facebook, flash, font, google_earth, google_latitude, google_map, google_navigation, google_translate, google_voice, GPS, gtalk, image, input, IPV6, keyboard, language, location, lock, memory, message, network, notification, picassa, proxy, radio, region, ringtone, rSAP, screen_shot, SD_card, search, setting, signal, SIM_card, synchronize, system, time, touchscreen, twitter, UI, upgrade, USB, video, voicedialing, voicemail, voice_call, voice_recognition, VPN, wifi,* and *youtube*.

For Mozilla, Eclipse, and OpenOffice repositories we have utilized the Vowpal Wabbit online learning tool [29] to extract the topics by LDA. For each of these repositories 20 topic lists is generated, each one including 25 words, using this approach. These word lists are labeled as Topic*i* where *i* ranges from 0 to 19.

- **Random English words:** To investigate the influence of contextual word lists on the accuracy of detecting duplicate bug reports, we created a collection of randomly selected English dictionary words. In other words, we have created this "artificial context" to study if adding noise data to the features of the bug reports can improve deduplication even though the added data does not represent a valid context. This collection is organized in 26 word lists, labeled "a" through "z". In each of these word lists there are 100 random English words starting with the same English letter as the label of the word list.

Chapter 4

Methodology

In this section, we describe our approach of duplicate bug report identification. First of all, we explain our bug report preprocessing approach. Next, we describe our similarity measurement method to compare the bug reports by the means of their textual, categorical and contextual characteristics.

Afterwards, we propose our duplicate bug report retrieval method based on our bug report similarity measurements. Finally, we present our evaluation approach to assess our duplicate bug report retrieval method. Figure 4.1 displays the workflow of our method.

4.1 Preprocessing

After extracting the bug reports, we applied a preprocessing method consisting of the following steps:

1. The first step involves tokenizing the textual fields (*description* and *title*) of the bug reports and removing stop words.

2. The second step involves the organization of the bug reports into a list of buckets. All the bug reports are inserted in the same bucket with their master bug report (specified by their Merge_ID). The bug report with the earliest open time becomes the master report of the bucket.

   Then, the bug reports are converted into a collection of bug-report objects with the following properties: *Bug ID, description, title, status, component, priority, type, product, version, open date, close date,* and optional *master_id*, which is the ID of the bug report which is the master report of the bucket including the current bug report.

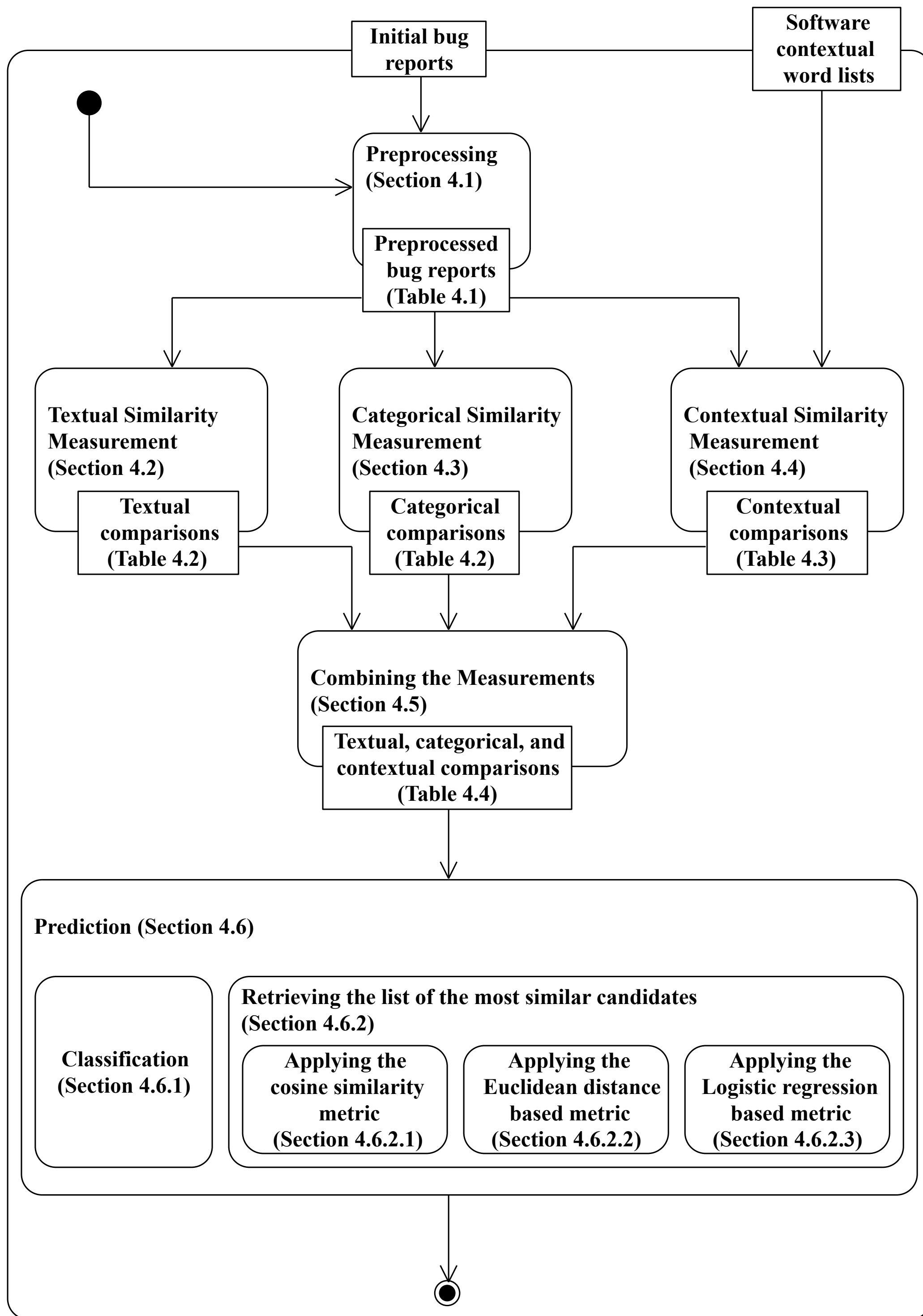

Figure 4.1: Workflow of our methodology. The typical rectangles represent the data-sets and the rounded corner rectangles represent the activities. The arrows emerging from the typical rectangles represent the data flows. And, the arrows emerging from the rounded corner rectangles represent the control flows.

Table 4.1 illustrates some examples of titles of Android bug reports before and after preprocessing.

Table 4.1: Examples of Android bug reports before and after preprocessing

ID	Primitive Title	Processed Title
3063	Bluetooth does not work with Voice Dialer	bluetooth work voice dialer
8152	Need the ability to use voice dial over bluetooth	ability voice dial bluetooth
3029	support for Indian Regional Languages	support indian regional languages
31989	[ICS] Question of Google Maps' location pointer	ics question google maps location pointer

4.2 Textual Similarity Measurement

To measure the textual similarity between a pair of bug reports, we take advantage of the $BM25F$ method introduced by Sun *et al.* [52]. $BM25F$ is designed for short queries, which usually have no duplicate words. For example, the queries in search engines usually include fewer than ten distinct words. However, in the context of duplicate bug report retrieval, each query is a bug report. The query is structured such that it contains a short summary and a long description, and it can sometimes be very long. So, the textual similarity measurement in this thesis is performed based on the extended version of $BM25F$ which is defined as follows [52].

$$BM25F_{ext}(d,q) = \sum_{t \in d \cap q} IDF(t) \times \frac{TF_D(d,t)}{k_1 + TF_D(d,t)} \times W_Q \tag{4.1}$$

$$W_Q = \frac{(k_3+1) \times TF_Q(q,t)}{k_3 + TF_Q(q,t)} \tag{4.2}$$

$$TF_Q(q,t) = \sum_{f=1}^{K} w_f \times occurences(q[f],t) \tag{4.3}$$

$$TF_D(d,t) = \sum_{f=1}^{K} \frac{w_f \times occurences(d[f],t)}{1 - b_f + \frac{b_f \times length_f}{average-length_f}} \tag{4.4}$$

$$IDF(t) = log \frac{N}{N_d} \tag{4.5}$$

In Equation (4.1), for each shared term t between a document d and a query q, the following components are calculated:

- One is the $TF_D(d,t)$ of a term t in a document d which is the aggregation of the importance of t in each textual field of d. In Equation (4.4), for each field f, w_f is the

field weight; $occurrences(d[f], t)$ is the count of the term t in the field f; $length_f$ is the size of the bag $d[f]$; $average - length_f$ is the average size of the bag $d[f]$ across all the documents in corpus; and b_f is a parameter ($0 \leq b_f \leq 1$) that determines the scaling by field length: b_f = 1 corresponds to full length normalization, while b_f = 0 corresponds to term weight not being normalized by the length.

- Another one is W_Q that involves weight from the query computed by $TF_Q(q, t)$. The free parameter $k_3 (k_3 \geq 0)$ is to control the contribution of the query term weighting; for example, if k_3 = 0, then the query term contributes no weight as W_Q becomes always equal to 1. $TF_Q(q, t)$ involves the frequency of a term t in a query q. In Equation (4.3), w_f represents the weight of a textual field f in a query q; and $occurences(q[f], t)$ shows the frequency of a term t in a textual field f of q.
- The last one is the $IDF(t)$ which has an inverse relationship with the frequency of a term t across all the documents in the repository. In Equation (4.5), N_d is the number of documents containing the term t. N is the total number of documents.

In the above mentioned functions, the value for the free variables are adapted from Sun *et al.*'s paper [52]. From now on, we use the $BM25F$ and $BM25F_{ext}$ terms interchangeably.

Here we give an example to show the functionality of the above stated formulas. Assume we have a repository including 3 documents; each one involving only one textual field (summary). The document summaries are as follows: "enhanced low-level bluetooth support", "bluetooth does not work with voice dialer", and "bluetooth phonebook access profile PBAP character problem generating service with well known UUID". We have a query q as follows: "bluetooth phonebook access profile PBAP character problem". To compare the query with the first document in the repository, the BM25F will be calculated as follows.

d = enhanced low-level bluetooth support

q = bluetooth phonebook access profile PBAP character problem

t = bluetooth

k_1 = 2.000

k_3 = 0.382

$w_{summary}$ = 2.980

$b_{summary}$ = 0.703

$$TF_D(d,t) = \sum_{f=1}^{1} \frac{w_f \times 1}{1 - b_f + \frac{b_f \times 5}{8.4}} = 4.168$$

$$TF_Q(q,t) = w_f \times 1 = 2.980$$

$$W_Q = \frac{(k_3 + 1) \times 2.980}{k_3 + 2.980} = 1.225$$

$$IDF(t) = log\frac{3}{3} = 1$$

$$BM25F_{ext} = 1 \times \frac{4.168}{k_1 + 4.168} \times 1.225 = 0.828$$

4.3 Categorical Similarity Measurement

To compare the categorical features of a pair of bug reports, we measure the similarity between them based on their basic features (*component, type, priority, product* and *version*) indicated in Table 3.2. According to Table 3.3, duplicate bug reports have similar categorical features. This motivates the use of categorical features in bug-deduplication. Figure 4.2 indicates the textual and categorical similarity measurement formulas applied in our method. These formulas are adapted from Sun *et al.*'s work [52].

$comparison_1(d_1, d_2) = BM25F(d_1, d_2)$ The comparison unit is unigram.

$comparison_2(d_1, d_2) = BM25F(d_1, d_2)$ The comparison unit is bigram.

$$comparison_3(d_1, d_2) = \begin{cases} 1 & \text{if } d_1.prod = d_2.prod \\ 0 & \text{otherwise} \end{cases}$$

$$comparison_4(d_1, d_2) = \begin{cases} 1 & \text{if } d_1.comp = d_2.comp \\ 0 & \text{otherwise} \end{cases}$$

$$comparison_5(d_1, d_2) = \begin{cases} 1 & \text{if } d_1.type = d_2.type \\ 0 & \text{otherwise} \end{cases}$$

$$comparison_6(d_1, d_2) = \frac{1}{1 + |d_1.prio - d_2.prio|}$$

$$comparison_7(d_1, d_2) = \frac{1}{1 + |d_1.vers - d_2.vers|}$$

Figure 4.2: Categorical and textual measurements to compare a pair of bug reports [52].

The first comparison defined in Figure 4.2 is the textual similarity measurement between two bug reports over the features *title* and *description*, computed by $BM25F$. The

second comparison is similar to the first one, except that the features *title* and *description* are represented in bigrams (a bigram consists of two consecutive words). The remaining five comparisons are categorical comparisons.

Since the $comparison_3$ is comparing the *product* of bug reports, it is not applicable for our Android bug repository as the *product* feature of each Android bug report is not specified. So, we set the value of this feature to 0 for all Android bug reports. Also, regarding Sun *et al.*'s [52] method, we are not considering the version comparison for the bug reports of Eclipse, Mozilla, and OpenOffice bug repositories.

$Comparison_4$ compares the *component* features of the bug reports. The *component* of a bug report may specify an architecture layer or a more specific module within an architectural layer. The value of this measurement is 1 if the two bug reports belong to the same component and 0 otherwise.

$Comparison_5$ compares the *type* of two bug reports, for example in Android bug-tracking system it shows whether they are both "defects" or "enhancements". This comparison has the value of 1 if the two bug reports being compared have the same *type* and 0 otherwise.

$Comparison_6$ and $comparison_7$ compare the *priority* and *version* of the bug reports. These measurements could have values between 0 and 1 (including 1).

The result of these comparisons establishes a data-set including all the pairs of bug reports with the seven comparisons shown in Figure 4.2; and a classification column which reports whether the compared bug reports are duplicates of each other. Table 4.2 demonstrates; a snapshot of this data-set with some examples of pairs of Android bug reports. The value of class column is "dup" if the bug reports are in the same bucket and "non" otherwise.

Table 4.2: Some examples of pairs of the bug reports from Android bug repository with categorical and textual similarity measurements ("textual_categorical" table).

ID1	ID2	$BM25F_{uni}$	$BM25F_{bi}$	Prod cmp	Compo cmp	Type cmp	Prio cmp	Vers cmp	Class
14518	14516	1.4841	0.0000	0	1	1	1.0000	1.0000	dup
29374	3462	0.6282	0.1203	0	0	1	1.0000	1.0000	non
27904	14518	0.1190	0.0000	0	0	1	0.3333	0.1667	non

Regarding the number of bug reports in the bug-tracking systems, a huge number of pairs of bug reports are generated in this step. Consequently, we need to sample the records of the "textual_categorical" tables before running the experiments. Since there are very few pairs of bug reports marked as "dup" in comparison to the number of all the pairs

($\binom{size}{2}$, $size =$ total number of reports in the repository), and we want to create a set of bug report pairs including 20% "dup"s and 80% "non"s, we have selected 4000 "dup" and 16000 "non" pairs of reports randomly. So, per each bug repository we produce 20000 sampled pairs of bug reports.

4.4 Contextual Similarity Measurement

As discussed earlier, in this thesis, we focus on the impact of software contexts on the bug report deduplication process. In this section, we describe our context-based bug report similarity measurement approach.

As mentioned in Chapter 2, most of the previous research on detecting duplicate bug reports has focused on textual similarity measurements and IR techniques. Some approaches consider the categorical features of the bug reports, in addition to the text. Here, we intend to describe our new approach which involves measuring the contextual similarity among the bug reports. We believe this new similarity measurement can help finding the duplicate bug reports more accurately by making the context of a bug report a feature during comparison.

In our method, we take advantage of the software contextual word lists described in Chapter 3. We explain the contribution of context in detail, using the NFR context as an example. As pointed out earlier, this contextual word collection includes six word lists (labeled as *efficiency, functionality, maintainability, portability, reliability,* and *usability*). We consider each of these word lists as a query, and calculate the similarity between each query and every bug report textually (using BM25F). For the case of NFR context, there are six BM25F comparisons for each bug report, which result in six new features for the bug reports. Table 4.3 shows the contextual features resulted by the application of the contextual measurements, using NFR context, for some of Android bug reports. Each column shows the contextual similarity between the bug report and each of the NFR word lists. For example, the bug with the id 29374 seems to be more related to usability, reliability, and efficiency rather than the other NFR contexts.

The same measurement is done for the other contextual word collections as well. At the end, there will be five different contextual tables for Android bug repository as we have five contextual word collections (Labeled-LDA, LDA, NFR, Android architecture, and English random words). And, there are four contextual tables for each of the other bug repositories since they lack the Labeled-LDA contextual words.

Table 4.3: Examples of the NFR contextual features for some of Android bug reports ("table of contextual measures")

Bug ID	Efficiency	Functionality	Maintainability	Portability	Reliability	Usability
3462	3.45	4.57	1.35	0.57	1.53	1.41
2282	2.88	2.51	1.07	3.37	4.53	4.91
29374	3.89	2.52	0.13	0.99	3.20	5.07
27904	2.93	1.03	0.50	0.00	3.36	4.55

4.5 Combining the Measurements

In this phase of the process, we have the "textual_categorical" table for pairs of bug reports (as shown in Table 4.2) and a number of tables reporting contextual similarity measurements, each one according to a different context for individual bug reports, as described in Section 4.4. Here, we describe the combination of the "textual_categorical" table and the "tables of contextual measures".

As our research objective is to understand the impact that contextual analysis may have on bug-deduplication, in this phase, we aim to produce five different tables, each one including pairwise bug-report comparisons across (a) textual features, (b) categorical features, and (c) one set of contextual features. One of these tables, the one corresponding to the "NFR" contextual feature is shown in Table 4.4 for Android bug repository. As demonstrated in this table, the first seven columns are the same as the ones in Table 4.2; they report the similarity measurements between the two bug reports according to the textual and categorical features. Next are two families of six columns each, reporting the NFR contextual features for each of the two bug reports (with Bug $ID1$ and Bug $ID2$ respectively). The second to last column of Table 4.4 reports the contextual similarity of the two bug reports based on these two column families. We consider the contextual features of the two bug reports as value vectors and measure the distance between these two vectors using the cosine similarity measurement. The formula for calculating this similarity is shown below.

$$cosine_sim = \frac{\sum_{i=1}^{n} C1_i \times C2_i}{\sqrt{\sum_{i=1}^{n}(C1_i)^2} \times \sqrt{\sum_{i=1}^{n}(C2_i)^2}} \tag{4.6}$$

In this formula, n is the number of word lists of the contextual data which is equal to the number of contextual features added to each bug report (in the case of NFR, n=6). $C1_i$ and $C2_i$ are the ith contextual features added to the first and second bug reports in the pair respectively. The cosine similarity feature is demonstrated in Table 4.4. This table reveals an example where bug reports with IDs 3462 and 2282, and bug reports with IDs 29374 and 3462 are compared in terms of NFR context. One of the records demonstrates the features

for two bug reports belonging to the same bucket (with class value of "dup"). And, the other one shows a pair of bug reports pertaining to different buckets (with the class value of "non"). Table 4.4 which includes textual, categorical, and the NFR contextual similarity measurements, is called the "NFR all-features_table". Note that there are five different such "all-features" tables, each one corresponding to a different context.

Table 4.4: Examples of the records in the data-set containing categorical, textual, and contextual measurements for the pairs of Android bug reports.

ID1	ID2	cmp_1	...	cmp_7	$Effic_1$	...	$Usability_1$	$Effic_2$	...	$Usability_2$	Cosine_sim	Class
3462	2282	1.52	...	0.29	3.45	...	1.41	2.88	...	4.91	0.73	dup
29374	3462	0.63	...	1.00	3.89	...	5.07	3.45	...	1.41	0.79	non

The class value in the "all-features" tables should be predicted by the machine learning classifiers in next phase. In other words, these classifiers decide whether the two bug reports are duplicates of each other.

4.6 Prediction

In this section, we explain how the duplicate bug reports are retrieved and how these retrieval approaches are evaluated. In the first approach, we take advantage of the machine learning classifiers to predict whether two specific bug reports are duplicates of each other (given their similarity measurements). In the second approach, we compare every incoming bug report to all the existing reports in the repository to provide the triagers with a sorted list of candidate duplicates. As a result, the triagers can make the final decision about the real duplicates.

4.6.1 Classification

In this section, we discuss the application of classifiers on different sets of our comparison metrics for deciding whether a pair of bug reports are duplicates or not. The idea is to use machine learning to amplify the impact of the work of the triager; as the triager identifies duplicate bugs, the classifier learns how to better recognize duplicates and may suggest candidates to the triager and thus simplify his/her task.

To retrieve the duplicate bug reports we are taking advantage of well-known machine learning classification algorithms. In each experiment, a table including pairs of bug reports with a particular combination of similarity metrics (i.e. textual, categorical, and contextual features) is passed to the classifiers. Each "all-features" table (described earlier) includes all the inputs necessary for our classifiers. The classifier should decide about the class

column's value for each pair of bug reports. In other words, given any pair of bug reports, the classifier should decide if the pair is a "dup" (the bug reports in the pair are in the same bucket) or a "non" (the bug reports in the pair are not in the same bucket) based on some combination of the similarity columns of the table. To avoid over fitting during training and evaluation, we use the 10-fold cross validation technique

The classifiers we use are implemented by Weka (Weikato Environment for Knowledge Analysis) [25]. Weka is a well-known tool of machine learning implemented in Java. This tool supports numerous data mining tasks such as preprocessing, clustering, classification, regression, visualization, and feature selection. Weka accepts data in a flat file in which each instance has a fixed number of attributes.

The machine learning classifiers applied in this approach are as follows. The 0-R algorithm is utilized to establish the baseline. The other applied algorithms are C4.5, K-NN (K Nearest Neighbours), Logistic Regression, and Naive Bayes. K-NN tends to perform well with many features, but as well if K-NN works it implies that the input data has a fundamentally simple structure that is exploitable by distance metrics. Here we provide an overview of these classifiers as follows.

- **0-R** is the simplest classification method. This algorithm simply predicts the majority class and does not provide any prediction power. However, it provides a baseline for the performance of the other classification algorithms. We provide an example of performance of this algorithm using Table 3.3.

  This table indicates some examples of pairs of bug reports from Android bug repository accompanied by their textual and categorical features and their class. For this table, the 0-R algorithm will provide the accuracy of 66.66% since it predicts "non" for all the instances.

- **K-NN** is an instance based or lazy learning method for classifying objects based upon the closest training examples [51]. This algorithm is sensitive to the local structure of the data. The training examples consist of multidimensional vectors of features including label (class). In the classification phase, unlabeled feature vectors are classified by assigning the most frequent label among the top k closest training instances. Finally, the instance is assigned to the most common class among its k nearest neighbors. Usually, the Euclidean distance is used as the distance metric in K-NN algorithm. K is a small positive integer. In the case K=1, the instance is assigned to the class of its nearest neighbor. In our experiments, K is set to 1.

A disadvantage of this algorithm is ineffective classifying of skewed data. In other words, the instances of a frequent class control the prediction of the new instances. To overcome this drawback, some solutions are proposed such as weighting the classification and abstraction in data representation.

- **Naive Bayes** is a probabilistic classifier based upon Bayes theorem [6] . This classifier assumes that the presence or absence of a particular feature is not related to the presence or absence of other features (considering that instances in train and test set are vectors of features). For some probability models, this classifier performs effectively. This algorithm needs a small amount of training data to predict the classes. Since the variables are considered independent, only the mean and variance of the variables should be calculated. Based on Bayes theorem, the probability model for a classifier is as follows.

$$P(C|F_1, ..., F_n) = \frac{P(C)P(F_1, ..., F_n|C)}{P(F_1, ..., F_n)} \tag{4.7}$$

  In Equation (4.7) C is the dependent class which is conditional on several feature variables F_1 through F_n.

- **C4.5** is an algorithm that generates a decision tree [42]. The resulting decision tree is exploited for classification. The decision tree is created based on the training data which includes already classified samples. Each sample includes an n-dimensional vector containing some features representing the attributes of the sample as well as the class of the sample. At every node of the decision tree, the C4.5 algorithm selects the attribute of the samples that effectively divides the set of samples into separate classes. Then the algorithm recurses on the smaller sublists. The splitting metric is information gain [35] .

- **Logistic Regression** is a type of regression analysis [30] exploited for predicting the value of a dependent variable that can take a few different values. The probabilities impacting the possible values of the resulting variable are modeled as a function of predicting variables using a logistic function. Logistic regression substantially refers to the problems in which the dependent variable can take only two different values. For the problems with more than two classes, mutinomial logistic regression could be utilized [18]. The logistic regression algorithm measures the relationship between

a dependent variable and some independent variables making use of probabilistic scores as the predicted values of the dependent variable.

4.6.1.1 Evaluation Metrics

The evaluation of the retrieval performance is carried out by the following metrics: accuracy, kappa, and Area Under the Curve (AUC). Accuracy is the proportion of true results (truly classified "dup"s and "non"s) among all pairs being classified. The formula for accuracy is indicated bellow.

$$acc = \frac{|true\ dup| + |true\ non|}{|true\ dup| + |false\ dup| + |true\ non| + |false\ non|} \tag{4.8}$$

True "dup" and false "dup" are the pairs of bug reports truly and wrongly recognized as "dup" respectively by classifiers. True "non" and false "non" have the same definition but for the "non" class value.

Kappa is a statistical measure for inter-rater agreement. For example, kappa demonstrates how much homogeneity there is in the rating given by raters. The equation for kappa is:

$$kappa = \frac{Pr(a) - Pr(e)}{1 - Pr(e)} \tag{4.9}$$

$Pr(a)$ is the relative observed agreement among the raters. $Pr(e)$ is the hypothetical probability of chance agreement, using the observed data, to calculate the probabilities of each observer randomly saying each category. If the judges are in complete agreement, then $kappa = 1$. If there is no agreement among them other than what would be expected by chance , then $kappa = 0$.

AUC is the area under the Receiver Operation Characteristic (ROC) curve. ROC curve is created by plotting the fraction of truly recognized "dup"s out of all the recognized "dup"s (True Positive Rate) versus the fraction of wrongly recognized "dup"s out of all the recognized "non"s (False Positive Rate) by the classifiers. AUC is the probability that a classifier will rank a randomly chosen "dup" instance higher than a randomly chosen "non" one (assuming that "dup" class has a higher rank than "non" class).

4.6.2 Retrieving the List of the Most Similar Candidates

As addressed in Chapter 1, usually duplicate report detection techniques are applied to alleviate the heavy load of triaging either by filtering the duplicate bug reports automatically

or providing a list of top-K identical bug reports to an incoming report. To apply our technique to a real world issue-tracking system and assist the triagers, we will provide a list of top-K similar reports for any incoming bug report as well. For this purpose, we will need a comparison metric to retrieve the candidate duplicates based on.

Figure 4.3 demonstrates the method of retrieving top-K similar bug reports to a specific incoming report. As indicated in this figure, there is a similarity criterion which compares the incoming bug reports against the existing reports in the repository and returns a sorted list of candidate duplicate reports. To compare the bug reports contextually, we have proposed some similarity criteria. These criteria could be divided into three categories: (1) Cosine similarity based metric; (2) Euclidean distance based metric; (3) Logistic regression based metric.

As addressed in Chapter 3, Sun *et al.* [52] have applied a linear function called REP to sort the candidate masters; which indicated promising results in detecting correct masters for duplicate reports. Therefore, we decided to combine our contextual comparison metrics with REP so that we can take advantage of all the textual, categorical, and contextual features when comparing bug reports.

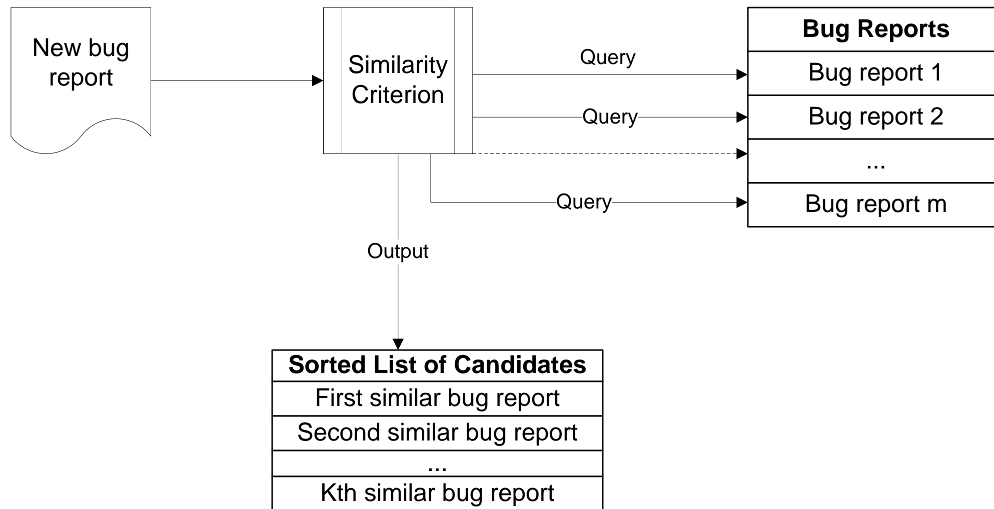

Figure 4.3: Overal workflow to retrieve duplicate bug reports

To implement this method, we have established a data-set including all the pairs of bug reports for each bug repository we are working on. This data-set includes all the pairs with the similarity criteria we would like to study in the experiment at hand (the "similarity_criteria" data-set). For instance, if the only similarity criterion is the REP function (Sun *et al.*'s method [52]), the data-set includes the IDs of the bug reports, the label for each pair

("dup" or "non") , and the REP result for each pair. Table 4.5 indicates some sample records from the "similarity_criteria" data-set with the REP criterion for Mozilla bug reports.

Table 4.5: Examples of pairs of bug reports from Mozilla bug repository with their REP comparisons result and their class (the "similarity_criteria" table)

Bug ID 1	Bug ID 2	REP	Class
563260	576854	2.617	dup
563250	596269	3.388	non
563325	612618	0.095	non
563308	582608	1.928	non
563276	602852	0.576	non

4.6.2.1 Cosine Similarity based Metric

The first similarity metric is the cosine similarity for a specific context which is presented in Equation (4.6). In this formula, the contextual weight vectors of the two bug reports at hand are compared. If two bug reports were exactly similar in terms of a particular context, their contextual cosine similarity is equal to 1. And, if the two bug reports were completely different in terms of that context, this value will be 0.

Since we expect this similarity criterion to assign a higher score to the more similar reports in comparison to the non-similar ones, we utilize it as a duplicate report retrieval criterion. This metric returns higher values when compares more similar vectors.

To combine the cosine similarity function and REP we have normalized REP to scale it in the range of 0-1. Then, we simply calculated the average of normalized REP and cosine similarity metric as indicated below:

$$combined_cosine_metric(B_1, B_2) = \frac{norm(REP(B_1, B_2)) + cosine_sim(C_1, C_2)}{2} \tag{4.10}$$

In this function C_1 and C_2 represent the contextual features of the bug reports B_1 and B_2 respectively. As an example, for the NFR context, C_1 and C_2 contain six dimensions each. Equation (4.10) combines all the textual, categorical, and contextual similarity metrics to compare a pair of bug reports. In some experiments only the $cosine_sim(C_1, C_2)$ formula is applied to compare the bug reports only contextually.

To apply this criterion, and its combination with the REP function, we have added some new comparison metrics to the "similarity_criteria" data-set. Table 4.6 indicates some sample records from the "similarity_criteria" data-set with the REP, and $cosine_sim$ criteria (for different contexts) from Mozilla bug reports. So, if we want to combine the REP and

$cosine_sim$ functions, we exploit two columns from this table, otherwise we only utilize one of them to compare the bug reports.

Table 4.6: Examples of pairs of bug report from Mozilla repository with their REP and cosine_sim comparisons for different contexts and their class

ID 1	ID 2	REP	Architecture cosine	NFR cosine	Random cosine	LDA cosine	Class
563260	576854	2.617	0.622	0.625	0.000	0.807	dup
563250	596269	3.388	0.955	0.877	0.000	0.392	non
563325	612618	0.095	0.000	0.854	0.474	0.076	non
563308	582608	1.928	0.256	0.916	0.000	0.077	non
563276	602852	0.576	0.000	0.802	0.000	0.000	non

4.6.2.2 Euclidean Distance based Metric

The second similarity metric is established based on the dimensional distance of the two context vectors pertaining to the bug reports being compared. The following formula depicts this metric.

$$contextual_distance(B_1, B_2) = \sum_{i=1}^{n} \frac{1}{1 + |C_{1i} - C_{2i}|} \tag{4.11}$$

In this function, n is the number of the word lists of the context at hand (such as 6 for NFR context). C_1 and C_2 are the contextual features for the bug reports B_1 and B_2 respectively. This similarity metric is analogous to the priority and version comparison metrics illustrated in Figure 4.2. In this function, as the distance between the two context vectors increases, the resulting value approaches to 0. And, when this distance decreases, the resulting value approaches to 1. Therefore, this function results in higher scores for the bug reports contextually close to each other.

To compare the bug reports textually, categorically, and contextually (utilizing the above contextual similarity metric) we have combined Equation (4.11) and the REP linear function. To that end, we simply added the $contextual_distance$ function to REP which could be considered as adding a few more features like priority and version to the REP function. The formula is shown bellow:

$$combined_euclidean_metric(B_1, B_2) = REP(B_1, B_2) + contextual_distance(B_1, B_2) \tag{4.12}$$

In the above formula, B_1 and B_2 are the two bug reports being compared. To apply this criterion, we have added some new comparison metrics to the "similarity_criteria" data-set (demonstrated in Table 4.5). Table 4.7 indicates some sample records from this data-set

with the REP, and $contextual_distance$ criteria (for different contexts) from Mozilla bug reports.

Table 4.7: Examples of pairs of bug reports from Mozilla repository with their REP and contextual_distance comparisons for different contexts and their class

ID 1	ID 2	REP	Arch distance	NFR distance	Random distance	LDA distance	Class
563260	576854	2.6170	2.2987	3.3549	0.9741	10.8501	dup
563250	596269	3.3880	1.6458	3.1567	1.1871	09.6204	non
563325	612618	0.0950	0.5789	1.9480	1.4071	10.3120	non
563308	582608	1.9280	1.3722	3.4686	1.1337	05.3772	non
563276	602852	0.5760	0.8739	2.5228	2.5519	02.5027	non

4.6.2.3 Logistic Regression Based Comparison

Among all the machine learning classification algorithms we have decided to exploit logistic regression to combine REP and contextual comparison metrics. The reason is that this algorithm provides us with reasonable performance in detecting duplicate bug reports in our experiments. Also, this algorithm provides us with appropriate coefficients for combining two different metrics applying a probabilistic model.

This similarity criterion works based on the logistic regression classifier. This classifier is applied in cases where the observed outcome (dependent variable) can accept only two possible values (in our case “dup” or “non”). Logistic regression is applied to predict the probability of being the desired case based on the values of the independent variables. The logistic regression takes the natural logarithm of the probability that a particular outcome is the desired case divided by the probability that it is not the case to create a continuous criterion as a converted form of the dependent variable. The natural logarithm of success is fit to the predictors using linear regression analysis [49]. The predicted value of this logarithm is converted back to the probability of success using exponential function. Hence, the logistic regression estimates the probability of being the case over the probability of not being the case as a continuous variable and a threshold is exploited to translate the predicted probabilities as a success or failure.

Logistic regression is adapted from the logistic function which always has a resulting value between 0 and 1. A logistic function is indicated below.

$$F(t) = \frac{e^t}{e^t + 1} = \frac{1}{1 + e^{-t}} \tag{4.13}$$

If t was a linear combination of descriptive (independent) variables such as x_1, x_2, ..., and x_m, the logistic function could be written as follows:

$$\pi(x) = \frac{e^{\beta_0+\beta_1 x_1+\beta_2 x_2+...+\beta_m x_m}}{e^{\beta_0+\beta_1 x_1+\beta_2 x_2+...+\beta_m x_m}+1} = \frac{1}{1+e^{-(\beta_0+\beta_1 x_1+\beta_2 x_2+...+\beta_m x_m)}} \quad (4.14)$$

This could be considered as the probability of the success. One of the characteristics of the logistic function is that it can accept any input in any range and provide an output in the range of 0 and 1. The probability of success over the probability of not being the case for the above logistic function is as follows:

$$g(x) = ln\frac{\pi(x)}{1+\pi(x)} = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + ... + \beta_m x_m \quad (4.15)$$

In the above formula, β_0 is called the intercept and $\beta_1 x_1$, $\beta_2 x_2$,..., and $\beta_m x_m$ are the regression coefficients multiplied by some values of predictors. The probability of the dependent variable equaling a success is equal to the values of the logistic function (shown in Equation (4.14)).

The regression coefficients of the logistic function are calculated applying maximum likelihood estimation [39]. Usually, an iterative process is utilized to estimate the coefficients. This process starts with trial values for the coefficients; and then revises the coefficients repeatedly to examine if they could be improved.

In logistic regression, the probability function is modeled as a linear function which includes a linear combination of the independent variables and a set of estimated coefficients. The predictor function for a particular data point i is written as follows:

$$f(i) = \beta_0 + \beta_1 xi1 + \beta_2 x_{2i} + ... + \beta_m x_{mi} \quad (4.16)$$

in which $\beta_0 - \beta_m$ are regression coefficients indicating the impact of each particular descriptive variable on the outcome.

In our approach, we have taken advantage of the prediction function (4.16) to establish a similarity criterion to sort the candidate duplicates based on. Considering the REP and the $cosine_sim$ (Equation (4.6)) functions as the descriptive variables, we are interested in distinguishing the appropriate coefficients for them in the similarity criterion. In other words, we would like to discover improved coefficients for the $cosine_sim$ and REP measures to combine them rather than simply calculating their average (such as what is done in Equation (4.10)). To that end, we have sampled the "similarity_criteria" data-set including the REP and $cosine_sim$ similarity functions (illustrated in Table 4.6). The sampled data-set includes 10000 records with 20% of records labeled as "dup" and 80% of records labeled as "non". Then, the logistic regression classifier is applied on this sampled data-set.

If we intend to exploit the REP function exclusively in our similarity criterion, we apply the logistic regression on the sampled data-set including only the REP similarity metric; and the resulting similarity criterion is as follows:

$$criterion = \beta_0 + \beta_1(REP) \tag{4.17}$$

In which the coefficients are estimated by the logistic regression classifier. Furthermore, when we aim to combine two different metrics such as REP and cosine_sim for NFR context to sort the candidate duplicates, we apply the logistic regression on the sampled data-set including the REP and the cosine_sim for NFR to predict the coefficients. The similarity criterion is defined as follows:

$$criterion = \beta_0 + \beta_1(REP) + \beta_2(cosine_sim_{NFR}) \tag{4.18}$$

Also, if we aim to investigate the effect of contextual comparison (for a specific context such as NFR) on the duplicate bug report retrieval, the similarity criterion changes as follows:

$$criterion = \beta_0 + \beta_1(cosine_sim_{NFR}) \tag{4.19}$$

In the above functions the coefficients are calculated by logistic regression. These criteria are in fact the prediction function demonstrated in Equation (4.16) that are exploited as the bug report comparison function.

4.6.2.4 Evaluating the List of Candidates

So far, we have presented our methods of duplicate bug report retrieval. In this section, we aim to describe the technique we exploit to assess this retrieval approach. As stated earlier in this Chapter, the retrieval functions are applied to compare every incoming bug report against all the existing reports in the repository and sort the existing bug reports based on their similarity to the incoming report. Since the list of candidates is sorted in descending order based on the similarity measure, the most similar reports are expected to be ranked higher in the resulting list. The quality of the sorted list of candidates is measured by studying the index of the actual duplicates (of the incoming report) in the list. If the actual duplicates are ranked relatively high in the list, we conclude that the similarity criterion at hand is performing successfully in identifying the duplicate bug reports.

To evaluate the sorted list of similar reports, we have utilized the Mean Average Precision (MAP) measure. MAP is a single-figure measure of ranked retrieval results independent from the size of the top list. It is designed for general ranked retrieval problem, where a query can have multiple relevant answers. In our duplicate bug report retrieval method, each bug report can have several duplicates since there may be several bug reports in a bucket. Given Q duplicate bug reports, for each of them, the system retrieves duplicates in descendant order of similarity (until all the duplicates of the bug report are retrieved) and records their indexes in the sorted list. The MAP measure is calculated as follows:

$$MAP = \frac{\sum_{q=1}^{Q} AvgP(q)}{Q} \tag{4.20}$$

$$AvgP(q) = \frac{\sum_{k=1}^{n} P(k) \times (Rel(K))}{number_of_relevent_documnets} \tag{4.21}$$

In the above functions, relevant_documents are the actual duplicates; $Rel(K) = 1$ when the documents are duplicates of each other and $Rel(K) = 0$ otherwise; n is the number of bug reports in the repository; and $p(k)$ is the precision at the cut-off k. The precision function is provided below:

$$precision = \frac{|\{relevant \quad docs\} \bigcap \{retrieved \quad docs\}|}{|retrieved \quad docs|} \tag{4.22}$$

Since in our experiments we would like to evaluate our list of candidates, we should know the actual duplicates of an incoming report. Therefore, in our experiments, the incoming reports consist of all the reports marked as duplicate in the repository (that have specific masters). This approach is similar to Sun *et al.*'s work [52]. However, Sun *et al.* have simplified MAP to an MRR-like measure addressed in Chapter 2. But, we believe that applying the original version is more meaningful in this case according to the fact that every incoming bug report can have multiple duplicates.

Here, we provide an example to illustrate how the MAP measure works. Assume that there are two duplicate reports with IDs 7 and 2 in a bug repository of 10 bug reports. We demonstrate the buckets for this repository in Figure 4.4.

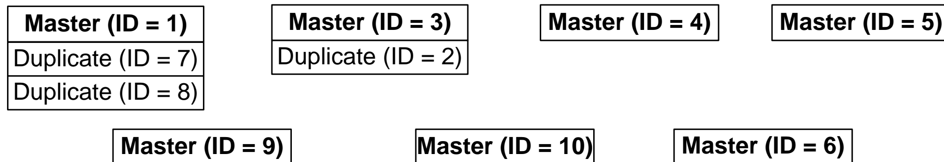

Figure 4.4: Buckets of the bug reports for a sample repository.

Assume a specific similarity criterion is being applied on this repository. And, returns the sorted lists of candidates for the duplicate bug reports with IDs 7 and 2 as demonstrated in Figures 4.5 and 4.6 respectively. In these figures, the highlighted bug reports are the duplicates of incoming reports at hand. In Figure 4.5, the indexes of the actual duplicates are 2 and 5. Thus, according to Equation (4.21), the resulting average precision for the duplicate bug report with the ID 7 is as follows:

$$AvgP(report(ID:7)) = \frac{\sum_{k=1}^{9} P(k) \times (Rel(K))}{2} = \frac{\frac{1}{2} + \frac{2}{5}}{2} = 0.45 \tag{4.23}$$

The bug report with the ID 2 has only one duplicate. The resulting average precision for the duplicate report with ID 2 is provided below:

$$AvgP(report(ID = 7)) = \frac{\sum_{k=1}^{9} P(k) \times (Rel(K))}{2} = \frac{1}{1} = 1 \tag{4.24}$$

Consequently, the MAP value for the similarity criterion at hand for this repository is 0.725 which is the mean of $AvgP$s. MAP returns values from 0 to 1. Higher values for the MAP imply the better performance of the duplicate report retrieval similarity criterion.

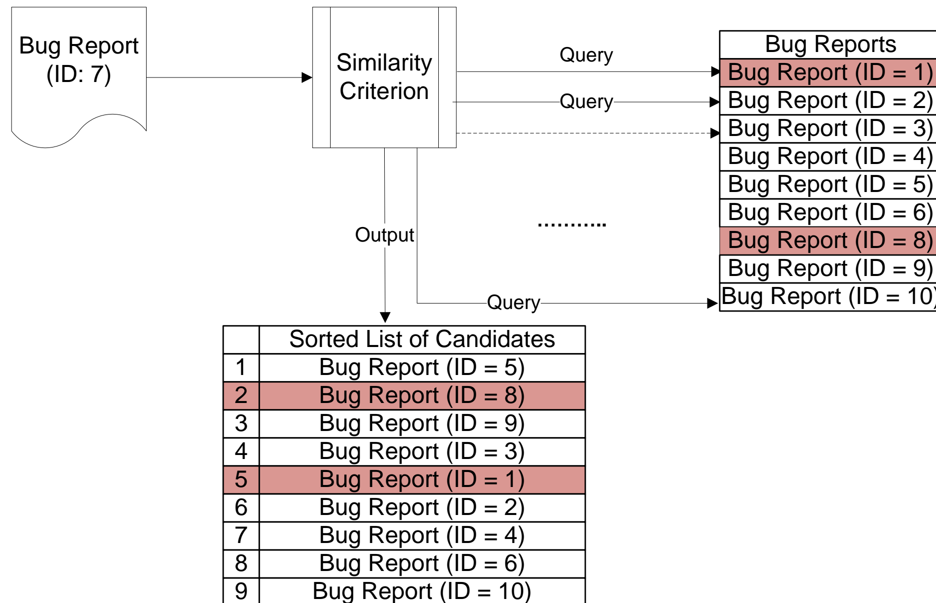

Figure 4.5: An example of bug report retrieval scenario for a duplicate bug report with ID 7 and evaluating the retrieval method using MAP measure.

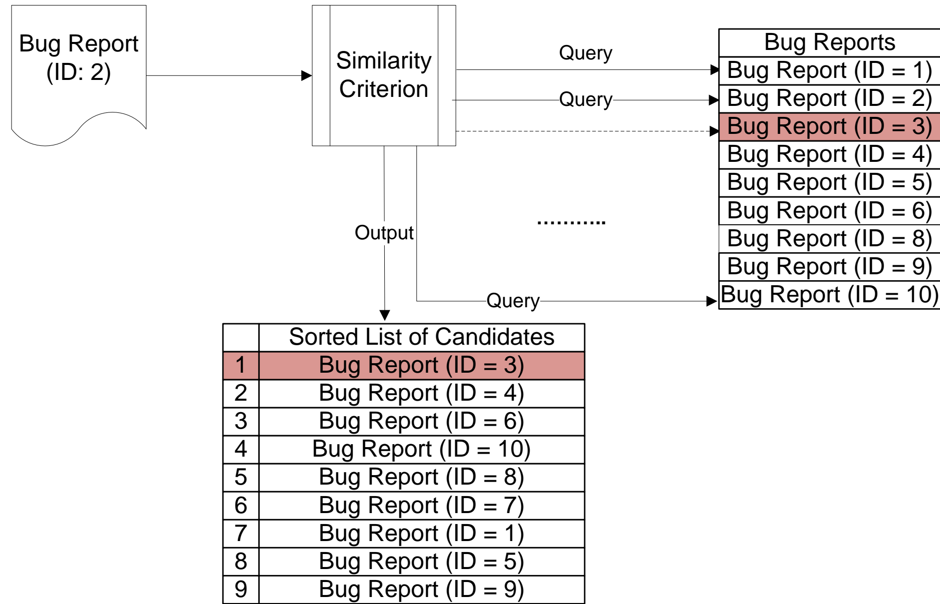

Figure 4.6: An example of bug report retrieval scenario for a duplicate bug report with ID 2 and evaluating the retrieval method using MAP measure.

4.6.2.5 The FastREP algorithm

In order to implement the logistic regression based retrieval function and evaluating this function by the MAP measure, we have proposed an algorithm called the "FastREP" algorithm. This algorithms involves the following steps:

1. As addressed in section 4.6.2.3, we sample the "similarity_criteria" data-set including the REP and $cosine_sim$ similarity criteria (illustrated in Table 4.6). The sampled data-set includes 10000 records with 20% of records labeled as "dup" and 80% of records labeled as "non". Then, we apply the logistic regression classifier on this sampled data-set. Depending on the experiment, the features involving in this classification may vary (the features may be only REP, only the contextual feature(s), or a combination of both).

2. Based upon the criterion coefficients returned by the logistic regression classifier, the criterion function is built and the value of this function for each record in the "similarity_criteria" data-set is calculated (applying a criterion function similar to either one of Equation (4.17), (4.18). or (4.19) depending on the experiment) and added to this data-set as a new column.

3. The resulting table that includes the criterion column is then sorted based on the criterion column's values in descending order.

4. Next, we establish a data structure keeping the index and the number of "dup" records in the sorted table for each one of the duplicate bug reports in the repository. This data structure provides all the required features for calculating the MAP function.

5. We walk through the sorted table and fill in the above discussed data structure. To that end, for every record marked as "dup" in the sorted table, we save the number of "dup"s observed for the duplicate bug report included in the current record and append the rank of the current "dup" to our list of dup ranks for that report.

6. Next, we calculate the average precision (indicated in Equation (4.21)) for every duplicate record in the repository.

7. Finally, we calculate the mean of all the $AvgP$s to report the MAP result.

This method is fast because we can answer many queries in parallel and have only 1 sort step. The runtime is $O(NlogN)$ as we use a merge sort (GNU sort) to sort the large tables.

Chapter 5

Case Studies

We applied our method on Android, Eclipse, Mozilla, and OpenOffice bug-tracking systems. To study the effect of contextual data on the accuracy of duplicate bug-report detection, we applied the classification algorithms (mentioned in Chapter 4) on Android bug repository in our recent work [1]. In this thesis, we applied the same approach on Eclipse, Mozilla, and OpenOffice bug repositories as well. We applied classification algorithms on three different data-sets extracted from each bug-tracker:

1. the data-set including all of the similarity measurements illustrated in the "all-features" tables (such as Table 4.4, the NFR all-features table);
2. the data-set including only the textual and categorical similarity measurements of the bug reports; and
3. the data-set including only the contextual similarity measurement features.

As mentioned before, these data-sets include 20000 pairs of randomly selected bug reports with 20% "dup"s and 80% "non"s.

we have also conducted some experiments to provide the list of candidate duplicates benefiting from the REP function presented by Sun *et al.* [52]. We have extended the REP function (in section 4.6.2) to apply our contextual approach when providing the list of candidates by three different methods: (1) cosine similarity based metric, (2) euclidean distance based metric, (3) logistic regression based metric. We discuss the experiments applying these retrieval methods in section 5.3.

5.1 Evaluating the Classification-based Retrieval Method

In this section we analyze the effect of context on detecting duplicate bug reports based on the results reported by the machine learning classifiers while applying them on the

“all_features” data-sets (described in Chapter 4) with and without contextual features. Here, we are eager to know the answer to the following question: Does the software context improve the duplicate bug reports retrieval using the machine learning classifiers?

Tables 5.1, 5.2, 5.3, and 5.4 show the statistical evaluation measurement values without considering the context of bug reports at the top. This part of the tables demonstrates the resulting evaluation measures of the machine learners with the input data created by Sun *et al.*'s method [52]. The maximum values are shown in bold in the preceding tables. These tables demonstrate that Sun *et al.*'s method definitely finds duplicates.

Tables 5.1, 5.2, 5.3, and 5.4 also report the statistical measurement results, using the contextual data-sets in bug-report similarity measurements. The highest value in each column is shown in bold.

As addressed in our previous work [1], classification algorithms can effectively identify the duplicate bug reports of Android bug repository. Table 5.1 reports the results of applying these algorithms on this repository. As shown in this table, the highest improvements are achieved by utilizing the LDA and Labeled-LDA contextual data.

Tables 5.2 and 5.3 report the results for Eclipse and Mozilla bug repositories respectively. As demonstrated in these tables, the LDA context makes the highest improvement again however this improvement is trivial. According to what is reported in tables 5.2 and 5.3, the contextual features exclusively improves the accuracy of detecting duplicate bug reports by around 6% over the baseline. This result is promising because LDA is an automatic method and not that expensive to run and if its topics can help boost deduplication performance, then we have an automatic method of improving duplicate detection. For OpenOffice bug repository, as indicated in Table 5.4, the highest improvement (which is still trivial) is achieved by using NFR context. The NFR contextual word lists are project independent so it can be considered as an automatic method of bug report deduplication as well.

Table 5.5 illustrates some examples of predictions made by K-NN machine learning algorithm for Android bug repository including textual, categorical, and Labeled-LDA context's data. The first pair of bug reports is correctly recognized as duplicates by the machine learner given that both of the reports are about “Bluetooth” (which is an Android Labeled-LDA topic). For the same reason the pair 4 is recognized as a duplicate by the machine learner while the reports in this pair are not duplicates of each other. In pair 2, the bug reports are categorically different and also textually not similar in terms of Android Labeled-LDA topics, but they are wrongly classified as non-duplicates by the machine learner. In

Table 5.1: Statistical measures resulted by the experiments on Android bug repository including textual, categorical, and contextual data

Context	Algorithm	Textual, Categorical, & Contextual			Contextual only		
		Accuracy %	Kappa	AUC	Accuracy %	Kappa	AUC
No Context	0-R	80.000%	0.0000	0.500			
	Logistic Regression	82.830%	0.3216	0.814			
	Naive Bayes	78.625%	-0.0081	**0.778**			
	C4.5	**84.525%**	0.4324	0.716			
	K-NN	82.380%	**0.4616**	0.737			
Architecture	0-R	80.000%	0.0000	0.500	80.000%	0.0000	0.500
	Logistic Regression	83.060%	0.3562	0.829	79.965%	0.0005	0.618
	Naive Bayes	77.950%	0.2185	0.732	75.255%	0.0825	0.603
	C4.5	87.990%	0.5947	0.880	91.690%	0.7083	0.916
	K-NN	85.580%	0.5632	0.794	86.330%	0.5553	0.843
NFR	0-R	80.000%	0.0000	0.500	80.000%	0.0000	0.500
	Logistic Regression	83.325%	0.3615	0.833	79.995%	0.0014	0.617
	Naive Bayes	78.735%	0.1106	0.758	77.880%	0.0509	0.619
	C4.5	89.450%	0.6661	0.856	96.145%	0.8792	0.952
	K-NN	85.295%	0.5766	0.813	83.165%	0.5222	0.788
Random English Words	0-R	80.000%	0.0000	0.500	80.000%	0.0000	0.500
	Logistic Regression	83.730%	0.3854	0.844	80.200%	0.0543	0.661
	Naive Bayes	51.845%	0.1341	0.665	39.260%	0.0515	0.606
	C4.5	89.995%	0.6673	0.901	91.590%	0.7101	0.917
	K-NN	87.955%	0.6384	0.834	87.620%	0.6119	0.863
LDA	0-R	80.000%	0.0000	0.500	80.000%	0.0000	0.500
	Logistic Regression	86.780%	0.5382	0.886	80.590%	0.1447	0.732
	Naive Bayes	77.290%	0.3179	0.767	73.565%	0.2523	0.712
	C4.5	91.245%	0.7284	0.866	**96.070%**	**0.8759**	**0.946**
	K-NN	88.615%	0.6854	0.887	89.345%	0.7034	0.894
Labeled LDA	0-R	80.000%	0.0000	0.500	80.000%	0.0000	0.500
	Logistic Regression	88.125%	0.5967	0.904	82.605%	0.3151	0.798
	Naive Bayes	79.655%	0.3508	0.788	77.560%	0.3082	0.747
	C4.5	**92.105%**	0.7553	0.888	95.430%	0.8574	0.939
	K-NN	91.500%	**0.7561**	**0.911**	92.405%	0.7801	0.921

the pair 3, the reports are categorically similar and they are correctly recognized as non-duplicates as they are about two different Android Labeled-LDA topics.

Figure 5.1 shows the ROC curves for results of applying K-NN algorithm on various "all-features" tables (such as Table 4.4) for Android bug repository. It also displays the ROC curve for the "textual_categorical" table (such as Table 4.2). The figure shows that the Labeled-LDA context outweighs the other ones. The "No context" curve shows the per-

Table 5.2: Statistical measures resulted by the experiments on Eclipse bug repository including textual, categorical, and contextual

Context	Algorithm	Textual, Categorical, & Contextual			Contextual only		
		Accuracy %	Kappa	AUC	Accuracy %	Kappa	AUC
No Context	0-R	80.000%	0.0000	0.500			
	Logistic Regression	96.610%	0.8922	**0.989**			
	Naive Bayes	96.500%	0.8896	0.985			
	C4.5	**96.650%**	**0.8947**	0.975			
	K-NN	95.270%	0.8522	0.915			
Architecture	0-R	80.000%	0.0000	0.500	80.000%	0.0000	0.500
	Logistic Regression	96.685%	0.8947	0.989	82.100%	0.2164	0.718
	Naive Bayes	96.125%	0.8786	0.983	77.660%	0.2157	0.648
	C4.5	96.700%	0.8961	0.966	83.720%	0.3462	0.700
	K-NN	94.395%	0.8240	0.917	80.910%	0.3852	0.714
NFR	0-R	80.000%	0.0000	0.500	80.000%	0.0000	0.500
	Logistic Regression	96.680%	0.8943	0.989	79.960%	0.0337	0.665
	Naive Bayes	96.350%	0.8848	0.980	79.960%	0.0269	0.643
	C4.5	96.585%	0.893	0.955	83.130%	0.3495	0.705
	K-NN	93.725%	0.8043	0.904	78.010%	0.3619	0.699
Random English Words	0-R	80.000%	0.0000	0.500	80.000%	0.0000	0.500
	Logistic Regression	96.605%	0.8921	0.989	80.720%	0.0983	0.661
	Naive Bayes	92.095%	0.7714	0.949	41.870%	0.0702	0.610
	C4.5	96.660%	0.8954	0.964	83.120%	0.3132	0.681
	K-NN	94.920%	0.8417	0.930	80.600%	0.3459	0.710
LDA	0-R	80.000%	0.0000	0.500	80.000%	0.0000	0.500
	Logistic Regression	**96.750%**	**0.8968**	**0.990**	**86.215%**	0.4716	0.854
	Naive Bayes	94.710%	0.8382	0.972	78.765%	0.3208	0.722
	C4.5	96.640%	0.8945	0.954	85.120%	0.5174	0.747
	K-NN	94.225%	0.8222	0.919	84.070%	**0.5376**	**0.792**

formance of K-NN algorithm using the data generated by Sun *et al.*'s measurements (only textual and categorical measurements) which show poor performance in comparison to the other curves. Thus, adding extra features with or without Sun *et al.*'s features improves bug-deduplication performance.

Figure 5.2 demonstrates the ROC curves for results of applying C4.5 algorithm on the "all-features" tables for Android bug repository. It also indicates the performance of C4.5 on the "textual_categorical" table. This diagram shows a tangible gap between the performance of C4.5 using different contextual data-sets and its performance without using any context.

Table 5.3: Statistical measures resulted by the experiments on Mozilla bug repository including textual, categorical, and contextual data

Context	Algorithm	Textual, Categorical, & Contextual			Contextual only		
		Accuracy %	Kappa	AUC	Accuracy %	Kappa	AUC
No Context	0-R	80.000%	0.0000	0.500			
	Logistic Regression	94.065%	0.8075	**0.971**			
	Naive Bayes	92.670%	0.7679	0.961			
	C4.5	**94.085%**	**0.8114**	0.943			
	K-NN	92.810%	0.7432	0.857			
Architecture	0-R	80.000%	0.0000	0.500	80.000%	0.0000	0.500
	Logistic Regression	94.340%	0.8169	0.973	80.450%	0.0980	0.719
	Naive Bayes	91.105%	0.7326	0.950	74.775%	0.1248	0.646
	C4.5	94.470%	0.8247	0.938	84.985%	0.4656	0.750
	K-NN	91.895%	0.7451	0.879	82.395%	0.4290	0.728
NFR	0-R	80.000%	0.0000	0.500	80.000%	0.0000	0.500
	Logistic Regression	94.210%	0.8133	0.973	79.825%	0.0136	0.637
	Naive Bayes	92.650%	0.7690	0.956	80.145%	0.0380	0.658
	C4.5	93.630%	0.7968	0.914	80.285%	0.1740	0.650
	K-NN	88.260%	0.6343	0.818	73.465%	0.2263	0.621
Random English Words	0-R	80.000%	0.0000	0.500	80.000%	0.0000	0.500
	Logistic Regression	94.390%	0.8188	0.973	82.110%	0.1994	0.684
	Naive Bayes	78.075%	0.4870	0.893	35.880%	0.0461	0.638
	C4.5	94.170%	0.8151	0.941	82.635%	0.2829	0.640
	K-NN	90.440%	0.7020	0.859	79.620%	0.3473	0.694
LDA	0-R	80.000%	0.0000	0.500	80.000%	0.0000	0.500
	Logistic Regression	**94.775%**	**0.8318**	**0.975**	84.280%	0.4049	**0.823**
	Naive Bayes	91.265%	0.7331	0.947	78.855%	0.3333	0.753
	C4.5	94.135%	0.8138	0.900	**85.980%**	**0.5257**	0.747
	K-NN	89.505%	0.6796	0.849	83.415%	0.5133	0.775

Figure 5.3 displays the ROC curves resulted by applying K-NN algorithm on "all-features" and "textual_categorical" tables for Eclipse bug repository. In this diagram, the LDA context shows the highest improvement in performance. Figure 5.4 depicts the same curves resulted by applying the Logistic Regression algorithm. This diagram indicates a very slight improvement when applying the LDA context.

Figure 5.5 displays the ROC curves resulted by applying C4.5 algorithm on "all-features" and "textual_categorical" tables for Mozilla bug repository. As indicated in this diagram, the LDA context shows the highest improvement. Figure 5.6 reveals the same curves resulted by applying the K-NN algorithm. As illustrated in this diagram, the highest improvement is achieved by the architecture context.

Table 5.4: Statistical measures resulted by the experiments on OpenOffice bug repository including textual, categorical, and contextual data

Context	Algorithm	Textual, Categorical, & Contextual			Contextual only		
		Accuracy %	Kappa	AUC	Accuracy %	Kappa	AUC
No Context	0-R	80.000%	0.0000	0.500			
	Logistic Regression	93.125%	0.7729	**0.961**			
	Naive Bayes	91.415%	0.6960	0.951			
	C4.5	**93.210%**	**0.7789**	0.932			
	K-NN	90.580%	0.7042	0.812			
Architecture	0-R	80.000%	0.0000	0.500	80.000%	0.0000	0.500
	Logistic Regression	93.210%	0.7761	0.961	80.000%	0.0000	0.600
	Naive Bayes	91.545%	0.7039	0.938	80.000%	0.0000	0.604
	C4.5	92.975%	0.7734	0.920	79.995%	0.0565	0.573
	K-NN	88.400%	0.6332	0.821	77.180%	0.1926	0.647
NFR	0-R	80.000%	0.0000	0.500	80.000%	0.0000	0.500
	Logistic Regression	**93.670%**	**0.7925**	**0.966**	80.010%	0.0011	0.624
	Naive Bayes	91.470%	0.6988	0.947	80.000%	0.0000	0.604
	C4.5	92.380%	0.7562	0.893	80.325%	0.1775	0.645
	K-NN	84.105%	0.5130	0.762	73.605%	0.2114	0.611
Random English Words	0-R	80.000%	0.0000	0.500	80.000%	0.0000	0.500
	Logistic Regression	93.275%	0.7803	0.961	80.550%	0.1045	0.631
	Naive Bayes	83.460%	0.5669	0.880	31.860%	0.0296	0.591
	C4.5	93.120%	0.7762	0.935	81.215%	0.1657	0.586
	K-NN	88.015%	0.6266	0.809	78.605%	0.2808	0.692
LDA	0-R	80.000%	0.0000	0.500	80.000%	0.0000	0.500
	Logistic Regression	93.385%	0.7825	0.964	**81.030%**	0.1435	**0.699**
	Naive Bayes	89.090%	0.6402	0.911	74.870%	0.1551	0.630
	C4.5	92.505%	0.7618	0.890	79.350%	0.2502	0.654
	K-NN	84.130%	0.5111	0.760	77.935%	**0.3507**	0.688

Table 5.5: Examples of predictions made by K-NN algorithm for Android bug repository including textual, categorical, and Labeled-LDA context's data

Pair	ID	Title	Comp	Prio	Type	Vers	Act	Pred
1	3063	Bluetooth does not work with Voice Dialer	Device	Med	Def		dup	dup
	8152	Need the ability to use voice dial over bluetooth		Med	Def			
2	3029	support for Indian Regional Languages...	Framework	Med	Enh		dup	non
	4153	Indic fonts render without correctly reordering..	GfxMedia	Med	Def			
3	8846	Bluetooth Phonebook Access Profile ...		Med	Def	2.2	non	non
	31989	[ICS] Question of Google Maps' location...		Med	Def			
4	719	enhanced low-level Bluetooth support	Device	Med	Enh		non	dup
	1416	Bluetooth DUN/PAN Tethering support	Device	Med	Enh			

Figure 5.7 displays the ROC curves resulted by applying C4.5 algorithm on "all-features" and "textual_categorical" tables for OpenOffice bug repository. As this diagram shows, the

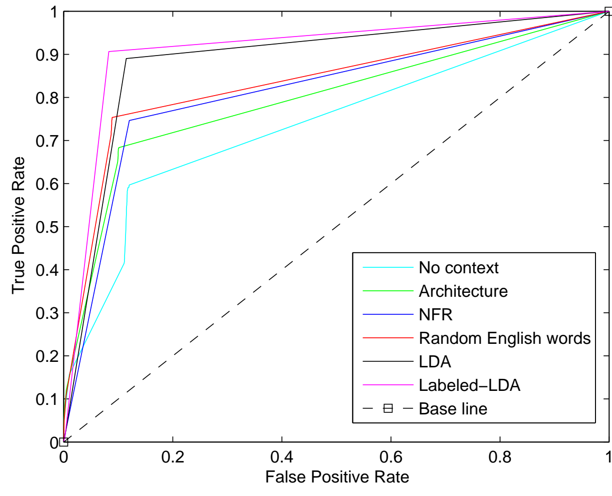

Figure 5.1: ROC curves resulted by applying K-NN algorithm on Android reports.

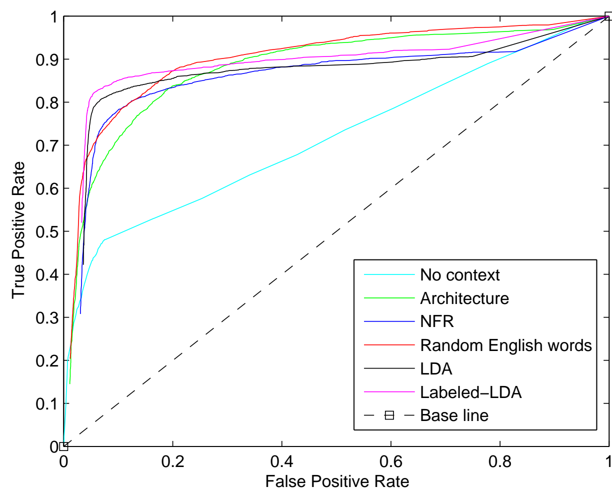

Figure 5.2: ROC curves resulted by applying C4.5 algorithm on Android reports.

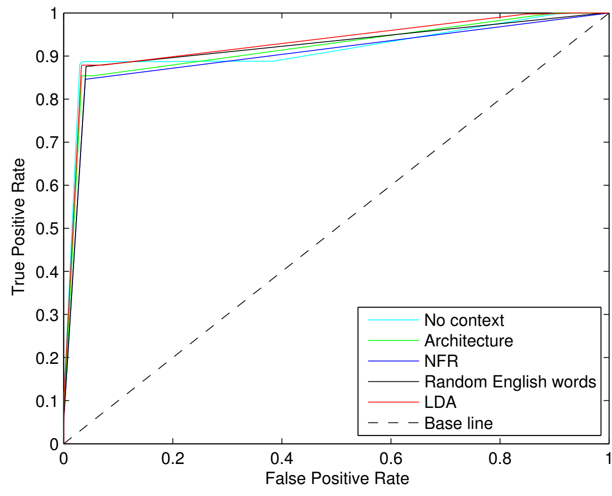

Figure 5.3: ROC curves resulted by applying K-NN algorithm on Eclipse reports.

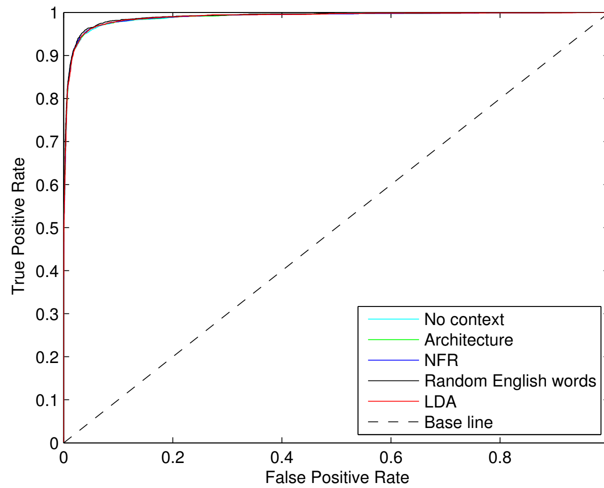

Figure 5.4: ROC curves resulted by applying logistic regression algorithm on Eclipse reports.

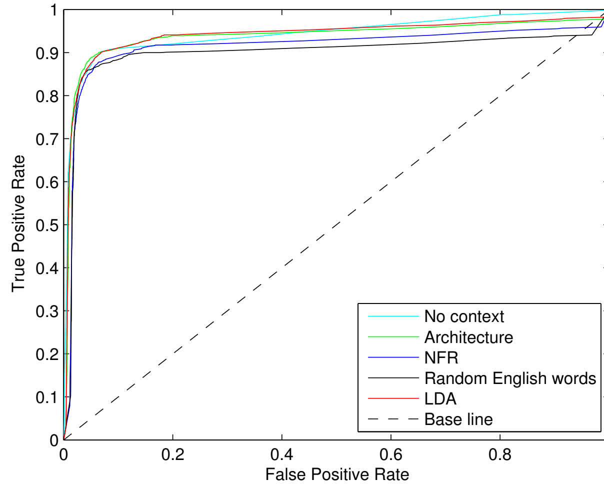

Figure 5.5: ROC curves resulted by applying C4.5 algorithm on Mozilla reports.

highest performance is achieved by applying the LDA context. Figure 5.8 reveals the same curves resulted by applying the Logistic Regression algorithm. This diagram indicates a slight improvement when applying the NFR context.

5.1.1 Discussion of Findings

Taking into account the reported measurements, the contextual features in some cases cause remarkable improvement while in some other cases only make a slight improvement in the duplicate report identification. Hence, the contextual information with no doubt can recognize the duplicate bug reports. But, our experiments demonstrate that there is a notable difference between the performance of identifying duplicate reports in Android and other bug repositories. We believe this difference is because of an important difference among the

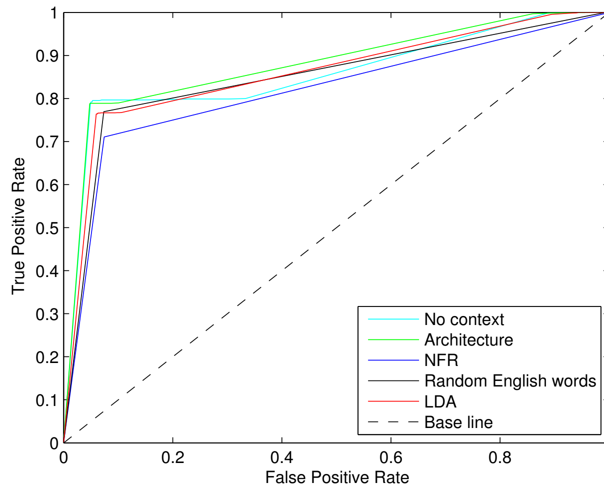

Figure 5.6: ROC curves resulted by applying K-NN algorithm on Mozilla reports.

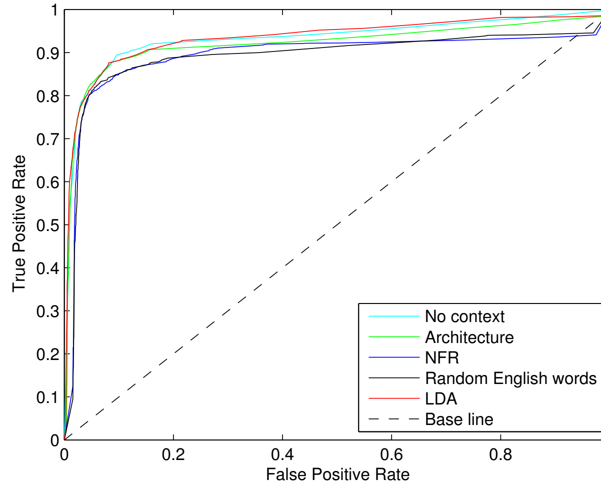

Figure 5.7: ROC curves resulted by applying C4.5 algorithm on OpenOffice reports.

structure of these bug repositories. As indicated in Figure 3.1, there is a considerably large bucket of duplicate reports in Android bug repository (including 188 duplicate reports). We believe that contextual features of the duplicate bug reports in the same bucket are recognized by the machine learning classifiers as similar patterns which help in recognizing duplicate bug reports efficiently. Also, according to the fact that larger buckets provide more pairs marked as "dup", they provide more training data for the classifiers. Consequently, duplicate bug reports pertaining to the large buckets are easily recognized by the machine learners when they are provided by contextual features.

To examine this idea, we have removed all the bug reports belonging to the largest bucket from Android bug repository and created a repository called Android_modified. Then, we conducted an experiment which traces the predictions made by the machine learn-

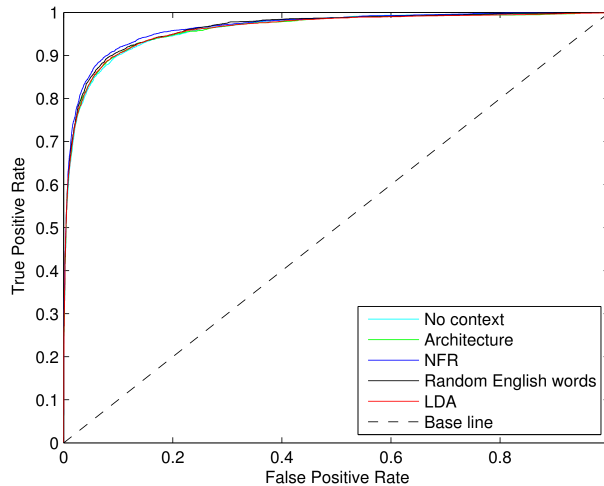

Figure 5.8: ROC curves resulted by applying logistic regression algorithm on OpenOffice reports.

ers for the data-sets including only the LDA contextual features. In this experiment, the machine learner with the highest prediction performance is taken into consideration for each bug repository. Then, the False Negative (FN) predictions made by the machine learners. I.e. we investigated the "dup" instances not recognized by the machine learner. Also, we divided the buckets of bug reports into two groups: the large buckets (buckets including 10 or more duplicate bug reports) and the small buckets (the buckets including less than 10 duplicate reports). As a result, we realized that 90% of the FNs for the android_modified are from the small buckets while only 37% of all the pairs marked as "dup" are from these buckets. For Eclipse repository, 88% of the FNs belong to the small buckets while 75% of the "dup"s are from these buckets. This experiment on Mozilla repository indicated that 81% of the FNs belong to small buckets while only 53% of the "dup"s are related to these buckets. Finally, for OpenOffice bug repository, 82% of the FNs belong to the small buckets while 68% of the "dup" instances are from these buckets. These result support the idea that machine learners can identify the duplicate reports belonging to the larger buckets more effectively in comparison to the duplicate reports from small buckets when applying contextual features.

As a result, we realized that classifiers identify the duplicate records belonging to large buckets more effectively in comparison to the duplicates from small buckets (like buckets of size 2 that are very common as demonstrated in Figure 3.1). These observations leaded us to the conclusion that duplicate records belonging to large buckets of duplicates are more easy to identify by the machine learners in comparison to the rest of the duplicate reports.

5.2 Effectiveness of Number of Features

As mentioned previously, each contextual data-set adds some new contextual features to each bug report. The number of these contextual features is equal to the number of word lists included in the contextual data-set. In this section, we analyze the influence of the number of added features (to the bug reports) on the bug-deduplication process. Here we answer the follwoing research question: Does adding more features (even junk) to the bug reports improve the accuracy of duplicate bug report detection regardless of their context?

Figures 5.9, 5.11, 5.13, and 5.15 show the relationship between the kappa measure and the number of added features to Android, Eclipse, Mozilla, and OpenOffice bug reports respectively. Each box-plot in these figures represents the distribution of kappa values for each context reported by the machine learning classifiers (0-R, Naive Bayes, Logistic Regression, K-NN, and C4.5). In Figure 5.9, there is a little difference between the performance of Random English Word context and NFR context, but NFR adds 20 fewer features. Consequently, context is more important than feature count. Figures 5.11, 5.13, 5.15, imply that although the English Random Words includes the maximum number of features, it resulted in the weakest performance among the other contexts. This result reveals that number of added features is not effective in improving the detection of duplicate bug reports. And, it is the context that impacts the prediction of duplicates.

Moreover, we display the correlation between the number of added features and AUC in Figures 5.10, 5.12, 5.14, and 5.16. The AUC measure for Naive Bayes, Logistic Regression, K-NN, and C4.5 is demonstrated in these figures. The Figure 5.10 shows the relation between the number of added features and AUC by fitting a linear regression function (the slope of this line is 0.0012). The measured correlation value for this figure is 0.46 which does not represent a high positive correlation. For Figures 5.12, 5.14, and 5.16 the slopes are -0.0002, -0.0006, and -0.0008 respectively which imply the low amount of correlation between the number of added features and the efficiency of detecting duplicate reports.

5.2.1 Discussion of Findings

Taking into account the points mentioned above, it is evident that adding more features can not improve the performance of duplicate bug reports detection per se.

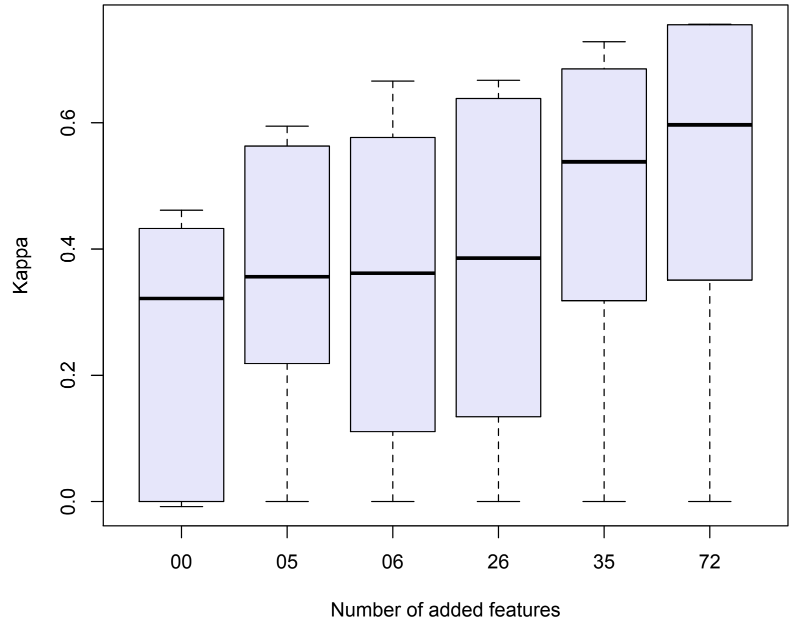

Figure 5.9: Kappa versus number of added features for Android bug repository. The x axis shows the number of features each context adds to the bug reports (which is equal to the number of word lists of the contextual data). The contexts from left to right are no context, architecture, NFR, Random words, LDA, and Labeled-LDA.

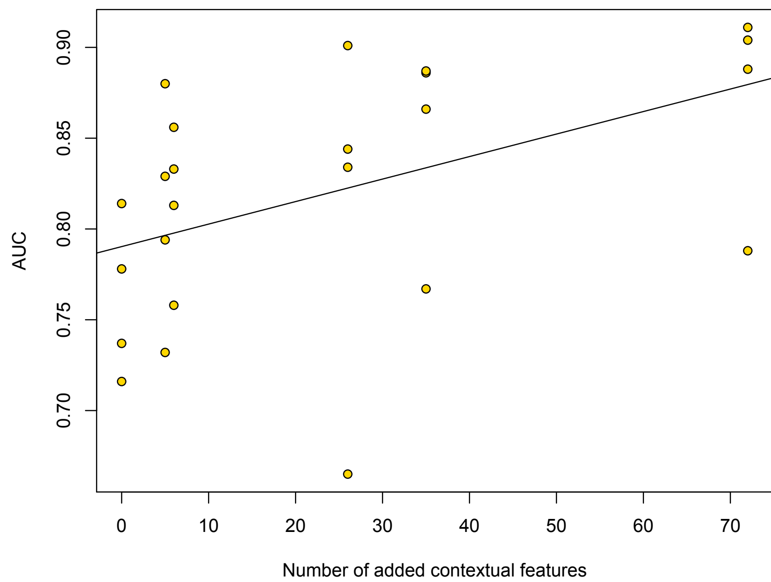

Figure 5.10: AUC versus number of added features for Android bug repository. The x axis shows the number of features each context adds to the bug reports. The contexts from left to right are no context, architecture, NFR, Random words, LDA, and Labeled-LDA.

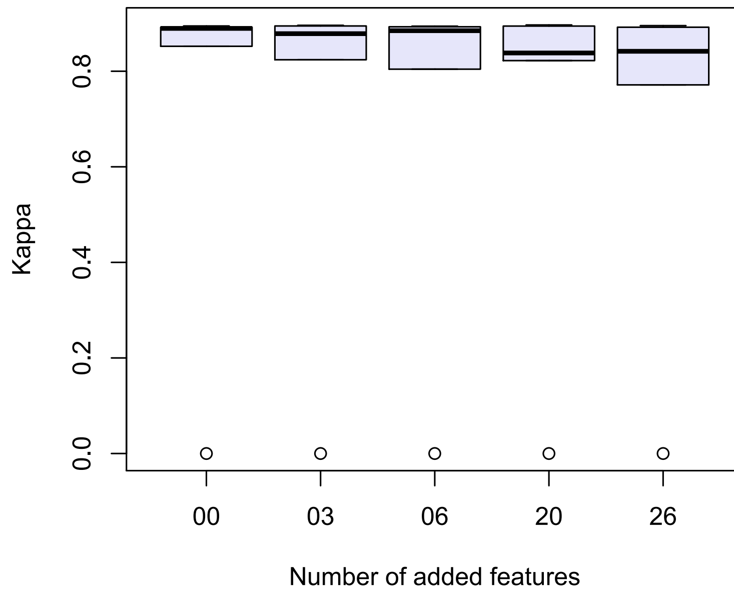

Figure 5.11: Kappa versus number of added features for Eclipse bug repository. The x axis shows the number of features each context adds to the bug reports (which is equal to the number of word lists of the contextual data). The contexts from left to right are no context, architecture, NFR, Random words, and LDA.

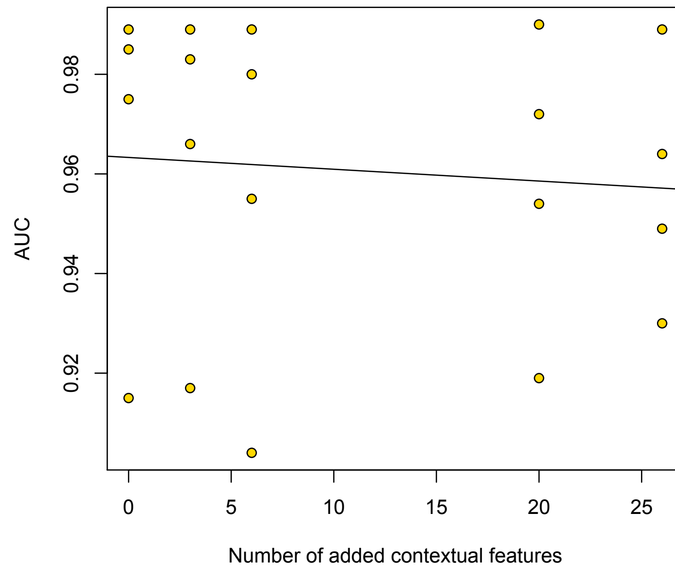

Figure 5.12: AUC versus number of added features for Eclipse bug repository. The x axis shows the number of the features each context adds to the bug reports. The contexts from left to right are no context, architecture, NFR, Random words, and LDA.

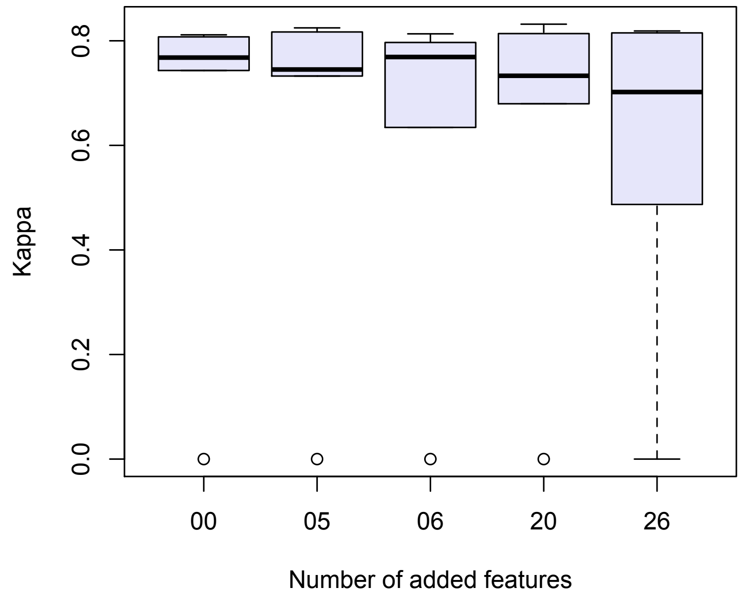

Figure 5.13: Kappa versus number of added features for Mozilla bug repository. The x axis shows the number of features each context adds to the bug reports (which is equal to the number of word lists of the contextual data). The contexts from left to right are no context, architecture, NFR, Random words, and LDA.

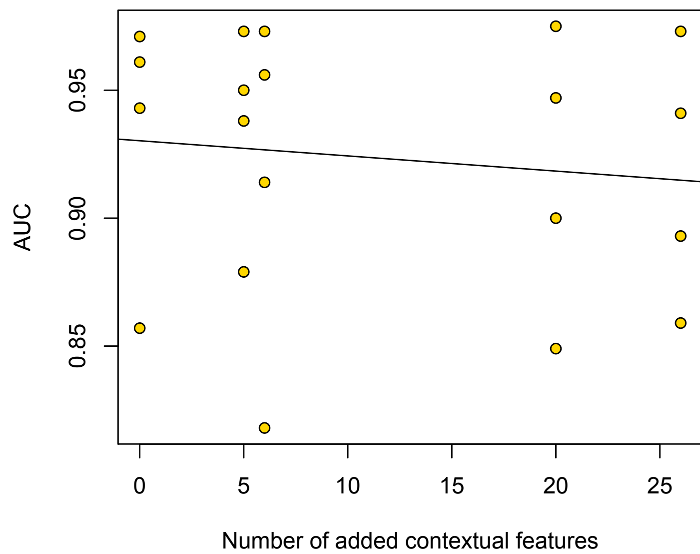

Figure 5.14: AUC versus number of added features for Mozilla bug repository. The x axis shows the number of the features each context adds to the bug reports. The contexts from left to right are no context, architecture, NFR, Random words, and LDA.

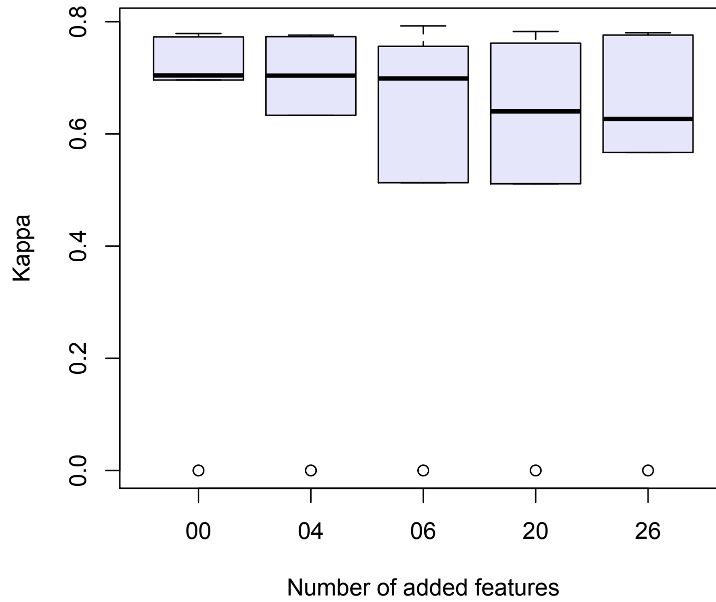

Figure 5.15: Kappa versus number of added features for OpenOffice bug repository. The x axis shows the number of features each context adds to the bug reports (which is equal to the number of word lists of the contextual data). The contexts from left to right are no context, architecture, NFR, Random words, and LDA.

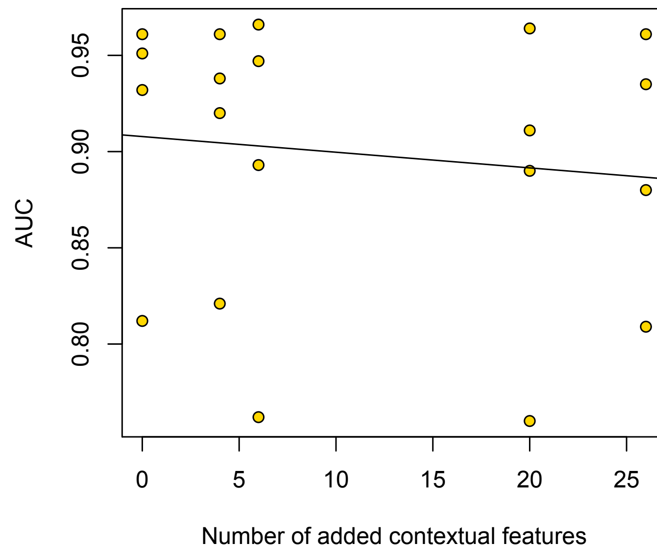

Figure 5.16: AUC versus number of added features for OpenOffice bug repository. The x axis shows the number of the features each context adds to the bug reports. The contexts from left to right are no context, architecture, NFR, Random words, and LDA.

5.3 Evaluating the List of Candidates

In this section, we discuss the impact of context on filtering the bug reports and providing a list of candidate duplicate reports to triagers, based on the reported results of the MAP measure. This bug report retrieval approach is different from the method applied in our recent work [1] in which only the machine learning classifiers are exploited to decide whether two bug reports are duplicates or not. Here, we aim to address the following research question: Could the software context improve the quality of the list of candidate duplicates to an incoming bug report?

Table 5.6: MAP results for the list of candidates of Android bug repository

Criterion	MAP
REP	0.410
Architecture context	0.020
NFR context	0.200
LDA context	0.293
LabeledLDA	0.298
REP and Architecture context (cosine similarity)	0.376
REP and NFR context (cosine similarity)	0.301
REP and LDA context (cosine similarity)	0.324
REP and LabeledLDA context (cosine similarity)	0.330
REP and Architecture context (Euclidean distance similarity)	0.383
REP and NFR context (Euclidean distance similarity)	0.414
REP and LDA context (Euclidean distance similarity)	0.318
REP and LabeledLDA context (Euclidean distance similarity)	0.412
REP (Logistic regression based)	0.505
all (Logistic regression based)	0.459
all without REP (Logistic regression based)	0.097
REP and architecture cosine (Logistic regression based)	0.499
REP and LDA cosine (Logistic regression based)	**0.513**
REP and Random English words cosine (Logistic regression based)	0.479
REP and NFR cosine (Logistic regression based)	0.468
REP and Labeled-LDA cosine (Logistic regression based)	0.501
Architecture context (Logistic regression based)	0.042
LDA context (Logistic regression based)	0.365
Random English words context (Logistic regression based)	0.101
NFR context (Logistic regression based)	0.003
Labeled-LDA context (Logistic regression based)	0.374

Tables 5.6, 5.7, 5.8, and 5.9 report the MAP results for Android, Eclipse, Mozilla, and OpenOffice bug reports respectively. In each one of these tables, three different types of similarity criterion functions are assessed exploiting the MAP measure. As addressed

Table 5.7: MAP results for the list of candidates of Eclipse bug repository

Criterion	MAP
REP	0.379
Architecture context	0.008
NFR context	0.069
LDA context	0.072
REP and Architecture context (cosine similarity)	0.306
REP and NFR context (cosine similarity)	0.280
REP and LDA context (cosine similarity)	0.209
REP and Architecture context (Euclidean distance similarity)	0.367
REP and NFR context (Euclidean distance similarity)	0.372
REP and LDA context (Euclidean distance similarity)	0.245
REP (Logistic regression based)	**0.457**
all (Logistic regression based)	0.453
all without REP (Logistic regression based)	0.135
REP and architecture cosine (Logistic regression based)	0.455
REP and LDA cosine (Logistic regression based)	0.455
REP and Random English words cosine (Logistic regression based)	0.455
REP and NFR cosine (Logistic regression based)	0.456
Architecture context (Logistic regression based)	0.021
LDA context (Logistic regression based)	0.095
Random English words context (Logistic regression based)	0.037
NFR context (Logistic regression based)	0.077

earlier in section 4.6.2, these criteria are as follows: cosine similarity based, Euclidean distance based, and logistic regression based metrics.

Some of the conducted experiments exclusively make use of specific contextual information to retrieve the duplicate reports. Some other ones exploit both the REP function and the contextual information. Moreover, in some of the experiments, all the contextual information (except the Random English words one) are applied without utilizing the REP function (all without REP) and with exploiting the REP function (all). The highest MAP value achieved for each repository is presented in bold in the tables.

As indicated in Tables 5.6, 5.7, 5.8, and 5.9, applying the logistic regression based technique could make considerable enhancement in identifying duplicate reports. For instance, when the similarity criterion exclusively exploits the REP function, the logistic regression based approach provides helpful coefficients that boost the MAP value 9.5%, 7.8%, 9.1%, and 8.3% for Android, Eclipse, Mozilla, and OpenOffice bug repositories respectively in comparison to the case of normal REP being applied as the similarity criterion.

Table 5.8: MAP results for the list of candidates of Mozilla bug repository

Criterion	MAP
REP	0.208
Architecture context	0.005
NFR context	0.008
LDA context	0.018
REP and Architecture context (cosine similarity)	0.208
REP and NFR context (cosine similaxrity)	0.207
REP and LDA context (cosine similarity)	0.208
REP and Architecture context (Euclidean distance similarity)	0.169
REP and NFR context (Euclidean distance similarity)	0.203
REP and LDA context (Euclidean distance similarity)	0.089
REP (Logistic regression based)	**0.299**
all (Logistic regression based)	0.294
all without REP (Logistic regression based)	0.042
REP and architecture cosine (Logistic regression based)	0.299
REP and LDA cosine (Logistic regression based)	0.296
REP and Random English words cosine (Logistic regression based)	0.299
REP and NFR cosine (Logistic regression based)	0.299
Architecture context (Logistic regression based)	0.013
LDA context (Logistic regression based)	0.029
Random English words context (Logistic regression based)	0.015
NFR context (Logistic regression based)	0.013

5.3.1 Discussion of Findings

According to the experiments mentioned above, the added contextual data did not improve the duplicate report retrieval performance significantly. As reported in the tables, two of the repositories (Android and OpenOffice) showed that the combination of REP and contextual similarity measure is able to improve the performance of duplicate bug report detection by up to 0.7%. However, two other repositories (Mozilla and Eclipse) did not show any improvement after applying the software context. Consequently, we could not elevate the quality of the list of candidate duplicates greatly by considering the contextual data.

5.4 Context Matters

This document describes one scenario where context matters. And, we also showed how software-development context matters in prior work [24]. This study provides more evidence that we can gain in performance by including contextual features into our software engineering related IR tasks, whether it is bug deduplication or LDA topic labelling and tagging. We hope this work serves as a call to arms for researchers to start building corpora of

Table 5.9: MAP results for the list of candidates of OpenOffice bug repository

Criterion	MAP
REP	0.238
Architecture context	0.003
NFR context	0.041
LDA context	0.038
REP and Architecture context (cosine similarity)	0.180
REP and NFR context (cosine similarity)	0.136
REP and LDA context (cosine similarity)	0.105
REP and Architecture context (Euclidean distance similarity)	0.234
REP and NFR context (Euclidean distance similarity)	0.236
REP and LDA context (Euclidean distance similarity)	0.237
REP (Logistic regression based)	0.321
all (Logistic regression based)	0.322
all without REP (Logistic regression based)	0.078
REP and architecture cosine (Logistic regression based)	0.321
REP and LDA cosine (Logistic regression based)	0.321
REP and Random English words cosine (Logistic regression based)	0.318
REP and NFR cosine (Logistic regression based)	**0.322**
Architecture context (Logistic regression based)	0.010
LDA context (Logistic regression based)	0.051
Random English words context (Logistic regression based)	0.021
NFR context (Logistic regression based)	0.052

software concepts in order to improve automated and semi-automated software engineering tasks.

5.5 Threats to Validity

Construct validity is threatened by our word-lists in the sense of how they are constructed and if the word-lists actually represent context or just important tokens. Our measurements rely on the status of bug reports in some real-world bug-tracking systems that have a huge number of bug reports not processed by the triagers (have the status value of "New" or "Unconfirmed"). And, there may be many duplicate bug reports among them. Also, for Android, Eclipse, Mozilla, and OpenOffice bug repositories we exploited in this study, there are only 2%, 6%, 8%, and 9% of the bug reports marked as "duplicate". There are likely many unlabeled duplicate bug reports.

We address internal validity by replicating past work (Sun *et al.*) but also by evaluating both on true negatives (non-duplicates) and true positives (duplicates), where as Sun *et al.*'s methodology only tested for recommendations on true positives. Furthermore in-

ternal validity is bolstered by searching for rival explanations of increased performance by investigating the effect of extra features.

External validity is threatened by the fact that some particular characteristics of a bug repository might lead to our experiment results. To reduce this risk, we have used four large bug repositories related to different software projects in our experiments.

Chapter 6

Conclusions and Future Work

In this thesis, we proposed a method that helps to automate the bug report triage process by improving the bug report deduplication. We developed a novel bug report similarity measurement method to detect duplicate bug reports. Also, we suggest a new approach for duplicate bug report retrieval, based on a specific set of bug report similarity criteria.

The existing research on bug report deduplication has mainly focused on the IR techniques and textual comparison of the bug reports to find the duplicates. In this study, we have taken advantage of software contexts in addition to the textual and categorical comparison metrics to address the ambiguity of synonymous software-related words within bug reports written by users, who have different vocabularies. We assume that bug reports are likely to refer to a non-functional requirement or some functionalities related to some architectural components in the system. So, we use contextual data-sets including words organized in a few word lists including non-functional requirement words, software architectural words, software topic words extracted by LDA/Labeled LDA, and random English words (as a control).

We replicated Sun *et al.*'s [53] method of textual and categorical comparison and extended it by adding our contextual similarity measurements. To that end, we have added contextual features to the bug reports exploiting the above mentioned contextual word lists. These features are taken into consideration in addition to the basic features (description, title, type, component, version, priority, and product) while comparing the bug reports and measuring their similarities.

To retrieve the duplicate bug reports, we have created a data-set such that any record in the data-set includes a pair of bug reports with their textual, categorical, and contextual similarity measurements. Each record in this data-set also includes a feature showing whether the bug reports are duplicates of each other. This data-set is then provided to the

machine learning classifiers to decide if the bug reports in each record are duplicates, given the textual, categorical, and/or the contextual comparison features of the reports (applying 10-fold cross validation technique).

Moreover, we have proposed another duplicate report retrieval method which compares every incoming bug report against all the existing ones in the bug-tracking system. This comparison is carried out by applying the REP function (proposed by Sun *et. al.* [52]) or a combination of REP and one of the bug report similarity measurement criteria we have proposed such as the cosine distance based, Euclidean distance based, and logistic regression based metrics. Based on the values returned by the similarity measurement criterion, the existing bug reports in the repository are sorted in a way that the bug reports at the top of the list are expected to be the most similar bug reports to the incoming report. The effectiveness of this approach is evaluated by applying the Mean Average Precision (MAP) measure.

We have conducted our experiments on four large real-world bug repositories consisting of the bug reports from Android, Eclipse, Mozilla, and OpenOffice software projects. We were able to improve the accuracy of the duplicate bug report retrieval approach by applying the machine learning classifiers. This improvement was significant for Android bug repository (up to 11.5% over the Sun *et. al.*'s method [52]) and notable for other repositories at hand. We have also investigated the impact of the number of added features to the bug reports on the deduplication process by exploiting the English random words context which revealed a poor performance in comparison to other contexts. These results leaded us to the conclusion that the software context matters when comparing the bug reports.

We applied our method to help the triagers find the duplicate bug reports easily by the means of providing a list of candidate duplicates to any incoming bug report. To that end, we proposed three different similarity criteria for pairwise comparison of the bug reports including cosine similarity, euclidean distance based, and logistic regression based metrics. As a result, we found logistic regression based similarity metric as the most efficient one. However, by adding the contextual data to the similarity criterion we could only improve the list of candidates slightly for Android bug repository but not the rest of the repositories.

As our experiments illustrate, adding software contextual features to the bug reports can improve the performance of bug report deduplication while retrieving the duplicates by the machine learning classifiers. By adding the context, the classifiers can decide more efficiently if the two bug reports are duplicates or not. On the other hand, adding the contextual features could not enhance the quality of the list of candidate duplicates for the majority of software projects.

6.1 Contributions

The main contributions of this thesis are as follows:

1. We proposed the use of software engineering context for the problem of duplicate bug reports detection by adding the contextual features to the bug reports (section 4.4). For this purpose, the contextual word lists including software architectural word, software non-functional requirement words, software topic words extracted by LDA/ Labeled-LDA are exploited (section 3.2).

2. We presented a new duplicate bug report retrieval method that considers not only the true-positive duplicate cases but also the true-negative ones. To that end, "all_features" data-sets including the pairwise comparisons of the bug reports are created (section 4.5) and provided to the machine learning classifiers to predict the duplicates (section 4.6.1).

3. By applying the machine learning classifiers on the "all_features" tables (section 4.6.1), we were able to improve the accuracy of duplicate bug reports detection by 11.5%, 41%, and 16.8% in accuracy, Kappa, and AUC measures respectively (section 5.1) over the Sun et al.'s approach which only utilizes the textual and categorical features of the bug reports.

4. Finally, we posited some new bug report similarity criteria exploiting both the REP function (proposed by Sun et al. [52]) and the logistic regression classifier's probabilistic model (section 4.6.2.3) that are used to provide the list of the most similar candidate duplicates to the triagers. As discussed in section 4.6.2.4, this approach was able to improve the quality of the list of candidates by 9.5%, 7.8%, 9.1%, and 8.3% for Android, Eclipse, Mozilla, and OpenOffice bug repositories respectively over the Sun et al.'s [52] approach.

6.2 Future Work

In the future, we would like to extend our approach to take advantage of more software contexts, such as the execution traces in addition to the textual, categorical, and existing contextual data to find the duplicate bug reports more accurately. Furthermore, we would like to implement our method as an embedded tool in an issue-tracker to empirically investigate the role that this method can actually play in assisting the triagers and save their time

and effort when looking for the duplicates of an incoming bug report. This way, we can take advantage of their helpful feedback to enhance the effectiveness and usability of our approach.

Bibliography

[1] A. Alipour, A. Hindle, and E. Stroulia. A contextual approach towards more accurate duplicate bug report detection. In *Proceedings of the Tenth International Workshop on Mining Software Repositories*, pages 183–192. IEEE Press, 2013.

[2] J. Anvik, L. Hiew, and G. C. Murphy. Coping with an open bug repository. In *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*, pages 35–39. ACM, 2005.

[3] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *Proceedings of the 28th international conference on Software engineering*, pages 361–370. ACM, 2006.

[4] L. Aversano, L. Cerulo, and C. Del Grosso. Learning from bug-introducing changes to prevent fault prone code. In *Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting*, pages 19–26. ACM, 2007.

[5] N. Ayewah and W. Pugh. The google findbugs fixit. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 241–252. ACM, 2010.

[6] T. Bayes. A demonstration of the Second Rule in the Essay towards the Solution of a problem in the Doctrine of Chances, published in the Philosophical Transactions, Vol. LIII. Communicated by the Rev. Mr. Price, in a letter to Mr. John Canton, MAFRS. *Philosophical Transactions*, 54:296–325, 1765.

[7] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann. What makes a good bug report? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 308–318. ACM, 2008.

[8] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim. Duplicate bug reports considered harmful really? In *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pages 337–345. IEEE, 2008.

[9] D. Binkley and D. Lawrie. Information retrieval applications in software maintenance and evolution. *Encyclopedia of Software Engineering*, 2009.

[10] D. M. Blei and J. Lafferty. Topic models. *Text mining: classification, clustering, and applications*, 10:71, 2009.

[11] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *the Journal of machine Learning research*, 3:993–1022, 2003.

[12] Android Community. Android Technical Information. `http://source.android.com/devices/tech/security/index.html`, 2013.

[13] Android Community. Life of a Bug. `http://source.android.com/source/life-of-a-bug.html`, 2013.

[14] D. Čubranić. Automatic bug triage using text categorization. In *In SEKE 2004: Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering*. Citeseer, 2004.

[15] S. T. Dumais, G. W. Furnas, T. K. Landauer, S. Deerwester, S. C. Deerwester, et al. Latent semantic indexing. In *Proceedings of the Text Retrieval Conference*, 1995.

[16] T. Dunning. *Statistical identification of language*. Citeseer, 1994.

[17] B. Ganter, R. Wille, and R. Wille. *Formal concept analysis*. Springer Berlin, 1999.

[18] W. H. Greene. Econometric Analysis. *Engelwood Cliffs*, 1993.

[19] V. Guana, F. Rocha, A. Hindle, and E. Stroulia. Do the stars align? Multidimensional analysis of Android's layered architecture. In *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*, pages 124–127. IEEE, 2012.

[20] D. Han, C. Zhang, X. Fan, A. Hindle, K. Wong, and E. Stroulia. Understanding Android Fragmentation with Topic Analysis of Vendor-Specific Bugs. In *Reverse Engineering (WCRE), 2012 19th Working Conference on*, pages 83–92. IEEE, 2012.

[21] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th international conference on Software engineering*, pages 291–301. ACM, 2002.

[22] L. Hiew. *Assisted detection of duplicate bug reports*. PhD thesis, The University Of British Columbia, 2006.

[23] A. Hindle, N. Ernst, M. W. Godfrey, R. C. Holt, and J. Mylopoulos. Whats in a name? on the automated topic naming of software maintenance activities. *submissio n: http://softwareprocess. es/whats-in-a-name*, 125:150–155, 2010.

[24] 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*, pages 163–172. ACM, 2011.

[25] G. Holmes, A. Donkin, and I. H. Witten. Weka: A machine learning workbench. In *Intelligent Information Systems, 1994. Proceedings of the 1994 Second Australian and New Zealand Conference on*, pages 357–361. IEEE, 1994.

[26] N. Jalbert and W. Weimer. Automated duplicate detection for bug tracking systems. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pages 52–61. IEEE, 2008.

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

[28] A. Kuhn, S. Ducasse, and T. Gírba. Semantic clustering: Identifying topics in source code. *Information and Software Technology*, 49(3):230–243, 2007.

[29] J. Langford, L. Li, and A. Strehl. Vowpal wabbit online learning project, 2007.

[30] D. V. Lindley. Regression and correlation analysis. *New Palgrave: A Dictionary of Economics*, 4:120–23, 1987.

[31] R. Lotufo, Z. Malik, and K. Czarnecki. Modelling the hurriedbug report reading process to summarize bug reports. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 430–439. IEEE, 2012.

[32] J. I. Maletic and A. Marcus. Supporting program comprehension using semantic and structural information. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 103–112. IEEE Computer Society, 2001.

[33] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic. An information retrieval approach to concept location in source code. In *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*, pages 214–223. IEEE, 2004.

[34] G. Maskeri, S. Sarkar, and K. Heafield. Mining business topics in source code using latent dirichlet allocation. In *Proceedings of the 1st India software engineering conference*, pages 113–120. ACM, 2008.

[35] T. M. Mitchell. Machine learning. 1997. *Burr Ridge, IL: McGraw Hill*, 45, 1997.

[36] N. K. Nagwani and P. Singh. Weight similarity measurement model based, object oriented approach for bug databases mining to detect similar and duplicate bugs. In *Proceedings of the International Conference on Advances in Computing, Communication and Control*, pages 202–207. ACM, 2009.

[37] T. Nakashima, M. Oyama, H. Hisada, and N. Ishii. Analysis of software bug causes and its prevention. *Information and Software Technology*, 41(15):1059–1068, 1999.

[38] 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*, pages 70–79. ACM, 2012.

[39] J. Pfanzagl. *Parametric statistical theory*. Walter de Gruyter, 1994.

[40] D. Poshyvanyk and A. Marcus. Combining formal concept analysis with information retrieval for concept location in source code. In *Program Comprehension, 2007. ICPC'07. 15th IEEE International Conference on*, pages 37–48. IEEE, 2007.

[41] D. Poshyvanyk, A. Marcus, R. Ferenc, and T. Gyimóthy. Using information retrieval based coupling measures for impact analysis. *Empirical Software Engineering*, 14(1):5–32, 2009.

[42] J. R. Quinlan. *C4.5: programs for machine learning*, volume 1. Morgan kaufmann, 1993.

[43] D. Ramage, D. Hall, R. Nallapati, and C. D. Manning. Labeled LDA: A supervised topic model for credit attribution in multi-labeled corpora. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing: Volume 1-Volume 1*, pages 248–256. Association for Computational Linguistics, 2009.

[44] 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*, pages 42–49. ACM, 2004.

[45] S. E. Robertson and S. Walker. Some simple effective approximations to the 2-poisson model for probabilistic weighted retrieval. In *Proceedings of the 17th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 232–241. Springer-Verlag New York, Inc., 1994.

[46] P. Runeson, M. Alexandersson, and O. Nyholm. Detection of duplicate defect reports using natural language processing. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 499–510. IEEE, 2007.

[47] G. Salton and M. J. McGill. Introduction to modern information retrieval. 1986.

[48] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Communications of the ACM*, 18(11):613–620, 1975.

[49] G. AF. Seber and A. J. Lee. *Linear regression analysis*, volume 936. John Wiley & Sons, 2012.

[50] N. Serrano and I. Ciordia. Bugzilla, ITracker, and other bug trackers. *Software, IEEE*, 22(2):11–13, 2005.

[51] G. Shakhnarovich, T. Darrell, and P. Indyk. *Nearest-neighbor methods in learning and vision: theory and practice*, volume 3. MIT press Cambridge, MA, USA:, 2005.

[52] 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*, pages 253–262. IEEE Computer Society, 2011.

[53] 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*, pages 45–54. ACM, 2010.

[54] A. Sureka and P. Jalote. Detecting duplicate bug report using character n-gram-based features. In *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*, pages 366–374. IEEE, 2010.

[55] J. Sutherland. Business objects in corporate information systems. *ACM Computing Surveys (CSUR)*, 27(2):274–276, 1995.

[56] G. Tassey. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology*, pages 02–3, 2002.

[57] The Bugzilla Team. The Bugzilla Guide. `http://www.bugzilla.org/docs/3.0/html/lifecycle.html`, 2013.

[58] E. M. Voorhees et al. The TREC-8 Question Answering Track Report. In *TREC*, volume 99, pages 77–82, 1999.

[59] 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*, pages 461–470. ACM, 2008.

[60] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang. SNIAFL: Towards a static noninteractive approach to feature location. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 15(2):195–226, 2006.