RECOMENDAÇÕES DE BUGS SIMILARES PARA MANTENEDORES DE SOFTWARE

HENRIQUE SANTOS CAMARGOS ROCHA

RECOMENDAÇÕES DE BUGS SIMILARES PARA MANTENEDORES DE SOFTWARE

Tese apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Doutor em Ciência da Computação.

Orientador: Marco Túlio de Oliveira Valente
Co-Orientador: Humberto Torres Marques-Neto

Belo Horizonte
Agosto de 2016

HENRIQUE SANTOS CAMARGOS ROCHA

RECOMMENDING SIMILAR BUGS TO SOFTWARE MAINTAINERS

Thesis presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Doctor in Computer Science.

Advisor: Marco Túlio de Oliveira Valente
Co-Advisor: Humberto Torres Marques-Neto

Belo Horizonte
August 2016

© 2016, Henrique Santos Camargos Rocha.
Todos os direitos reservados

Ficha catalográfica elaborada pela Biblioteca do ICEx - UFMG

Rocha, Henrique Santos Camargos.

R672r Recommending similar bugs to software maintainers. / Henrique Santos Camargos Rocha. – Belo Horizonte, 2016.
xxiv, 122 f.: il.; 29 cm.

Tese (doutorado) - Universidade Federal de Minas Gerais – Departamento de Ciência da Computação.

Orientador: Marco Túlio de Oliveira Valente.
Coorientador: Humberto Torres Marques Neto.

1. Computação - Teses. 2. Software – Manutenção 3. Sistemas de recomendação. 4. Mineração de dados (Computação). I. Orientador. II. Coorientador. III. Título.

CDU 519.6*32 (043)

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Recommending similar bugs to software maintainers

HENRIQUE SANTOS CAMARGOS ROCHA

Tese defendida e aprovada pela banca examinadora constituída pelos Senhores:

PROF. MARCO TÚLIO DE OLIVEIRA VALENTE - Orientador
Departamento de Ciência da Computação - UFMG

PROF. HUMBERTO TORRES MARQUES NETO - Coorientador
Instituto de Ciências Exatas e Informática - PUC/MG

PROF. EDUARDO MAGNO LAGES FIGUEIREDO
Departamento de Ciência da Computação - UFMG

PROFA. GAIL C. MURPHY
Departamento de Ciência da Computação - UBC

PROF. LEONARDO GRESTA PAULINO MURTA
Instituto de Computação - UFF

PROF. MARCELO DE ALMEIDA MAIA
Departamento de Ciência da Computação - UFU

Belo Horizonte, 30 de agosto de 2016.

Acknowledgments

I would like to acknowledge and thank all those who helped me along the long road that is a doctoral degree.

First and foremost, I thank God for everything.

A special thanks for my parents, siblings, and family members who gave me the emotional support through all those years. Another special thanks to my girlfriend for her support and understanding. I also would like to thank my friends, specially the ones who noticed my absence in many social events while I was busy with my research.

I would like to express my sincere gratitude to my advisor Prof. Marco Túlio de Oliveira Valente and my co-advidor Prof. Humberto Torres Marques-Neto for the continuous support of my Ph.D study and related research, for their patience, motivation, and immense knowledge. Their guidance helped me in all the time of research and writing of this thesis. I could not have imagined having better advisors and mentors for my Ph.D study.

My sincere thanks also goes to Dr. Gail C. Murphy, who provided me an opportunity to join her team as intern, and who gave access to the laboratory. Without her precious support it would not be possible to conduct this research.

I also thank all the research colleagues at the LabSoft in UFMG and SPL in UBC. I consider many them good friends, who also share the search for knowledge and research. Only in the academia we can find so many different and interesting people with so many common goals.

I would like to thank the Mozilla foundation and its contributors who provided me with their bug tracking data which was invaluable to my research.

Finally, I thank the remaining members of my thesis committee: Prof. Dr. Eduardo Magno Lages Figueiredo, Prof. Dr. Marcelo de Almeida Maia, and Prof. Dr. Leonardo Gresta Paulino Murta, for participating in my defense and their insightful comments and revision which improved my research from various perspectives.

"I am wiser than this man, for neither of us appears to know anything great and good; but he fancies he knows something, although he knows nothing; whereas I, as I do not know anything, so I do not fancy I do. In this trifling particular, then, I appear to be wiser than he, because I do not fancy I know what I do not know."

(Socrates – Plato's Apology, 21-d)

Resumo

Normalmente, a manutenção de um software gira em torno do tratamento de relatórios que descrevem erros, ou bugs. Isto representa um grande esforço do time envolvido nesta tarefa, especialmente em projetos de código aberto. Desenvolvedores destes projetos selecionam os bugs que desejam resolver em repositórios específicos. Geralmente, o esforço geral não é coordenado e pode implicar em trabalho redundante. Além disso, destaca-se que desenvolvedores enfrentam trocas de contexto o que pode impactar negativamente na sua respectiva produtividade. Assim, entende-se que se os desenvolvedores trabalharem continuamente em bugs similares, essas trocas de contexto podem ser mitigadas. Nesta tese é proposto e avaliado um sistema de recomendação de bugs que auxilie um desenvolvedor a tratar bugs similares, minimizando assim a troca de contexto. Primeiramente, realizou-se uma caracterização do fluxo de tratamento de bugs seguido pelos desenvolvedores Mozilla, onde foram identificadas oportunidades de incluir recomendações de bugs similares, como por exemplo, que desenvolvedores menos experientes gastam mais tempo procurando por um bug do que realizando a sua correção. Em seguida, o sistema de recomendação proposto na tese, denominado NextBug, foi avaliado usando bugs resolvidos no passado do ecossistema Mozilla. Os resultados foram comparados a uma técnica de vanguarda para detecção de bugs duplicados e mostrou-se que o NextBug desempenha tão bem quanto esta técnica de referência. Finalmente, se reporta um estudo de campo realizado para monitorar os bugs corrigidos para sistemas Mozilla durante uma semana e se analisou e-mails enviados aos desenvolvedores que corrigiram estes bugs perguntado se eles trabalhariam nas recomendações fornecidas pelo NextBug. Do total de 66 desenvolvedores, 39 (59%) afirmaram que eles poderiam trabalhar nas recomendações. Além disso, 44 desenvolvedores (67%) expressaram interesse em usar o NextBug como um plug-in instalado no repositório de bugs.

Palavras-chave: Manutenção de software, tratamento de bugs, relatório de erro, sistema de recomendação, mineração de texto.

Abstract

The maintenance work to be performed on software systems, whether feature developments or defects, is typically described as a bug or issue report. Bug handling represents a major effort in most software projects, specially for popular open source systems. In such systems, developers self-select bugs from the many open bugs in a repository when they wish to perform work on the system. However, developers' work is not coordinated which can lead to redundant work when handling bugs. Moreover, developers face context changes during their work which has a negative impact on their productivity. Therefore, if developers are guided to work on similarly related bug reports, the context changes can be mitigated, improving their productivity. In this thesis, we propose and evaluate a recommender that suggests similar bugs that a developer can fix after handling a giving bug. The recommender is inspired by techniques originally designed to detect duplicated bug reports. To the best of our knowledge, there is currently no other research aimed to recommend similar bugs to help developers on addressing more bugs. First, we characterize the bug handling workflow followed by Mozilla developers, when we identify an opportunity to include recommendations of similar bugs in this workflow. For instance, we discovered that less skilled developers require more time to find a bug than actually fixing it. Second, we evaluate the proposed recommender, called NextBug, using past bugs resolved in the Mozilla ecosystem and we compare the results with a state-of-the-art technique for detecting duplicated bugs. The results show that NextBug performs just as well as this more complex technique. Finally, we report a field study where we monitored the bugs fixed for Mozilla during a week. We sent mails to the developers who fixed these bugs, asking whether they would consider working on the recommendations provided by NextBug; 39 developers (59%) stated that they would consider working on these recommendations; 44 developers (67%) also expressed interest in seeing NextBug plugin installed in their bug tracking systems.

Palavras-chave: Software maintenance, bug handling, bug report, recommendation system, text mining.

List of Figures

1.1 Amazon screenshot adapted to show the books' recomendation. 5

2.1 Modification Requests classification. Source: ISO/IEC 14764(2006) 10
2.2 Modeling in the Information Retrieval Process. 15
2.3 Cosine Similarity. 17

3.1 Standard Bugzilla Workflow (version 5.x) [Bugzilla Team, 2015]. 29
3.2 Workflow used by Mozilla's customized version of Bugzilla. The cells with a check mark represent valid transitions and white cells indicate there is no transition between these states. Dark gray cells are used when the beginning and end state are the same (i.e., there are no loops in the workflow). Resolved and Verified (colored red) indicate states that a bug is considered closed. This image was provided by a BMO Maintainer in 2015-12-29. . . . 30
3.3 Bug Life Cycle Workflows: (a) Mozilla's customized version, (b) Older Bugzilla version (3.x) 30
3.4 Fragment of a BFG for Firefox 32
3.5 Violin plots showing the distribution of resolution times (in days) per bug status. 34
3.6 Distribution of resolution times (in days) for each developer skill category. 36
3.7 BFG for all Mozilla bugs. 37
3.8 BFG for Fixed Mozilla bugs. 40
3.9 BFGs for fixed bugs on Mozilla systems: (a) *Core*, (b) *Firefox*. 42
3.10 BFGs according to Developers' Skill Profiles: (a) *Newbie*, (b) *Junior*, (c) *Senior*, and (d) *Expert*. 44

4.1 Proposed approach for recommending similar bugs 48
4.2 Screenshot showing the original Bugzilla interface for browsing a bug enhanced with the NextBug plug-in. Similar bugs are shown in the lower right corner. 51

4.3 NextBug dialog with the configuration options for filtering recommendations. 53
4.4 NextBug architecture showing the main components and their interactions. 54

5.1 Fixed issues per month 58
5.2 Files changed per mapped issue 59
5.3 Maximal overlap scenarios (boxes represent files changed to conclude a bug): (a) the context of the first bug is reused by the recommended bug; (b) no new files are included in the context of the recommended bug. 64
5.4 Feedback (Fb) 68
5.5 Precision (P) 69
5.6 Likelihood (L) 70
5.7 Recall (Rc) and Maximum Recall ($Max\ Rc$) 71
5.8 F-score (F_1) 71
5.9 NextBug Feedback (Fb), Precision (P), Likelihood (L), Recall (Rc), and F-score (F_1) without the component filter 72
5.10 F-score (F_1): (a) $\Omega = 0.25$, (b) $\Omega = 0.5$ 74
5.11 F-score (F_1): (a) $\Omega = 0.75$, (b) $\Omega = 0.5$ 74
5.12 Precision considering assigned developers 76
5.13 Likelihood considering assigned developers 76
5.14 Fixed issues per month 78
5.15 Files changed/browsed per mapped issue 79
5.16 Feedback (Fb) results: (a) Commits, (b) Context 80
5.17 Precision (P) results: (a) Commits, (b) Context 81
5.18 Likelihood (L) results: (a) Commits, (b) Context 81
5.19 Recall (Rc) results: (a) Commits, (b) Context 82
5.20 F-score (F_1) results: (a) Commits, (b) Context 82

6.1 E-mail sent to a Mozilla developer 86

List of Tables

3.1 Mozilla bugs (2012) classified by their resolution status 33
3.2 Types of Mozilla Users Registered in Bugzilla 35
3.3 Developers Classification by Skill 36
3.4 Top-10 Mozilla products considering fixed bugs registered in BMO 41

5.1 Issues per Severity 58
5.2 Files changed per mapped issue 59
5.3 Top-10 Mozilla systems by issues with changed files 60
5.4 Features of $REP(q, b)$ 62
5.5 Parameters of REP 63
5.6 Execution Time (ms) 77

6.1 Field study results 87
6.2 Examples of useful recommendations, according to Mozilla developers (including the textual similarity between the query and the provided recommendations) 88
6.3 Examples of incorrect recommendations, according to Mozilla developers (including the textual similarity between the query and the provided recommendations) 88
6.4 Reasons for not following a recommendation 89
6.5 Usefulness of a Bugzilla extension 90

A.1 Survey Answers 107

Contents

Acknowledgments **ix**

Resumo **xiii**

Abstract **xv**

List of Figures **xvii**

List of Tables **xix**

1 Introduction **1**

1.1 Problem and Motivation 1

1.2 Objective and Contributions 3

1.3 Publications 5

1.4 Outline of the Thesis 6

2 Background and Related Work **9**

2.1 Software Maintenance 9

2.1.1 Benefits of Periodic Maintenance 11

2.1.2 Context Changes 12

2.1.3 Issue Tracking Systems 13

2.2 Information Retrieval 14

2.2.1 Pre-processing Index Terms 15

2.2.2 Vector Space Model 16

2.3 Bug Characterization and Visualization 17

2.4 Detecting Duplicated Bug Reports 18

2.5 Assigning Bugs to Developers 21

2.6 Analyzing Similar Issue Reports 24

2.7 Recommendation Systems in Software Engineering 24

2.8 Final Remarks . . . 25

3 Bug Characterization Study **27**
3.1 Mozilla Maintenance Process . . . 27
3.1.1 The Life Cycle of Mozilla Bugs . . . 28
3.1.2 Understanding the Workflow . . . 31
3.2 Bug Flow Graphs . . . 31
3.3 Dataset Overview . . . 33
3.3.1 Mozilla Bugs . . . 33
3.3.2 Mozilla Users . . . 35
3.3.3 Developers Profile . . . 35
3.4 Study Results . . . 37
3.4.1 Overall Workflow Analysis . . . 37
3.4.2 Fixed Bugs Workflow . . . 39
3.4.3 Workflow of Top Systems in Number of Fixed Bugs . . . 41
3.4.4 Developers Workflow . . . 43
3.5 Threats to Validity . . . 46
3.6 Final Remarks . . . 46

4 Recommendations of Similar Bugs **47**
4.1 Proposed Approach . . . 47
4.1.1 Rationale . . . 49
4.2 NextBug: A Prototype Implementation . . . 50
4.2.1 Main Features . . . 50
4.2.2 Architecture and Algorithms . . . 53
4.3 Final Remarks . . . 55

5 Retrospective Study **57**
5.1 Comparative Study . . . 57
5.1.1 Data Collection . . . 57
5.1.2 Technique for Comparison: REP . . . 60
5.1.3 Study Design . . . 63
5.1.4 Evaluation Metrics . . . 65
5.1.5 Comparison Results . . . 68
5.1.6 Configuration Without the Component Filter . . . 71
5.1.7 Oracle Sensibility Testing . . . 73
5.1.8 Alternative Evaluation: Assigned Developers . . . 75
5.1.9 Execution Time . . . 77

5.1.10 Summary of Findings . . . 78
5.2 Second Study: Mylyn . . . 78
5.2.1 Mylyn Dataset . . . 78
5.2.2 Study Design . . . 79
5.2.3 Results . . . 79
5.3 Threats to Validity . . . 83
5.4 Final Remarks . . . 83

6 Field Study **85**
6.1 Study Design . . . 85
6.2 Results . . . 87
6.2.1 Question #1: Would you consider working on one of these bugs next? . . . 87
6.2.2 Question #2: Would you consider useful a Bugzilla extension with recommendations? . . . 89
6.3 Threats to Validity . . . 91
6.4 Final Remarks . . . 91

7 Conclusion **93**
7.1 Discussion . . . 93
7.1.1 Can we avoid context changes on issue handling? . . . 93
7.1.2 Does our approach increase developer's productivity? . . . 94
7.1.3 Applicability . . . 95
7.1.4 Can we improve the measured evaluation results? . . . 96
7.2 Contributions . . . 97
7.3 Future Work . . . 98

Bibliography **99**

Appendix A Field Study Answers **107**

Chapter 1

Introduction

In this chapter, we start by stating our problem and motivation (Section 1.1). Next, we discuss our objectives, goals, and intended contributions (Section 1.2). Then, we present our current publications (Section 1.3). Finally, we present the outline of this thesis (Section 1.4).

1.1 Problem and Motivation

The importance of software maintenance in software engineering is well-established, as well as the costs and complexity inherent to this activity [Tan and Mookerjee, 2005; Mookerjee, 2005; Aziz et al., 2009; Junio et al., 2011]. In most software organizations, systems are maintained periodically, i.e., maintenance requests[1] are grouped and implemented as part of software projects [Tan and Mookerjee, 2005; Marques-Neto et al., 2013]. Studies show the advantages of periodic maintenance such as dilution of costs, decrease of system degradation, and better use of software engineering practices [Banker and Slaughter, 1997; Tan and Mookerjee, 2005]. However, open-source projects typically adopt continuous maintenance policies, where the maintenance requests are addressed by developers with different skills and commitment levels, as soon as possible, after being registered in an issue tracking system [Mockus et al., 2002; Tan and Mookerjee, 2005; Liu et al., 2012]. Therefore, maintenance in open-source systems usually does not take benefit of the gains of scale that happen when requests are handled in batch [Tan and Mookerjee, 2005].

[1] In this thesis, we preferably use the term bug or issue to refer to maintenance work items such as tasks, requests, reports, defects, enhancements, tickets, etc. The main reason is that bug is the term used in the Mozilla systems we analyze in this research. We also use issue because it is a very common terminology among related work.

Another problem faced by popular open-source systems is related to the issue reporting process. Usually, certified developers, as well as common users, can report any fault perceived in the software. However, this process is not coordinated, which results in a high amount of reported issues from which many are invalid or duplicated [Liu et al., 2012]. In 2005, a certified maintainer from the Mozilla Software foundation made the following comment on this situation: "everyday, almost 300 bugs appear that need triaging. This is far too much for only the Mozilla programmers to handle" [Anvik et al., 2006]. Our data indicates that, in 2012, the number of reported issues for the Mozilla projects increased approximately 110% when compared to 2005. In this context, any process or tool that improves issue processing can provide a relevant impact on the maintenance and evolution of open-source systems.

Furthermore, software developers make several changes of context in a typical workday. These changes normally happen due to meetings, mails, instant messaging, etc., or when a given task is concluded and they need to choose a new task to work on. Regardless the reasons, the negative effects of context changes on developers' productivity are well-known and studied. For example, in a recent survey with industry developers, more than 50% of the participants answered that a productive workday is one that flows without context changes and having no or few interruptions [Meyer et al., 2014]. Another study shows that developers spent at least two thirds of their time in activities related to acquiring task context, i.e., searching, navigating, and understanding the code relevant to the task at hand [Ko et al., 2005]. We argue that this time can be reduced if developers consistently decide to work on a new task similar to a previously concluded one. More specifically, context changes are reduced by guiding developers to work on a set of bugs B_0, B_1, ..., B_n, where B_i requires changes on parts of the system related to a previous bug B_{i-1}, for $i > 0$. By following this workflow, context changes are mitigated because the order of the bugs naturally reproduces the work performed under a periodic maintenance policy, i.e., a bug that is selected, comprehended, and fixed at a given time helps on further bug corrections. The same principle also applies when fixing such bugs simultaneously.

Therefore, we hypothesize that developers contributing to software projects can work more efficiently if support is provided to work on similar bugs, i.e., bugs requiring changes in the same parts of the system. This hypothesis is better suited for open source projects where developers have the liberty to choose most of their work [Mockus et al., 2002]. By working on similar bugs, developers avoid the effort of locating and understanding different code fragments on each bug they volunteer to work on [Kersten and Murphy, 2006; Parnin and Rugaber, 2011]. Moreover, since hundreds of bugs appear daily in large open source projects, developers may not notice bugs in the

tracking system that are very similar to one she is currently working on (or planning to work on). Therefore, productivity can also increase by reminding developers of other bugs that are potentially similar to one she is currently browsing in the tracking system.

To clarify our motivation, consider the following Mozilla bug that was fixed by a developer on February 12, 2009:[2]

Bug 475327 - [Linux] New Tab button is still right side of the Tab bar

When the developer fixed this bug, another bug with a similar description was also available in Mozilla's tracking system.

Bug 474908 - Dragging a tab in a window with only one tab detaches the tab

This second bug was fixed by the same developer of the first one, 24 days after the first bug. Fixing the second bug required changes in one XML file that also was changed by the developer when fixing the first bug. In the interval between working on these bugs, the respective developer remained active, fixing another 12 non-related bugs. To find out how often such a situation occurs, we retrospectively analyzed bugs reported for Mozilla systems from January 2009 to October 2012. For 67% of the bugs, just after the developers fixed a bug B, they chose to work on a bug B'' even though a more similar bug B' in terms of changed files was available in the tracking system. As a result, for 43K Mozilla bugs, a developer may have required less context changing and been more productive by choosing a different bug.

1.2 Objective and Contributions

Our main objective is to *design and evaluate an approach to recommend similar bugs to maintainers* based on the textual description of each bug stored in issue tracking platforms, such as Bugzilla and Jira. The recommendations should be presented each time a developer browses the page containing the main description of a bug, with a minimal runtime cost. The intention is to remind the developer about bugs similar to the one he/she is browsing. In our previous motivating example (Section 1.1), the page describing Bug 475327 should be extended with a list of bugs with a similar textual description, like Bug 474908.

More specifically, suppose that a developer manifests interest in a bug with a short description q (a textual document). In this case, we intend to rely on text

[2] `https://bugzilla.mozilla.org/show_bug.cgi?id=475327`, verified 2016-05-16.

mining techniques to retrieve open bugs with short descriptions d_j similar to q and to recommend them to the maintainers.

We claim the proposed approach is compatible with the current software maintenance process followed by open-source systems for the following reasons: (a) it is based on recommendations, and therefore maintainers are not required to accept extra bugs to fix; (b) it is fully automatic and unsupervised and therefore does not depend on human intervention; (c) it relies on information readily available in issue tracking systems and therefore requires little effort to compute. Despite that, assuming the recommendations effectively capture similar bugs, we claim they can contribute to introduce gains of scale similar to the ones achieved with periodic maintenance policies.

The proposed recommender is based on techniques originally designed for detecting duplicated bug reports. Even tough contemporaneous and more complex techniques for detecting duplicate bugs usually perform better than simpler ones, our results show this is not the case for similar bugs recommendations. More specifically, our results show that the technique proposed in the thesis—which is lightweight and simple—can perform just as well as more complex techniques. Therefore, our focus is to analyze the feasibility of similar bugs recommendations and not to argue that our approach fosters productivity gains, even though it is possible to see indirect evidence of that effect.

Our inspiration for this work are the recommendation systems widely used by online stores (e.g., Amazon)[3] which serves as an interesting analogy to our approach. Online stores recommend other "related" products whenever you browse a product. For example, when we browse a book on Amazon, other similar books on related subjects are also presented (Figure 1.1). Even though not every client accepts the recommendations, those that do contribute to increase the store's sales. Similarly, we propose in this thesis a recommendation system to improve the number of bugs handled by maintainers of open source systems.

The intended contributions of this thesis are: (a) a novel application of bug mining techniques to recommend similar bug reports to developers; (b) a tool to enhance a bug tracking system with recommendations of similar bugs; (c) a lightweight technique to characterize the maintenance process followed by software developers, which showed the importance of supporting developers on finding more bugs of interest; (d) a quantitative study on a large-scale bug repository, which compares our approach with a state-of-the-art technique and shows that it is possible to predict similar bugs; (e) a survey study with Mozilla developers, which showed that developers can recognize similar bugs recommendations.

[3] `http://www.amazon.com/`, verified 2016-05-16

Figure 1.1: Amazon screenshot adapted to show the books' recomendation.

In the quantitative study, we analyze 65K Mozilla and 2.6K Mylyn bugs by applying our recommender technique. We check whether the recommendations provided by the recommender share a subset of the same source files. The results show that we were able to predict bugs whose fixes indeed share the same source files, and as a consequence, context change can be mitigated by working on them.

In the qualitative study, we surveyed 66 Mozilla developers, by presenting them to similar bugs recommendations (that were still unresolved in the system) and by asking them two questions. The main goal of this survey is to verify if developers can recognize similar bugs recommendations. We also plan to reveal the developers opinion on extending their bug tracking system with similar bugs recommendations.

1.3 Publications

The work described in this thesis includes material from the following publications:

- Henrique Rocha, Guilherme de Oliveira, Humberto Marques-Neto, Marco Túlio Valente. Characterizing Bug Workflows in Mozilla Firefox. 30th Brazilian Symposium of Software Engineering (SBES), pages 1-10, 2016.

- Henrique Rocha, Marco Túlio Valente, Humberto Marques-Neto, Gail Murphy. An Empirical Study on Recommendations of Similar Bugs. 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER), pages 46-56, 2016.

- Henrique Rocha, Guilherme de Oliveira, Humberto Marques-Neto, Marco Túlio Valente. NextBug: A Bugzilla Extension for Recommending Similar Bugs. Journal of Software Engineering Research and Development (JSERD), vol. 3, issue 1, pages 1-14, 2015.

- Henrique Rocha, Guilherme de Oliveira, Humberto Marques-Neto, Marco Túlio Valente. NextBug: A Tool for Recommending Similar Bugs in Open-Source Systems. Brazilian Software Conference (CBSoft) – Tool Track, pages 53-60, 2014 (best tool award).

- Henrique Rocha, Humberto Marques-Neto, Marco Túlio Valente. Agrupamento Automático de Solicitações de Manutenção. First Latin-America School of Software Engineering (ELA-ES), pages 9-9, 2013 (3rd best PhD work award).

The following publications represent earlier research efforts during this Ph.D:

- Henrique Rocha, Cesar Couto, Cristiano Maffort, Rogel Garcia, Clarisse Simoes, Leonardo Passos, and Marco Tulio Valente. Mining the impact of evolution categories on object-oriented metrics. Software Quality Journal, vol. 21, issue 4, pages 529-549, 2013.

- Hugo Brito, Humberto Marques-Neto, Ricardo Terra, Henrique Rocha, and Marco Tulio Valente. On-the-fly extraction of hierarchical object graphs. Journal of the Brazilian Computer Society, vol. 19, issue 1, pages 15-27, 2013.

- Henrique Rocha and Marco Tulio Valente. How Annotations are Used in Java: An Empirical Study. In 23rd International Conference on Software Engineering and Knowledge Engineering (SEKE), pages 426-432, 2011.

1.4 Outline of the Thesis

We organized the remainder of this work as follows:

- **Chapter 2** describes background information on software maintenance, maintenance policies, issue tracking systems, and information retrieval techniques. We

also discuss work on issue reports, including characterization and visualization techniques, work aiming to detect duplicate issues, to assign issues to developers, and to group similar issues.

- **Chapter 3** describes the Mozilla maintenance process focusing on its bug handling workflow. We analyze and characterize a large dataset of Mozilla bugs. We also present and discuss a lightweight technique, called Bug Flow Graphs, aimed to better understand and visualize the workflow followed in the maintenance of open source systems. Finally, we perform a characterization study focusing on the workflow followed when fixing bugs in the Mozilla ecosystem.

- **Chapter 4** presents our approach to recommend similar bug reports to developers. We also present a supporting tool, called NextBug, implemented as a Bugzilla addon.

- **Chapter 5** presents a retrospective study to evaluate our approach in a quantitative dimension. First, we compare NextBug against REP [Sun et al., 2011], a state-of-art technique to detect duplicate bugs. Second, we show how NextBug performs on a smaller system (Mylyn), which has fewer bugs.

- **Chapter 6** shows the results of a field study conducted with Mozilla developers, where they were presented with similar bugs provided by NextBug and asked a few questions about them.

- **Chapter 7** presents the main conclusions and outlines future work ideas.

Chapter 2

Background and Related Work

In this chapter, we present background information related to this thesis. First, we present central concepts on Software Maintenance (Section 2.1) and Information Retrieval (Section 2.2). Then, we present work on bug characterization and visualization (Section 2.3) that is related to our characterization study in the next chapter. We also discuss techniques to detect duplicated bug reports (Section 2.4), and to assign bug reports to developers (Section 2.5), which are related to our work. Moreover, we present a brief review on studies that also work with similar bug reports (Section 2.6). Besides, we provide an overview on Recommendation Systems on Software Engineering (Section 2.7), since our approach is based on recommendation system principles. Finally, we conclude with general remarks on the discussed topics (Section 2.8).

2.1 Software Maintenance

Software maintenance is an important activity in a software's life cycle. Basically, it includes all the tasks performed to modify a software while preserving its integrity [ISO/IEC 14764, 2006; ISO/IEC 12207, 2008]. Normally, understanding the software which is being modified corresponds to a great part of a maintenance effort [Tan and Mookerjee, 2005]. The importance of maintenance is well known both in the academia and in the industry. The Lehman and Belady [1985] laws point out that a system must be continuously adapted otherwise it becomes progressively less satisfactory until it is discarded. The complexity of maintenance tasks is also well-known. Usually, maintenance activities are one of the most problematic parts of a software life cycle due to their inherent complexity [Tan and Mookerjee, 2005; Ahn et al., 2003; Heales, 2002].

Software maintenance is also a costly process [Tan and Mookerjee, 2005]. There

are several studies that highlight and discuss the costs associated with maintenance tasks. Coleman et al. [1994] claim that 40% to 60% of production costs are devoted to maintenance. Other study presents a worse case scenario, showing that software maintenance activities may consume up to 80% of the total costs during a system's life time [Alkhatib, 1992]. Erlikh [2000] shows that for legacy systems up to 90% of the system's resources are consumed by maintenance tasks. There are also studies that highlight the increasing amount of money spent in software maintenance activities. In the 1980's, it was estimated that annually US$ 30 billion was spent with maintenance tasks worldwide [Chan et al., 1996; Swanson and Beath, 1989]. In the 1990's, IEEE estimated that US companies expended over US$ 70 billion per year with software maintenance [Sutherland, 1995]. We did not find recent studies with more current estimates on the costs of maintenance activities. However, there are no evidences that they have decreased.

Software maintenances is typically performed based on a proposed Modification Request (or Change Request). According to ISO/IEC 14764 [2006] these requests are classified into two categories: *correction* or *enhancement*. Correction activities can be sub-categorized into *corrective* or *preventive* tasks. Enhancement activities are also sub-categorized further into *adaptive* or *perfective*. Figure 2.1 shows an overview of this classification [ISO/IEC 14764, 2006].

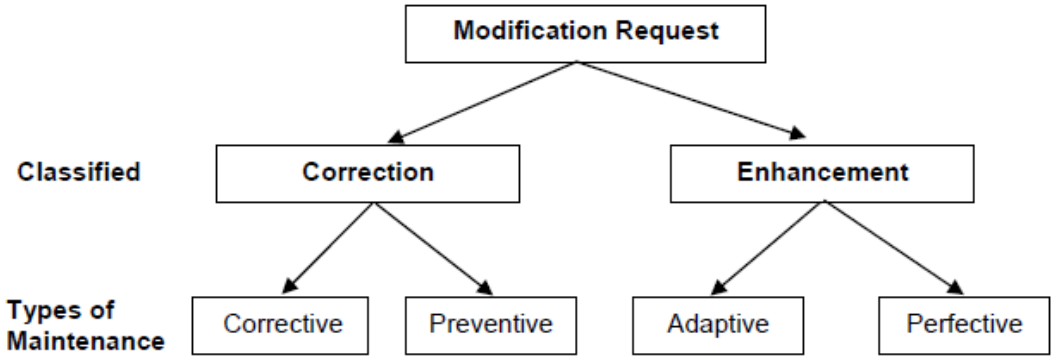


Figure 2.1: Modification Requests classification. Source: ISO/IEC 14764(2006)

Correction maintenance involves the modification of a software to fix a problem. In corrective tasks, the problem is only discovered after the software product is released to end-users. Frequently, corrective maintenance requests represent a significant portion of all requests [Tan and Mookerjee, 2005]. On the other hand, preventive maintenance tasks try to anticipate the problems before they become operational faults perceived by the users. Enhancement maintenance is the modification of a software to keep its value when dealing with a changing environment. Adaptive tasks provide nec-

essary changes in which a software must operate and perfective tasks intend to improve the software beforehand.

There are two policies to deal with software maintenances: continuous or periodically. Under a continuous policy the modification requests are handled in a ad-hoc way, i.e., as soon as possible after the assignment of a request to a maintainer. On the other hand, under periodic policies, the maintainers handle the software maintenance at predefined time intervals and, thus, several requests are addressed together. Open-source systems most often employ a continuous maintenance policy, while organizations prefer to adopt a periodic policy [Tan and Mookerjee, 2005; Junio et al., 2011].

As this thesis focus on maintenance tasks addressed following a periodic policy, we discuss the central characteristics of such policies in Section 2.1.1.

2.1.1 Benefits of Periodic Maintenance

Most open-source systems adopt an uncoordinated reporting process, where both users and testers can report a modification request [Liu et al., 2012]. This practice usually results in a continuous maintenance process, where maintainers address the change requests a soon as possible.

On the other hand, commercial software is often maintained under a periodic policy. Basically, the preference for this policy happens due to two reasons: *scale economics* and *software degradation* [Tan and Mookerjee, 2005]. The first reason is the gain obtained with scale economics. Periodic policy provides scale economics by reducing the fixed costs inherent to maintenance activities by 36% [Banker and Slaughter, 1997]. Other reason for following periodical policies is the mitigation of system degradation, since each maintenance activity may contribute to the structural degradation of a system, as pointed out by Lehman and Belady [1985]. Thus, instead of performing maintenance tasks after each change request, it is better to optimize the tasks implementing more change requests in each maintenance cycle [Tan and Mookerjee, 2005].

Banker and Slaughter [1997] conducted an empirical evaluation to measure software maintenance productivity. They investigated how maintenance can be improved by analysing maintenance projects in a large financial services organization. Their findings indicate that scale economics can be achieved by clustering maintenance requests into larger projects. On the other hand, as pointed out by the authors, batching several maintenance requests may incur on opportunity costs for delaying the correction. According to the authors, grouping maintenance requests has the potential to reduce the maintenances costs up to 36%. Tan and Mookerjee [2005] presented a

study comparing two types of periodical maintenance policies: *uniform* and *flexible*. Uniform policy performs periodical maintenance tasks at fixed time intervals, while flexible policy perform maintenance tasks at varying intervals. The authors propose a model that considers previous empirical studies to reduce the costs of maintenance and replacement activities. They evaluated several parameters and scenarios on periodical maintenance such as: percent of reuse of original code, system degradation, waiting costs, productivity improvements, and useful life of the system. The results showed several situations where flexible policies perform better than uniform ones.

Junio et al. [2011] evaluate the advantages of grouping maintenance requests to achieve scale economics. They proposed a process, called PASM, that fosters the usage of best software engineering practices when handling maintenance requests. The authors applied their approach in the information technology department at PUC Minas. The results showed that after adopting PASM, maintainers spent more time on software engineering activities and less on codification tasks. Marques-Neto et al. [2013] proposed a quantitative approach to evaluate software maintenance services. Their approach is an adaptation of a method originally proposed to characterize the workload of e-commerce services. They rely on clustering techniques to generate a compact model that represents maintenance tasks workflow. The authors claimed the approach can help organizations to better understand and improve their own maintenance process.

2.1.2 Context Changes

In software development, developers are more productive when focusing on related tasks and topics. Context changes may interrupt a developer focus and, therefore, affect his/her productivity. In fact, the negative impacts of context changes have been revealed by several researchers [Meyer et al., 2014; Murphy, 2014]. Most developers agree that context changes are bad for their productivity, as 72% responded in a survey [Meyer et al., 2014]. Moreover, over 50% of software developers considered few interruptions and context changes as a definition for a productive day [Murphy, 2014]. Ko et al. [2005] measure the amount of time developers spend understanding the code when working on maintenance tasks. They measured the percentage of time developers spend on activities such as reading code or API, and navigating dependencies. The results showed that, on average, developers spend two thirds of their time in activities related to task context.

After analyzing 10,000 programming sessions of 86 programmers, Parnin and Rugaber [2011] observed that only 7% of the sessions do not require navigation to other locations prior to editing. One approach introduced to reduce the cost of context

changing is Eclipse Mylyn [Kersten and Murphy, 2006]. As a developer works on an indicated task, Mylyn tracks the artifacts touched and changed, building a degree-of-interest model that describes how important each part of each artifact is to the task. This model is stored per task and when a task change occurs, Mylyn can re-display to a developer the artifacts earlier worked on as part of that task. Cassandra is a task recommender that aims to schedule the maintenance work in order to minimize conflicting changes in parallel software development [Kasi and Sarma, 2013]. The system relies on Mylyn to get contextual data, which is communicated to a centralized scheduler component. This component identifies potential conflicts in order to recommend task orders that restrict dependent tasks that share common files from being concurrently edited.

We claim that the recommendation of similar bus proposed in this thesis also helps to reduce context changes and consequently improve developers' productivity. If developers consistently work on similar tasks (either sequentially or concurrently), their time caused by context changes will decrease.

2.1.3 Issue Tracking Systems

Issue tracking systems (ITS) are software systems that maintain a dataset of maintenance modification requests (i.e., problems, issues or bug reports) for one or more projects. Usually, these systems provide a web interface that allows users to interact with the tracking system and the reported issues [Anvik et al., 2006]. Such interaction may include different operations like reporting a new issue, sending an attachment file, or providing comments for an issue.

Most open-source projects use an ITS to support their maintenance processes. The ITS provides a central knowledge repository about the issues handling progress. It also serves as a communication channel for geographically distributed developers and users [Anvik et al., 2006; Ihara et al., 2009]. Currently, there are several ITSs that are used in software maintenance and development. Following, we describe some of the most popular systems:

- **Bugzilla:**[1] is a free ITS developed by Mozilla, implemented in Perl. According to Bugzilla web site, there are over 1200 organizations or projects using its system.

- **Jira:**[2] is a commercial system developed by Atlassian. Jira also provides management features integrated with its bug tracking features.

[1] `http://www.bugzilla.org`, verified in 2016-05-16.
[2] `https://www.atlassian.com/software/jira`, verified in 2016-05-16.

- **Mantis:**[3] is a another free and open-source ITS implemented in PHP.
- **Trac:**[4] is an open-source ITS that supports integration with version control platforms such as Subversion, Git, Mercurial, among others. Trac also supports some project management tools.
- **RedMine:**[5] is another ITS that provides project management features along with issue tracking services. Another important feature is integration with version control systems.

The recommender proposed in this thesis uses the Bugzilla repository. However, it is possible to generalize our proposal to other ITSs that maintain similar issue data.

2.2 Information Retrieval

Information retrieval (IR) deals with the representation, storage, and organization of structured or unstructured data [Baeza-Yates and Ribeiro-Neto, 1999; Raghavan and Wong, 1986]. As IR techniques handle textual documents [Baeza-Yates and Ribeiro-Neto, 1999; Wang et al., 2008], and most artifacts required by a software maintenance process—including code—are textual, IR techniques can be used to handle such documents [Marcus and Menzies, 2010]. Our approach relies on IR techniques to extract semantic information from issue reports.

Usually, applications that employ IR techniques need to be properly modeled. This modeling consists in building a logical framework for representing the documents in the dataset. An IR model also defines a ranking function to quantify the similarity between documents and queries (user information needs). The ranking function is used to classify the documents, ordering by their relevance to the query (Figure 2.2).

Formally, an IR model is as a quadruple $[D, Q, F, R(q_i, d_j)]$ where D is a logical representation for the documents in the dataset, Q is a logical representation for the user queries, F is a framework for modeling documents and queries, and $R(q_i, d_j)$ is a ranking function between a query q_i and a document d_j.

Most IR models adopt index terms to represent the data (documents and queries). In a general form, any word with a semantic significance can be an index term. Thus, index terms summarize the textual content adding simplicity to the documents and queries representation. Since not every term is equally useful to represent the data,

[3] `http://www.mantisbt.org`, verified in 2016-05-16.
[4] `http://trac.edgewall.org`, verified in 2016-05-16.
[5] `http://www.redmine.org`, verified in 2016-05-16.

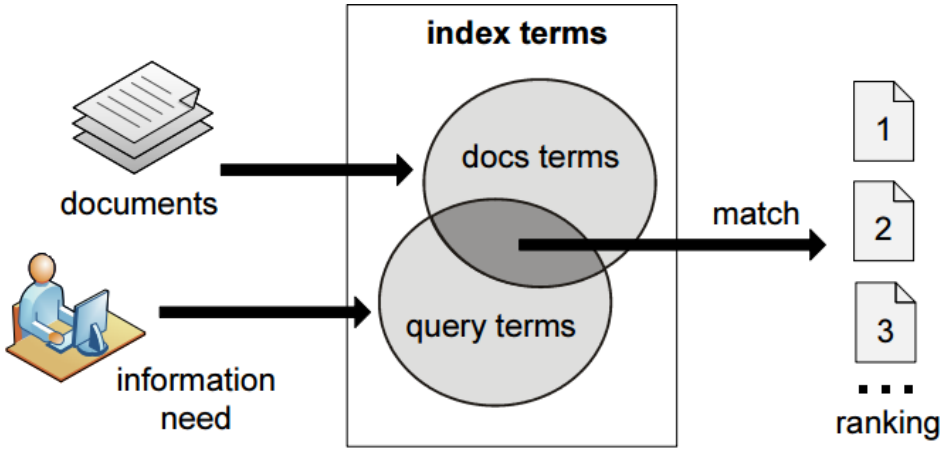


Figure 2.2: Modeling in the Information Retrieval Process.

it is a common step for several models to assign numerical weights to each indexed term [Raghavan and Wong, 1986].

2.2.1 Pre-processing Index Terms

Some works recommend to apply pre-processing techniques to better generate the index terms and improve the data representation and the overall IR process [Baeza-Yates and Ribeiro-Neto, 1999; Runeson et al., 2007; Wang et al., 2008; Sun et al., 2010]. Pre-processing techniques usually include: tokenization, stop-words removal, and stemming. Tokenization is the process of receiving a sequence of characters as input and producing a stream of tokens as output. A token is a string of alphanumeric characters without delimiters [Sun et al., 2010]. Stop-words are very common words occurring in most documents in the dataset and, because of that, their information for IR has little significance. These words occur in every document and have little relation to the documents textual content. Most of these words are prepositions, conjunctions, or pronouns, such as: the, of, that, etc. If stop-words are not removed from the documents, they could negatively affect the ranking function [Wang et al., 2008; Runeson et al., 2007]. As a document may contain different grammatical forms of a word, stemming tries to reduce each word to its radical form by removing affixes and other lexical components. The reason to perform stemming is because different forms of words generally have similar semantic information. For instance, the stemming process would reduce “crashing” and “crashed” into “crash”, and an automated process could easily identify both words as the same data [Sun et al., 2010; Runeson et al., 2007].

2.2.2 Vector Space Model

Previous studies that apply IR techniques to issue reports have adopted the vector space as its IR model [Ko et al., 2006; Runeson et al., 2007; Wang et al., 2008; Alipour et al., 2013]. The vector space model (VSM) is a classic modeling approach to process documents and queries by decomposing them into t-dimensional vectors, where t is the number of different index terms in the document collection. In the classical VSM, the index terms are assumed to be mutually independent [Baeza-Yates and Ribeiro-Neto, 1999; Raghavan and Wong, 1986]. The weights w_i are positive real numbers that represent the i-th indexed term in the vector. The weights calculation is based on the following observations: (i) high frequency terms are more important for describing documents (term frequency – tf_i), and (ii) terms that occur in most documents are less important to discriminate relevant documents (inverse document frequency – idf_i). The Equation 2.1 uses the term frequency and the inverse term frequency to calculate w_i, which is referred as *tf-idf* weighting scheme [Baeza-Yates and Ribeiro-Neto, 1999].

$$w_i = tf_i \times idf_i = (1 + \log_2 f_i) \times \log_2 \frac{N}{n_i} \tag{2.1}$$

where f_i is the frequency of the i-th term in the document, N is the number of documents in the collection, and n_i is the number of documents in which the i-th term occurs.

The similarity between the vectors of a document d_j and a query q is described by the ranking function in Equation 2.2:

$$Sim(d_j, q) = \cos(\Theta) = \frac{\overrightarrow{d_j} \bullet \overrightarrow{q}}{||\overrightarrow{d_j}|| \times ||\overrightarrow{q}||} = \frac{\sum_{i=1}^{t} w_{i,d} \times w_{i,q}}{\sqrt{\sum_{i=1}^{t}(w_{i,d})^2} \times \sqrt{\sum_{i=1}^{t}(w_{i,q})^2}} \tag{2.2}$$

This function is often called cosine similarity because it measures the cosine of the angle between two vectors as illustrated in Figure 2.3. Since all weights are greater or equal to zero, we have $0 \leq Sim(d_j, q) \leq 1$, where zero indicates that there is no relation between the two vectors, and one indicates the highest possible similarity, i.e., both vectors are actually the same.

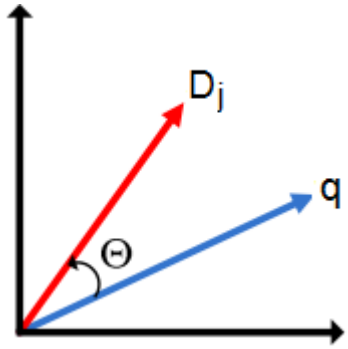


Figure 2.3: Cosine Similarity.

2.3 Bug Characterization and Visualization

In this section, we present works aiming either to characterize or to visualize bugs. These work share common goal to help developers better understand the bugs reports and the maintenance process.

Guo et al. [2010] analyse the factors that could affect the fixing of bugs in Windows 7 and Vista. They also performed a survey with 358 developers, which showed how reputation and seniority affect the chances of bugs to be fixed. Based on the characterization, the authors propose a prediction model to identify the probability of a new bug to be fixed. Their prediction showed 68% of precision and 64% of recall for Windows 7 bug fixes.

Zimmermann et al. [2012] characterize which bugs are reopened for Windows 7 and Vista. This is a continuation on their previous work on characterizing fixed bugs [Guo et al., 2010]. Basically, they show the impact of different bug report features on the probability of bug reopening. Joorabchi et al. [2014] characterize non-reproducible bugs from six systems, for a total of 32K bugs. Their study tries to understand the causes for a bug to be marked as non-reproducible. They classify non-reproducible bugs into six categories, and show workflow patterns for them.

D'Ambros et al. [2007] propose an approach to follow the evolution and maintenance of systems by looking at the bug life cycle. They present two types of visualization techniques: *system radiography* and *bug watch*. Dal Sassc and Lanza [2013] propose a web visual analytics platform, called in*Bug, to help visualize bug reports stored in tracking systems. Jeong et al. [2009] investigate the process called Bug Tossing, i.e., when bug reports are assigned to the "wrong" developer and later they are reassigned to a more proper developer. The graphical representation from Jeong et al. [2009] focus only on the bug tossing, which shows developers and their relations.

Minelli and Lanza [2013] developed a tool, called DFlow, to visualize the workflow of developers. DFlow is implemented in Pharo Smalltalk, and it analyses the developers tasks inside an IDE.

Ihara et al. [2009] propose a graph based visualization technique to analyze bug resolution process. They propose a general model and the bug workflow is adapted to fit such model. Therefore, it is a more generic and automated technique. However, the adaption do not accurately represent all the states and transitions of the workflow followed by developers.

Critical Assessment: The presented characterization studies analyze a specific type of bug report (fixed, reopen, or non-reproducible) to understand the bugs and them work towards a prediction model. The visualization techniques tries to help managers and developers by showing the bugs to better analyze and understand the maintenance process.

2.4 Detecting Duplicated Bug Reports

Recent studies have focused on finding duplicated issue reports in bug tracking systems. Duplicated reports can hamper the bug triaging process and may waste maintenance resources [Cavalcanti et al., 2013]. Typically, studies for finding duplicate issues employ traditional information retrieval techniques such as natural language processing, vector space model, and cosine similarity. For example, Ko et al. [2006] analyze the linguistic characteristics of bug reports. They proposed an approach that uses natural language processing to parse the bug reports' summaries and suggest a better structured report. They also claim that their approach can help in finding similar bug reports. Although, they did not provide any experimental evaluation on this.

Runeson et al. [2007] worked only with natural language data in their approach to find duplicated reports. They used VSM and measured similarity using three different metrics: Cosine, Jaccard, and Dice. Their experiments showed better results using the Cosine Similarity when compared to the other two. To evaluate the approach, they measured only recall rate achieving around 31% for their top-5 list of duplicated reports. They also conducted interviews with maintainers who evaluated positively the approach.

Wang et al. [2008] used a trace with execution data with issues' reports to find duplicated reports. The authors tested three heuristics combining both execution data and natural language information. After the experiments they chose a Classification-

Based Heuristic which favors the natural language information. The natural language processing uses the classic Vector Space Model (VSM) and Cosine Similarity to process the bug reports. They also analyzed the impact of using summaries and full descriptions to detect duplicates. Their dataset was composed by bug reports from two open-source systems: Eclipse and Firefox. The Eclipse dataset has 220 bug reports, from which 44 were duplicates manually inserted for testing. The Firefox dataset is composed by 1,492 bug reports. In this dataset, they used the first 50 days of reports as a training set (744 bugs) and the remaining as a test set (754 bugs). The authors measured high recall rate values (between 55% and 95%) for their approach.

Jalbert and Weimer [2008] proposed an approach for bug duplication detection that uses a model classifier employing textual semantic information, linear regression, and graph clustering. The textual information processing follows different steps when compared to other bug duplication studies. The authors used a different weight formula and they not used an Inverse Term Frequency (IDF). They performed a statistical analysis that indicates the IDF weight gives worst results for the bug duplication problem. They used a graph clustering technique originally designed for social networks and applied it to issue reports. In this case, the bug reports represent the nodes and the edges connect nodes with similar textual information. The authors build a classifier based on linear regression that uses the clusters and textual information as features. The evaluation was based on Mozilla bug reports from February 2005 to October 2005, for a total of 29,000 bug reports. They evaluated their approach measuring the recall rate, which performed only 1% better than the approach considered as baseline [Runeson et al., 2007].

Sun et al. [2010] proposed an approach to detect duplicate reports by using discriminative models. First, they preprocessed each issue using standard techniques (stemming, stop-words removal, etc.) and organized them into a hash-map like structure. Then, the approach trains a classifier based on a Support Vector Machine (SVM) algorithm. A noteworthy aspect of this work is the analysis of features selection and several possible similarity measurements. Their evaluation showed a recall rate around 65% for a recommendation list with 20 reports. According to the authors, this recall rate represents an improvement up to 43% over previous techniques.

Sun et al. [2011] proposed a retrieval function, called REP, to measure the similarity between bug reports and to improve the detection of duplicate reports. The REP function analyses both textual and categorical information of each report. The authors also extended the BM25F [Robertson et al., 2004] method that considers the query weights. They applied their method to bug reports from three open-source systems: Eclipse, Firefox and OpenOffice. The Eclipse dataset was composed by over

200K bug reports. Their evaluation measures the recall rate@k and the mean average precision. They show a small improvement of their extended BM25F method over the classic one. We compare REP to our proposed recommender in Chapter 5. We also describe REP in more detail in Section 5.1.2.

Tian et al. [2012] proposed another approach to classify whether a bug report is a duplicate or not. Initially, the authors discussed that there are basically two approaches for duplicated report detection: report retrieval and report classification. Report retrieval searches for duplicate reports in a similar way as information retrieval searches for documents. Report classification address the task as a classification problem and assign a label (duplicate or not) to new reports. The authors claim that such lines complement each other. Despite that, their approach falls on the classification category. They implemented several extensions over the work of Jalbert and Weimer [2008] such as: (i) a variant of the BM25F method, (ii) a relative similarity measure to better differentiate a duplicate report from a similar one, and (iii) using the product information on each bug report to improve the similarity measurement. They also compared their results to the work of Jalbert and Weimer [2008], and their approach showed a better accuracy and F-score. Their evaluation results showed accuracy and F-score values of 24% and 38% respectively.

Nguyen et al. [2012] proposed an approach, called DBTM, to detect duplicate bug reports that combines features provided by information retrieval and topic-based techniques. The authors used a machine learning generative process called Latent Dirichlet Allocation (LDA) to extract the topic-based information presented in bug reports. The information retrieval technique used to extract the textual information was the BM25F. Both techniques were combined using another machine learning technique called Ensemble Averaging. They used the same dataset presented by Sun et al. [2011]. The authors measured only the recall rate in their evaluation and their results were better than the ones from Sun et al. [2011].

Liu et al. [2012] proposed an approach to improve the search quality in bug tracking systems. The authors claim that searching for reports is an important step for bug triaging and to find duplicate reports. The approach uses a supervised machine learning technique to better rank the bug reports and therefore improve the search quality. Their evaluation measured the average rank of the results, mean average precision, and F-score. They achieved better values for all measured metrics over other searching techniques.

Alipour et al. [2013] analyzed approaches for detecting duplicated bugs using information retrieval techniques. They also proposed a more accurate approach for detecting duplicated issues by exploring the bug's contextual information along with

the software architecture and quality guidelines. The approach uses BM25F [Robertson et al., 2004] method and cosine similarity to retrieve the textual, categorical, and contextual information. The authors then compare the following classifiers algorithms: 0-R, C4.5, KNN (nearest neighbors), Logistic Regression, and Naive Bayes. The dataset used for their experiments is composed by five years of bug reports from the Android operating system, for a total of 37,236 reports with 1,063 duplicates. Their evaluation measured accuracy (precision), kappa (which measures the agreement between the classifier and the classes), and the area under a Receiver Operation Characteristic (ROC) curve. They report that the algorithm C4.5 achieved the better results, presenting high precision values (above 84%).

Cavalcanti et al. [2013] presented an exploratory study on bug repositories and the task of finding duplicate bug reports. The authors used nine software projects with different characteristics. The bug duplication problem was found in all of the studied projects. Their work showed many guidelines that may contribute to reduce the number of duplicates and improve the techniques used to detect them.

Critical Assessment: the presented works aim to identify duplicate issue reports by using information retrieval techniques. Most of them rely on the Vector Space Model and on standard pre-processing techniques (e.g., stemming) to extract and model the information in each issue. The similarity measurement is usually either a cosine or a customized BM25F. None of the discussed works deal with similar reports.

2.5 Assigning Bugs to Developers

Approaches to assign the most suitable developer to correct a software issue are also reported in the literature. Anvik et al. [2006] proposed a recommendation system to assign bug reports to developers. Their approach is based on supervised machine learning that requires training to create a classifier. This classifier assigns the data (bug reports) to the closest category or class (developer). The authors evaluated the performance of three machine learning algorithms: Support Vector Machines (SVM), C4.5, and Naive Bayes. They also verified which recommendation list size brings better results. After their experiments, they recommended the SVM algorithm and a list size of only one recommendation. The dataset used in the experiments is composed by issue reports for Eclipse and Firefox from September 2004 to May 2005. The training set used 8,655 and 9,752 bug reports from Eclipse and Firefox, respectively. The test set was composed by bug reports from May 2005, using 122 bug reports from Eclipse and

22 bugs from Firefox. They achieved low recall values (bellow 10%) but their precision was high (around 60%).

Anvik and Murphy [2011] proposed another approach to recommend which developer to assign a bug report. This work is an extended version of the paper discussed in the last paragraph [Anvik et al., 2006]. One of the improvements is that the recommendation system suggests additional information besides the developers. The revised recommendation system also suggests which component the report should be assigned to and which other project members should be aware of the issue. Despite this, the approach remains basically the same as in the previous paper. However, the authors proposed a new set of heuristics to label each report and also analyzed six machine learning algorithms to improve precision. They concluded that the Support Vector Machine (SVM) performs better than the other algorithms and their precision increased considering their previous work (around precision 75% for the same systems).

Tamrawi et al. [2011] proposed an approach, called Bugzie, based on fuzzy sets to assign bug reports to developers. The authors claim that a bug report describes technical aspects related to the issue. Bugzie models the fixing expertise of developers to correct these technical aspects using fuzzy sets. They use the fuzzy membership function to rank developers to each indexed term and a combination formula to find the most capable fixer for a bug report. The approach incrementally updates the fuzzy sets as new bug reports are resolved. The Bugzie accuracy for top-5 recommendations is in the range of 70% to 83%.

Kagdi et al. [2012] proposed another approach to recommend suitable developers and also the source code part to resolve an issue report. Their work is an extension of the xFinder method proposed by [Kagdi et al., 2008]. They begin by extracting identifiers and comments from the source files to create a system corpus which is processed using the Latent Semantic Indexing (LSI) algorithm to create a indexed representation and to reduce dimensions. The issue's summary is used as a query for the LSI index. The result is a ranked list of source code units (classes, methods, etc.) that best represents the queried issue. The commit log history of these source code units in the version control system is then analyzed. The analyzed log is used to determine developers that corrected (committed) the source code. For the evaluation, the authors rely on an accuracy measure that resembles a likelihood metric. They conducted experiments on three open-source systems: ArgoUML, Eclipse, and KoOffice. They measured an accuracy for developer recommendations between 47% and 96% for bug reports and between 43% and 60% for feature requests.

Poshyvanyk et al. [2012] also proposed an approach to recommend the most suitable developer to handle an issue report. Unlike other works, their approach does

not require mining the bugs or commits repository and does not require training. The source files are preprocessed using Latent Semantic Indexing (LSI) to create an indexed corpus. The new arriving issue report is queried on the corpus to find the most related source files. Thus, the header comments in such source files are used to extract the developer information for the issue report. For the evaluation, they used issue reports and source files from three open-source systems: ArgoUML, jEdit, and MuCommander. The number of issue reports tested ranged from 91 (ArgoUML) to 141 (jEdit), which may be considered a small set. They measured precision and recall and compared their approach to other works [Anvik et al., 2006; Kagdi et al., 2012]. Their approach performed 20% better for one test but on the others the difference did not have a statistical relevance.

Shokripour et al. [2013] proposed a two-phase approach to assign developers to issue reports. They indexed terms from the nouns presented in the source code files used to correct past issues. According to the authors, these index results are simpler than in other similar studies. They used the information presented in the version control system and the patch files to map which files were modified to correct each issue. The first phase consists in searching the index for the source files that are most likely to be changed to correct a new issue. In the second phase, they use the source files from the previous phase to find which developers corrected the files, and then recommend the appropriate developers to address the new issue. Their dataset is composed by two open-source software: Eclipse and Firefox. In their evaluation, they measured an accuracy of 89% for Eclipse and 59% for Firefox.

Naguib et al. [2013] propose an approach based on activity profile to recommend developers to work on an issue. First, they categorize the bug reports into topics using Latent Dirichlet Allocation (LDA) algorithm. Second, they mine historic data to analyze the developer's actions on reviewing, assigning, and resolving bugs to identify roles. After that, the authors associate the LDA topics with the developer's role. This creates an activity profile that reveals the developers areas of expertise.

Critical Assessment: There is limited common ground among the presented works to assign issue reports to developers. They use very diverse techniques: classifiers, fuzzy logic, LSI or LDA, and source code mapping using patch files and version control systems.

2.6 Analyzing Similar Issue Reports

There are few studies that deal with similar issue reports. Weiß et al. [2007] developed an approach that suggests the fixing effort (in person-hours) required to correct an issue. As argued by the authors, the implementation time to fix an issue is particularly challenging to predict because fixing a bug is a search process that involves understanding the source code, execution traces, states, and history. Their approach uses the nearest neighbor algorithm to group similar issues together and estimate the effort based on the average time spent fixing similar issues.

Wang et al. [2010] proposed an approach called Rebug-Detector that detects source code defects related to polymorphism. The authors claim that if a method in the source causes a defect, then overridden methods may also cause similar ones. The approach begins by extracting term information from bug reports. Then it locates the defected methods from the source code using the bug report information. After that, Rebug-Detector measures the similarity among all overridden methods related to the defective one. Finally, Rebug-Detector shows which methods may cause similar defects. The approach detected 61 defects in the experiments from which 21 are real defects and 10 are suspected ones.

Critical Assessment: In spite of the fact that those works identify similarity in issues or code fragments, none of them aim to recommend similar bugs to mitigate the context changes.

2.7 Recommendation Systems in Software Engineering

In the current software engineering scenario, developers are continuously exposed to excessive information and new technologies. Recommendation systems for software engineering (RSSE) aim to assist software engineers in handling the information overload that recently characterized the area [Robillard et al., 2010]. Normally, RSSEs are applied in the context of code or artifact reuse, improving development quality and even on writing good bug reports [Anvik et al., 2006; Holmes et al., 2005; Robillard et al., 2010; Zimmermann et al., 2004]. In this thesis, we claim that RSSEs can also be used in the context of software maintenance for recommending similar bug reports, aiming to avoid the change of context and, consequently, improving the productivity of the maintenance team.

Zimmermann et al. [2004] designed the ROSE recommendation system, which uses association rules to detect co-changes in source files. The evaluation strategy of the ROSE system is specially noteworthy. Unlike other studies that measure only precision and recall, the ROSE evaluation also measured feedback and likelihood which are more appropriate metrics for recommendation systems. Since these metrics are useful to evaluate any type of recommendation system, we employ these in our retrospective study (Chapter 5).

Holmes and Murphy [2005] presented a RSSE called Strathcona. This system helps developers in finding API code examples by analyzing the structural context of the source code. According to the authors their main contributions are: a set of heuristics to determine structural similarity, and to show the utility of structural similarity in software examples.

Some of the studies described in Section 2.5 can be considered recommendation systems that suggest the most capable developer to correct an issue. Even though several RSSEs have been proposed, to the best of our knowledge there is no recommendation system aimed to suggest similar bug reports. Usually, a maintainer who wants to correct related issues, must search for them manually, which is a time consuming task that may dissuades developers to work on maintenance requests on a regular basis.

2.8 Final Remarks

In this chapter, we provided background information to better understand the solution proposed in this thesis. First, we presented the definitions, classifications, and costs of software maintenance (Section 2.1) and the benefits of periodical maintenance policies (Section 2.1.1). We also discussed context changes (Section 2.1.2) and their negative impact on software development, because our approach could contribute to decrease context changes and improve the overall developers' productivity. We also presented information about issue tracking systems (Section 2.1.3) because the data used in our experiments is extracted from such systems.In Section 2.2, we presented background information about information retrieval techniques since our approach relies on such techniques to find similar issues. In Section 2.3, we described research on bug characterization and we also presented tools to help visualize bugs. We analyzed these topics because, in Chapter 3, we present a characterization study and a visualization technique to better understand bug reports. In Sections 2.4, 2.5 and 2.6, we discussed several works that rely on issue reports as source of information. We investigate approaches proposed to detect duplicated reports which are closely related to our thesis.

We also discussed works conducted to assign the most capable developers to issue reports. We presented works that also attempt to group similar issues, however aiming to achieve very different goals from our thesis. To conclude, in Section 2.7, we presented a brief introduction on recommendation systems in software engineering.

Chapter 3

Bug Characterization Study

In this chapter, we present a characterization study focused on how open source systems handle bugs. Our goal is to shed light on the maintenance process followed by a major open source project and identify possible bottlenecks and improvement opportunities. Basically, our characterization relies on graphs—called Bug Flow Graphs (BFG)—that describe the workflow followed when resolving bugs.

The chapter is organized as follows. Section 3.1 describes the overall maintenance process followed by Mozilla systems (our selected open source ecosystem for this study), detailing Mozilla bug life cycle workflow. Section 3.2 describes the concept of Bug Flow Graphs. Section 3.3 presents and analyzes the dataset of Mozilla bugs used in this study. Section 3.4 reports the study results, using BFGs to analyze the maintenance workflow of Mozilla bugs. Section 3.5 presents threats to validity. Finally, Section 3.6 concludes the chapter.

3.1 Mozilla Maintenance Process

In Mozilla software, a maintenance request begins when someone posts an issue report in Bugzilla. This report includes pre-defined fields, natural language text, attachments, and dependencies. Bugzilla stores the issue reports information in a relational database (MySQL by default). After the new issue is reported and saved in Bugzilla repository, it must be analysed by a project member to confirm whether it is valid or not. This process is called triage (or bug triage), and the person responsible for the triaging is called the *triager* [Anvik and Murphy, 2011].

The Bugzilla user who submits an issue report is called *reporter*. When this user creates a new report, the following fields are automatically filled by the system and cannot be changed later: issue identification key, the *reporter*, creation date, and time.

Other fields can be filled by the reporter but may be changed later by the triager, such as: product, component, hardware, operating system, version, priority, and severity.

The most important fields rely on natural language text, such as full description, summary, and comments. The full description contains detailed information describing the issue and how someone can reproduce it. The short description, or summary, is a compact version of the full description highlighting the most important information on the issue. There is also an additional comments field, which contains discussions about the issue and possible approaches to solve the problem. The comments can also contain links to other issues with more information or duplicated reports. The *reporter* may attach a file in the issue report, which is usually a screenshot of the perceived bug or a patch containing the source code correction.

There are other fields that are filled by the *triager* and not by the *reporter*, as the issue progress through its states. These fields are the developer assigned to handle the issue, the current status of the issue report, and the resolution status.

3.1.1 The Life Cycle of Mozilla Bugs

A bug life cycle documents the workflow followed by the bug until its resolution [Bugzilla Team, 2015]. The Mozilla Foundation employs Bugzilla as the official issue tracking system for all its projects, including Firefox. Bugzilla provides a standard bug workflow (as illustrated in Figure 3.1) which is composed of the following states: *Unconfirmed*, *Confirmed*, *In_Progress*, *Resolved*, and *Verified* [Bugzilla Team, 2015]. However, Firefox workflow shows different states than those presented by the standard Bugzilla workflow. We questioned a BMO[1] maintainer about these differences, who answered the following to us:

"This is because default workflow of Bugzilla changed a few years ago, and Mozilla still uses the old one. (...) However, if it's different to the default for earlier Bugzilla, those will be customizations." – BMO's Maintainer (2015-12-29)

In other words, Mozilla systems are using a customized and older version of the Bugzilla workflow. The states found for Mozilla bugs in 2012 are: *Unconfirmed*, *New*, *Assigned*, *Resolved*, *Verified*, and *Reopen*.[2] From these states, three have the same names and representation as the standard Bugzilla (*Unconfirmed*, *Resolved*, and *Verified*), and two states have different names but representing equivalent information

[1]BMO is Mozilla's customized version of Bugzilla (`https://bugzilla.mozilla.org`, verified 2016-04-20).

[2]These states are still in use for BMO during the writing of this thesis.

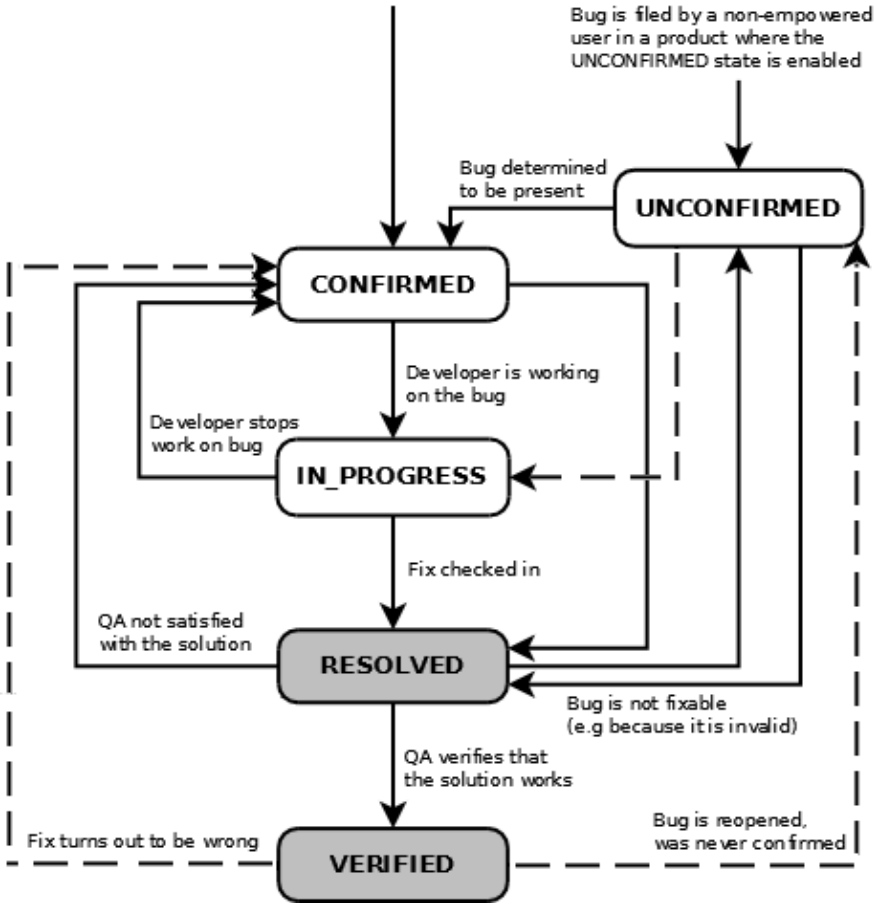


Figure 3.1: Standard Bugzilla Workflow (version 5.x) [Bugzilla Team, 2015].

(*Confirmed* ≡ *New* and *In_Progress* ≡ *Assigned*). Only *Reopen* does not have an equivalent status in the current Bugzilla workflow. Figure 3.2 shows the states for BMO and the valid transitions between them. *{Start}* represents the possible states a new bug report can start in the workflow, i.e., it denotes which states a newly created bug is registered in the tracking system.

Figure 3.3a shows the BMO workflow where the vertices represent the states and the edges represent the transitions between them. This workflow resembles an earlier version of Bugzilla (version 3.x and earlier), which is reproduced in Figure 3.3b. When we compare both workflows, there are indeed many customizations in BMO over the older version. For instance, BMO has more transitions and one less state (*Closed*). We searched Mozilla bugs data and found that BMO used to have this state in the past, however the last bug marked as *Closed* was resolved in 2006. We asked a BMO maintainer why they removed the *Closed* status and his answer was:

"Because it was, in practice, pointless. So while it remains in the config, it is unreachable." – BMO's Maintainer

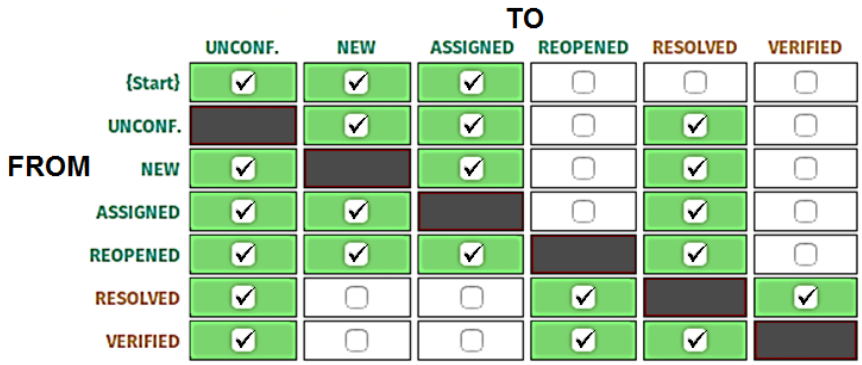
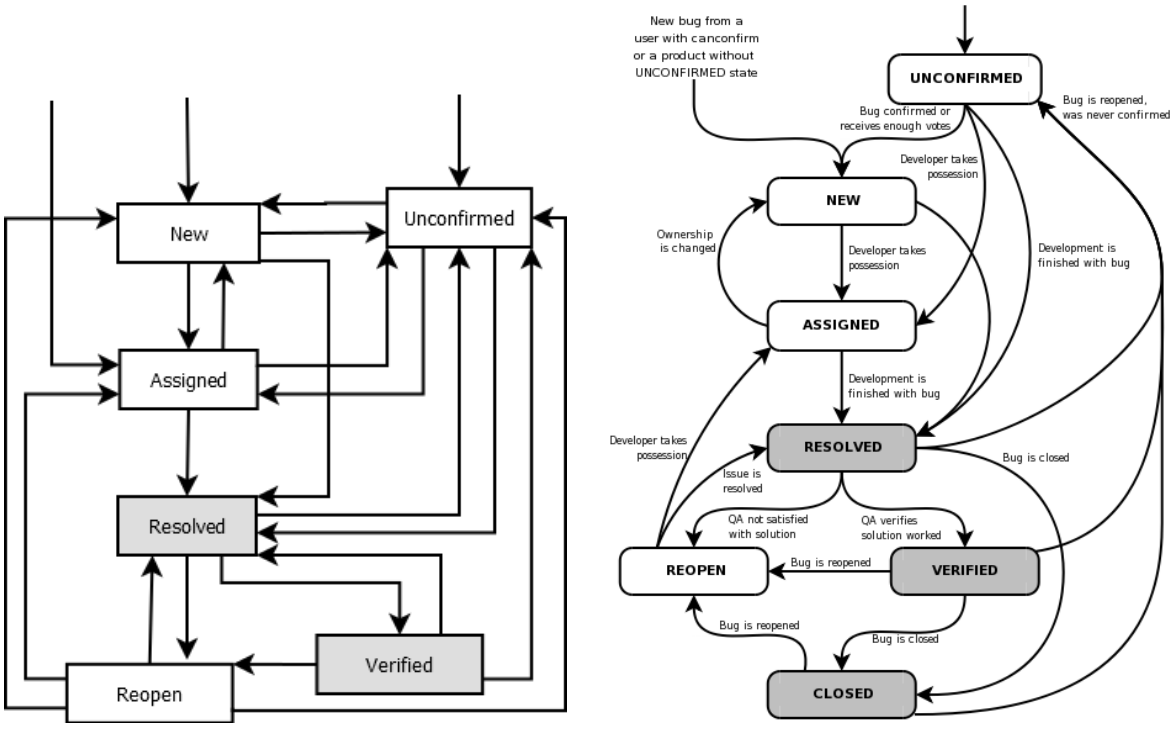


Figure 3.2: Workflow used by Mozilla's customized version of Bugzilla. The cells with a check mark represent valid transitions and white cells indicate there is no transition between these states. Dark gray cells are used when the beginning and end state are the same (i.e., there are no loops in the workflow). Resolved and Verified (colored red) indicate states that a bug is considered closed. This image was provided by a BMO Maintainer in 2015-12-29.



(a) BMO (b) Older Bugzilla

Figure 3.3: Bug Life Cycle Workflows: (a) Mozilla's customized version, (b) Older Bugzilla version (3.x)

Although the states from BMO and the standard workflow are very similar, the transitions between them are not. There are transitions in BMO that would be con-

sidered invalid in the standard workflow. For example, the transition from *New* to *Unconfirmed* is present in BMO (Figures 3.2 and 3.3a), but it does not exist in the standard old Bugzilla workflow (Figure 3.3b). The BMO workflow is specific to the Mozilla ecosystem, and may not reflect other open source projects.

3.1.2 Understanding the Workflow

In this study, we focus on the workflow followed by Firefox bugs under the BMO version. Figure 3.3a shows the possible paths followed by a Firefox bug during its life cycle. When a bug is registered, it is set as *Unconfirmed*, *New*, or *Assigned*. In some systems, common users can only register unconfirmed bugs. On the other hand, super-users may register unconfirmed, new, or assigned bugs. They may also promote unconfirmed bugs to a new or assigned status. The bug status changes to *Assigned* whenever a developer is assigned to work on it. When the developer finishes his work, its status changes to *Resolved*. Unconfirmed and new bugs may also move directly to *Resolved* without passing through *Assigned*. There are two main reasons for this event: (i) the bug is not valid, or (ii) a volunteer posted a correction before someone was officially assigned to work on the bug. Finally, the quality assurance (QA) team verifies the bug resolution. If the bug passes this verification, its status changes to *Verified*.

If a resolved or verified bug proves to be incorrect or flawed, its status changes to *Reopen* to developers to start working on it again. A reopened bug can return to the earlier states of the maintenance process (*New*, *Assigned*) or be concluded as resolved. A reopen, resolved, or verified bug can also return to *Unconfirmed* if the bug is never confirmed on the system. A bug is closed when it is either resolved or verified, otherwise it is considered open. A closed bug has one of the following resolution status: *Fixed*, *Duplicate*, *WontFix*, *WorksForMe*, *Invalid*, and *Incomplete*. A successfully fixed bug is marked as *Fixed*. If there is in the tracking system another report describing the same bug, then the bug is marked as *Duplicate*. A bug that will not be fixed is marked as *WontFix*. When a developer can not reproduce the bug, it is marked as *WorksForMe*. If the bug is not valid, then it is marked as *Invalid*. A vague report description that developers cannot understand or a support request are marked as *Incomplete*.

3.2 Bug Flow Graphs

As proposed in this work, a Bug Flow Graph (BFG) is a directed graph that summarizes a bug life cycle workflow. It provides a visual representation that shows

the bugs flowing through the maintenance process. In a BFG, the nodes represent the status a bug may take throughout the maintenance workflow. The edges represent the transitions between status. Loops indicate bugs that stayed in the same status, i.e., bugs that did not change status and continued in a particular state. Since loops are not represented in the original workflow, we used dotted edges to represent them. An edges' weight is a multi-valued information $P(Md)$, where P is the percentage of bugs that went through the transition and Md is the median time to make the transition. We use the median because it is more robust to skewed distributions, as it is the case of most state transition times. For better visualization of the BFG, we remove loops with no bugs. However, removing other edges with no bugs would hinder the understanding of the maintenance process. For this reason, we colored gray these edges.

Example: Figure 3.4 shows a fragment of a BFG computed for Firefox bugs. The BFG reveals that, during the time frame used to collect the bugs, 90% of the bugs that reached a state called *Assigned* advanced to a *Resolved* state. On the median, three days are needed to perform this transition. It also shows that 6% of the bugs returned to the *New* state, after spending 17 days in *Assigned*, on the median. Finally, 4% of the bugs that reach *Assigned* do not leave this state. All together, their median time in this state is 79 days.

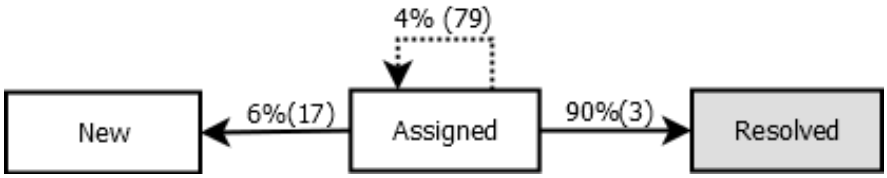


Figure 3.4: Fragment of a BFG for Firefox

The basic steps to create a BFG are: (i) acquire the bugs status changes; (ii) calculate the edges information (percentage and median time); and (iii) draw the workflow.

BFGs are an improvement over directly acquiring workflow data from the ITS. Although the information showed by BFGs are indeed present in tracking systems, it is not easily accessible [Ihara et al., 2009; Luijten et al., 2010]. For example, it is not possible to acquire this data with simple queries, requiring more complex process to extract and to mine historical data. Due to this difficulty, it is reasonable to assume that developers and managers might overlook such information. By contrast, the visualization provided by BFG is an easier way to reason about bug workflows.

Although the BFGs presented in this work are specific to the workflow followed by the Mozilla ecosystem, the methodology can be applied to other projects.

Table 3.1: Mozilla bugs (2012) classified by their resolution status

Status	Bugs		Resolution Time (in days)				
	Number	%	Min	Max	Avg	Dev	Med
Open	13,512	14.08%	–	–	–	–	–
Fixed	49,229	51.28%	0	1,482	76	178	9
Duplicate	10,840	11.29%	0	1,496	94	228	2
WorksForMe	8,028	8.36%	0	1,506	286	346	150
Invalid	6,546	6.82%	0	1,485	129	274	1
WontFix	5,396	5.62%	0	1,511	413	432	241
Incomplete	2,443	2.54%	0	1,488	450	443	290
Total	95,994	100.00%	0	1,511	136	272	7

3.3 Dataset Overview

In this section, we present the dataset—composed of Mozilla bugs—used in the study reported in this chapter. We also provide a general analysis of the bugs and the developers working on them.

3.3.1 Mozilla Bugs

Table 3.1 presents Mozilla bugs in 2012 according to their resolution status. *Open* indicates bugs still not resolved when the data was acquired. The table also shows the resolution time (in days) for each status (minimum, maximum, average, standard deviation, and median). As we can see, 14% of the bugs registered in 2012 are open and still waiting resolution.[3] The most common resolution status is fixed (51.28%), followed by duplicate (11.29%). The less common resolution is *Incomplete* (2.54%).

When we analyze the resolution times in Table 3.1, duplicate and invalid bugs show the lowest resolution time (on the median). Indeed, Bugzilla provides features to aid in the detection of duplicate bugs,[4] consequently, we expect this kind of bugs to be detected more quickly. Invalid bugs are also more easily detectable since they describe intended system's functionalities (i.e., it works like that on purpose) or they are not a Mozilla bug (e.g., a bug in a third-party library). For this reason, the time to resolve invalid bugs is also low. The highest resolution time is recorded for incomplete bugs (average and median). An incomplete bug includes a vague description or it is a support request. The high resolution time to close this kind of bug may indicate that

[3]Those bugs were still waiting resolution at 2016-03-01, when we updated our bug data to the current BMO.

[4]Bugzilla has internal features designed to detect duplicated bugs. For instance, when registering a new bug, Bugzilla automatically warns the user if there is another bug very similar to the one he/she is currently registering.

developers need better ways to identify incomplete bugs. *WorksForMe* and *WontFix* also show high average and median values. This is expected for *WorksForMe* because developers may take a long time trying to reproduce such bugs [Joorabchi et al., 2014]. Since only module owners can resolve bugs with *WontFix* status, it is also expected the longer resolution time because few developers have the permission to apply such status. Fixed bugs show higher median resolution time than Duplicates and Invalids, but they show much lower median values when compared to *Incomplete*, *WorksForMe*, or *WontFix* resolutions. Finally, fixed bugs show the lowest average resolution time.

The average, standard deviation, and median resolution times presented in Table 3.1 suggest that the bugs' resolution time do not fit into a normal distribution. We expected the bugs to follow a skewed distribution based on previous research and experience [Luijten et al., 2010]. To verify our expectation, we plotted the resolution times for each status (and considering all bugs together) as violin charts (Figure 3.5). This chart shows a high density of bugs resolved within a few days. Moreover, the plots suggest that some bugs present very high resolution times. For this reason, measurements like median better represent the resolution data than average and standard deviation.

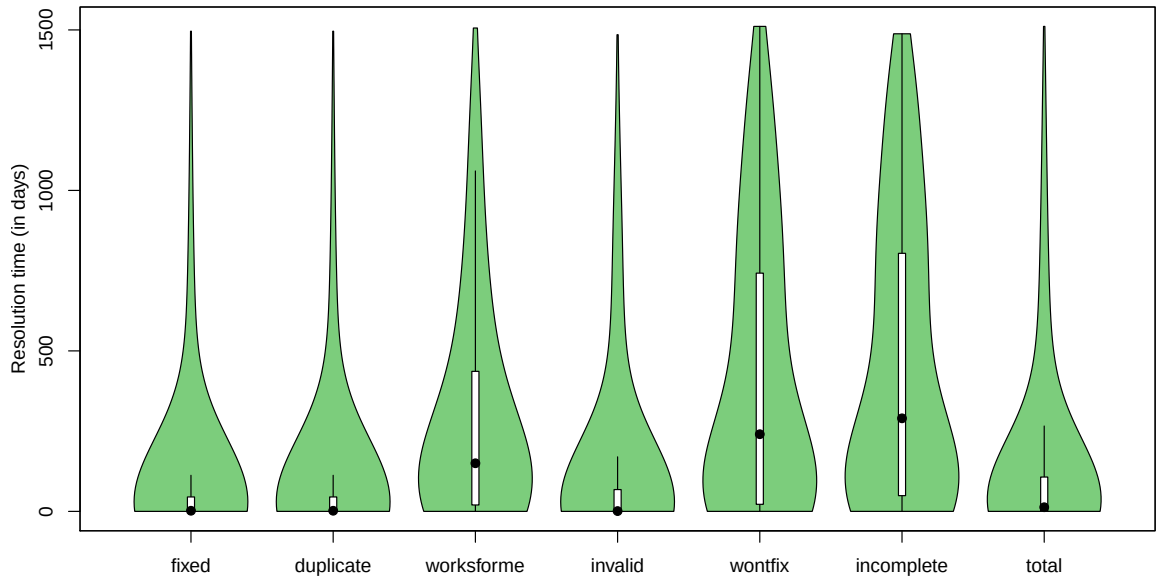


Figure 3.5: Violin plots showing the distribution of resolution times (in days) per bug status.

We also applied the Kruskal-Wallis test to compare the bugs status resolution times. The test showed that the distributions are different (p-value < 0.001). Therefore, the bug nature may impact on the bug resolution time.

Table 3.2: Types of Mozilla Users Registered in Bugzilla

Type	Number	%
Reporter	12,206	65.45%
Commenter	9,341	50.09%
Patcher	1,495	8.02%
Developer	1,911	10.25%
User (total)	18,648	100.00%

3.3.2 Mozilla Users

By analyzing the bugs and their resolution time, we can get a general overview of the bugs. However, there is another key aspect on bug handling, the users interacting with the issue tracking system. We therefore consider the distinction among different types of users according to their interaction with bugs. We classify users into four types: reporter, commenter, patcher, and developer. A reporter is someone who reports a bug (as already described in Section 3.1). A commenter is an user who posts comments on bugs. A patcher is an user who posts a patch file for a bug. And a developer is someone who was assigned to handle a bug. For us, an user is someone who interacts in any way with a bug, i.e., users are the union of the four types. However, these types are not mutually exclusive, and it is possible for an user to belong to more than one user type.

As presented in Table 3.2, 18,648 users interacted with Mozilla bugs in 2012. As we expected, the number of reporters and commenters is greater than those of patchers and developers. Patchers and developers are related groups and share many common users, almost 80% of patchers also fall into the developer category. For this study, we focus on the developers because they are the users officially assigned to fix bugs.

3.3.3 Developers Profile

We classify developers into specific categories according to their bug handling skills. For this study, we excluded the developer identified as "nobody"[5] because such user is not a person. We consider the number of bugs assigned to the developer for skill based classification. In our dataset, developers are assigned 23 bugs on average, with 60 bugs as standard deviation, and three bugs on the median. The average, deviation, and median suggest that bugs assigned to developers follow a skewed distribution. For this reason we divided the developers into quartiles presented in Table 3.3. We call

[5]"nobody" is used in Bugzilla as a placeholder, until the bug is properly assigned to a real developer. However, a bug can be resolved and its assigned field not updated to the responsible user, and as such "nobody" takes credit for the resolution.

Table 3.3: Developers Classification by Skill

Quartile	Assigned Bugs	Skill Level
1st	1	Newbie
2nd	3	
3rd	22	Junior
4th	>22	Senior

Newbies the developers from the first and second quartiles, i.e., developers who are assigned to work on three bugs or less. The third quartile are *Junior* developers who work on 22 bugs or less. *Senior* developers hold the fourth quartile and they work on more than 22 bugs.

We compute the resolution time (in days) for every bug handled by developers in each skill category. We also add a new category called *Expert*, which are the top-10 developers in number of worked bugs (*Expert* is a subset of *Senior*). Figure 3.6 shows the resolution times as violin charts. The charts suggest the bug resolution time for each developer skill category follows a skewed distribution. We can also observe a trend on developer skill versus resolution time, i.e., bugs have a faster resolution time as the developer is more skilled. For instance, *Newbies* showed an average, standard deviation, and median of 282, 344, and 188 days, respectively. On the other hand, *Senior* developers showed 62, 155, and 8 days, respectively for the same measures.

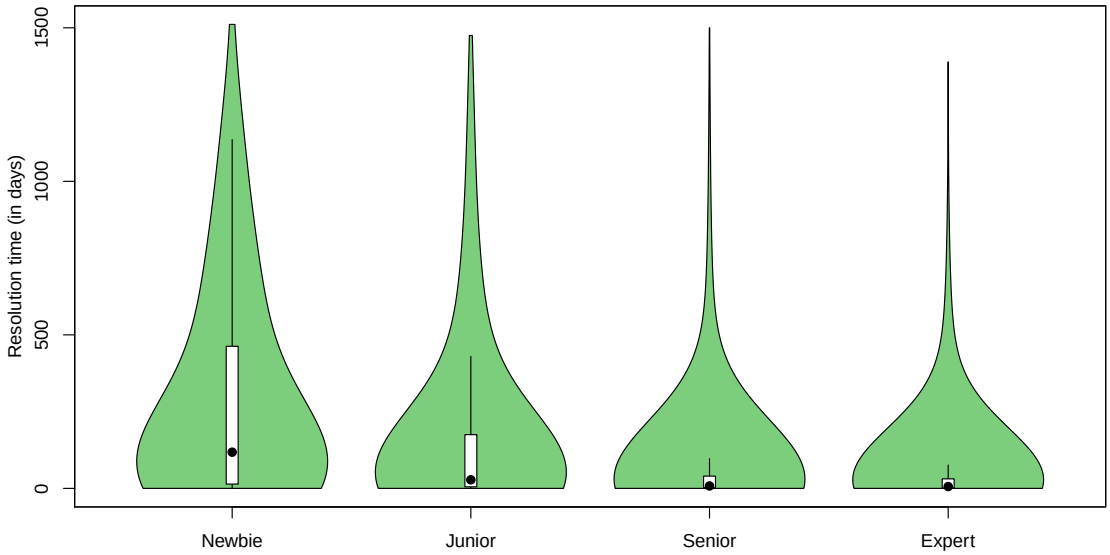


Figure 3.6: Distribution of resolution times (in days) for each developer skill category.

We also applied the Kruskal-Wallis test to compare resolution times according to developers category. We found the distributions are different (p-value < 0.001). Therefore, the developers skill indeed impact on the bug resolution time.

3.4 Study Results

In this section we present and analyze the results of our study by creating and analysing BFGs. Our motivation is to perform an exploratory research and to learn lessons from the results we obtain.

3.4.1 Overall Workflow Analysis

The dataset discussion in Section 3.3 shows a general overview of bugs registered for the Mozilla ecosystem. However, we must analyze the workflow followed by bugs to perform a more thoroughly characterization. Figure 3.7 shows the BFG for all Mozilla bugs considered in this study.

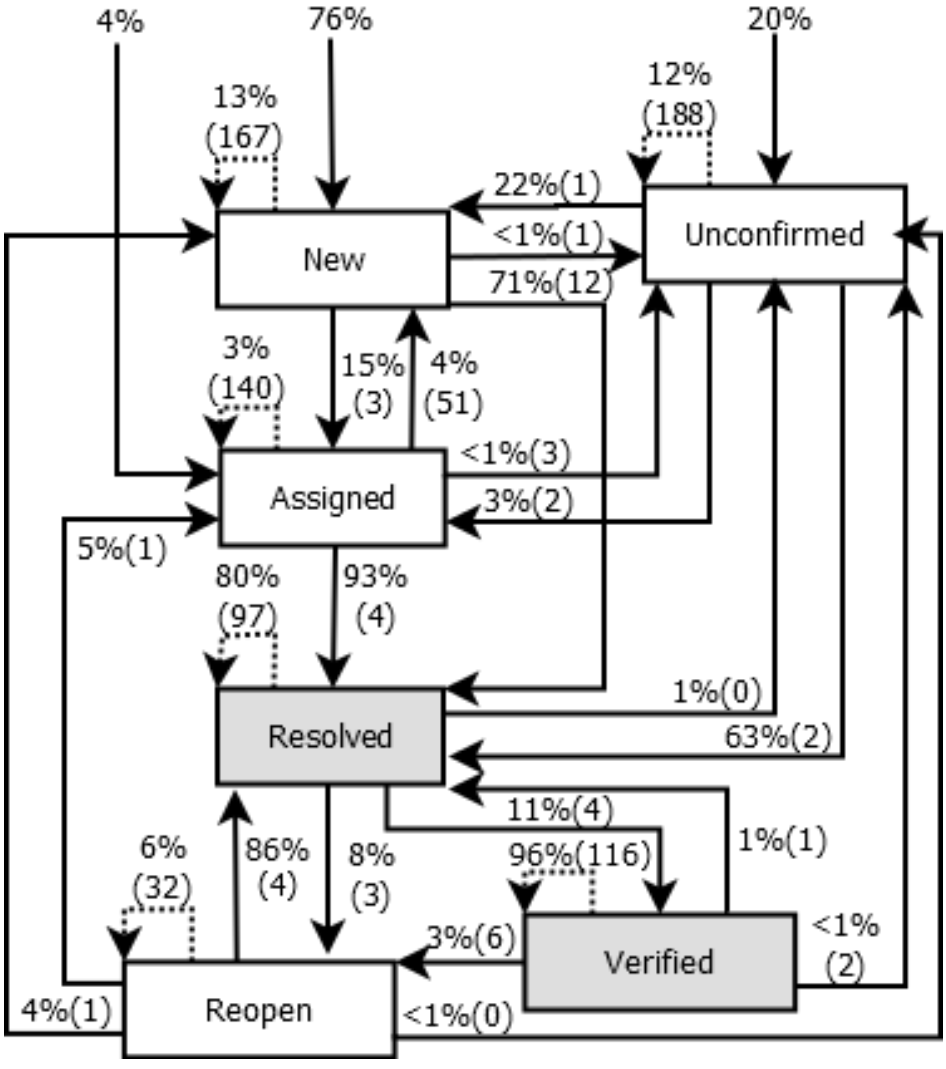


Figure 3.7: BFG for all Mozilla bugs.

By analyzing this BFG, we can make at least the following observations about the workflow followed when resolving these bugs:

- Most bugs start as *New* ($\approx$ 76%). However, there is a large portion of bugs starting the workflow as *Unconfirmed* ($\approx$ 20%). As expected, it is a rare condition for a bug to start as *Assigned* ($\approx$ 4%).

- For the unconfirmed bugs, few of them stay in the unconfirmed status ($\approx$ 12%) and few are confirmed as a new bug within one day (transition *Unconfirmed* to *New*, $\approx$ 22%). Almost two thirds of unconfirmed bugs are resolved in six days (transition *Unconfirmed* to *Resolved*, $\approx$ 63%).

- A small proportion of new bugs wait three days (on the median) until they are officially assigned to a developer (transition *New* to *Assigned*, $\approx$ 15%). Furthermore, many new bugs are directly resolved without being officially designated to a developer after waiting 12 days (transition *New* to *Resolved*, $\approx$ 71%). There is also bugs that remain open in the *New* status ($\approx$ 13%). It is very rare for a new bug to return to the *Unconfirmed* status (less than 1%).

- An interesting finding is that new bugs are resolved faster when they are properly assigned to a developer. It usually takes seven days (three days from *New* to *Assigned* plus four days from *Assigned* to *Resolved*) for a new bug to be resolved if it is assigned to a developer. By contrast, it takes 12 days for a new bug to be closed without being formally designated to a developer (transition *New* to *Resolved*).

- Most resolved bugs stay closed, either in the *Resolved* ($\approx$ 80%) or *Verified* ($\approx$ 96%) states. Few bugs are verified by the quality control team, which usually takes four days (transition *Resolved* to *Verified*, $\approx$ 11%). Only a portion of bugs are reopened (transition *Resolved* to *Reopen*, $\approx$ 8%) and even fewer return to the beginning of the workflow as unconfirmed bugs (transition *Resolved* to *Unconfirmed*, $\approx$ 1%).

Positive Points: Most resolved (80%) and verified (96%) bugs remain closed (i.e., they are not reopened), which appoints to the efficacy of Mozilla developers in their bug resolutions.

Negative Points: Many open bugs (12% *Unconfirmed* and 13% *New*) remain in the same state. The time these bugs stay in the initial states of the workflow is very

high (*Unconfirmed* 188 days; and *New* 167 days), which could indicate they are being ignored or neglected.

Opportunities for Improvements: We found that bugs are resolved five days faster when properly assigned to a developer. Therefore, tools to assist developers and volunteers on finding bugs of their interest could contribute to reduce the resolution time of Firefox bugs. As examples, we can mention tools to allocate developers to bugs [Anvik et al., 2006] or to recommend similar bugs that can be fixed after a given bug, as the recommender tool proposed in Chapter 4. Another effort can be directed on helping developers finding more open bugs, especially those being neglected for a long time in the *New* and *Unconfirmed* states.

Although the BFG presented in Figures 3.7 provides a starting point to understand the Mozilla workflow, grouping all types of bugs may hide peculiarities of the maintenance process. For example, this BFG does not discriminate fixed bugs from other types. Since the recommendation system described in Chapter 4 is designed to help developers on fixing more bugs, we decided to analyze BFGs focusing only on fixed bugs.

3.4.2 Fixed Bugs Workflow

Figure 3.8 shows a BFG that considers only fixed Mozilla bugs. By analyzing this BFG, we can make the following observations:

- Most fixed bugs bugs start as new ($\approx$ 87%), while unconfirmed ($\approx$ 6%) and assigned ($\approx$ 7%) bugs are less frequent. This contrasts with the BFG for all bugs where new bugs show a lower percentage ($\approx$ 76%) as the starting state of bugs.

- The BFG also shows that $\approx$ 24% of the new bugs are formally assigned to a developer within two days (transition *New* to *Assigned*). The remaining 76% are resolved directly after seven days (transition *New* to *Resolved*).

- The BFG also confirms the findings we discovered analyzing the BFG of all bugs, which is that bugs not formally assigned to a developer take a longer time to be resolved.

- Only 17% of resolved bugs are verified by the quality control team within four days after they are fixed (transition *Resolved* to *Verified*).

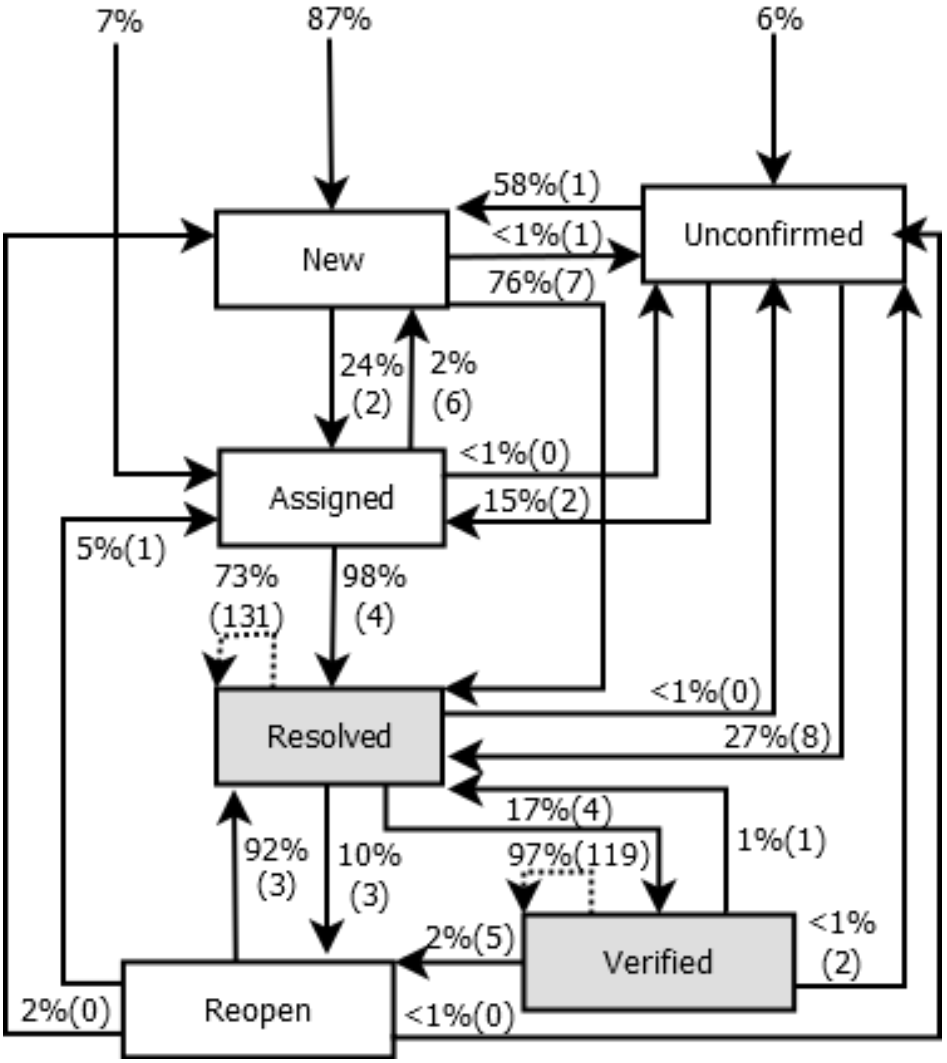


Figure 3.8: BFG for Fixed Mozilla bugs.

Positive Points: Unconfirmed bugs are quickly confirmed within one day as they arrive in the workflow (*Unconfirmed* to *New*). This is important because new bugs follow faster paths towards resolution than unconfirmed ones. Moreover, there is an increase in the percentage of assigned bugs, which takes less time to fix.

Negative Points: Although the percentage of verified bugs is higher than in the workflow for all bugs (17% for fixed only versus 11% for all bugs), it is still lower than expected for a mature process. Another negative aspect is the time spent for unconfirmed bugs to be directly resolved, which is much greater than the one showed for all bugs (eight days for fixed, and two days for all bugs).

Table 3.4: Top-10 Mozilla products considering fixed bugs registered in BMO

Product Name	Bugs	%
Core	12,594	13.12%
Infrastructure & Operations	3,106	3.24%
Release Engineering	3,045	3.17%
Firefox OS	2,929	3.05%
Firefox	2,688	2.80%
Firefox for Android	2,200	2.29%
www.mozilla.org	1,722	1.79%
Marketplace	1,577	1.64%
Testing	1,459	1.52%
addons.mozilla.org Graveyard	1,235	1.29%

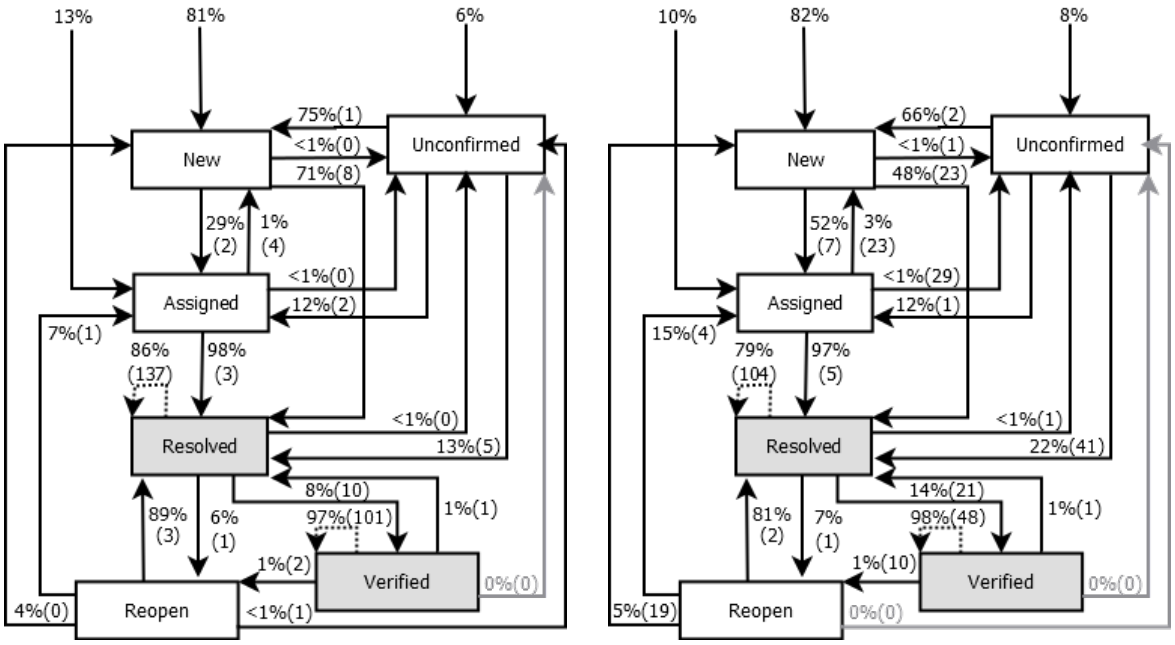
Opportunities for Improvements: The verification of bugs can be a bottleneck in the workflow because only a few portion of resolved bugs are verified by the quality control team. Even though only 10% of resolved bugs are reopened, this transition indicates rework performed by developers. We also found that confirmed (or *New*) bugs are fixed faster when they are properly assigned to a developer. Therefore, this finding reinforces that techniques to assist developers on finding appropriate bugs can contribute to optimize the maintenance process.

3.4.3 Workflow of Top Systems in Number of Fixed Bugs

Table 3.4 shows the top-10 Mozilla products according to the number of fixed bugs in BMO. The top-3 are internal products to Mozilla, i.e., they are not systems available to end-users. *Core* is composed of shared components used by Mozilla systems. *Infrastructure & Operations* registers bugs related to server and network infrastructure. *Release Engineering* catalogs bugs related to Mozilla releases (e.g., branded releases, release automation, and building releases). The next three products are related to the Firefox browser. *Firefox OS* is an open source operating system designed for mobile devices. *Firefox* is the second most popular web browser in the world.[6] *Firefox for Android* is Firefox browser designed for Android devices. The next systems in number of fixed bugs are *www.mozilla.org* (Mozilla.org website), *Marketplace* (marketplace to distribute applications), *Testing* (automated testing of Mozilla clients), *addons.mozilla.org Graveyard* (retired components from addons.mozilla.org).

Figure 3.9 shows a BFG that considers only fixed bugs for the Core and Firefox systems. By analyzing this BFG, we can make at least the following observations:

[6]According to W3Schools (`http://www.w3schools.com/browsers/browsers_stats.asp`, verified 2016-05-22).



(a) Core (b) Firefox

Figure 3.9: BFGs for fixed bugs on Mozilla systems: (a) *Core*, (b) *Firefox*.

- In the *Core* system, a directly resolved bug (transition *New* to *Resolved*) takes one day longer when compared to all fixed bugs (Figure 3.8). On the other hand, the resolution time when a bug is assigned to a developer is one day shorter when compared to all fixed bugs.

- In *Firefox*, a directly resolved bug takes 16 days longer than in the BFG for all fixed bugs. On the other hand, the resolution time when a bug is assigned to a developer is six days longer (five days longer to be assigned to a developer, and one day longer to be resolved).

- For both systems, it takes a longer time to verify resolved bugs (transition *Resolved* to *Verified*). The BFG for all bugs shows a median of four days to verify a bug, while in the *Core* system this time is ten days and in Firefox it is 21 days.

- For both systems, it takes only one day for a resolution to be found incorrect and to reopen the bug (transition *Resolved* to *Reopen*). By contrast, when all fixed bugs are considered it takes four days on median for a fixed bug to be found incorrect and be reopened.

- For *Firefox*, unconfirmed bugs take 41 days (on the median) to be resolved directly. This contrasts with the median considering all fixed bugs which is only eight days.

Positive Points: In both systems, bugs assigned to developers are resolved faster than the ones directly resolved. Another positive point is that incorrect resolutions are discovered faster in both systems, as it usually takes only one day for such bugs to be reopened.

Negative Points: Both systems show a increased time to verify bugs and to resolve new bugs directly (one more day for *Core*, and 16 more days for *Firefox*). Specially transition times in *Firefox* are usually higher when compared to all fixed bugs considered together.

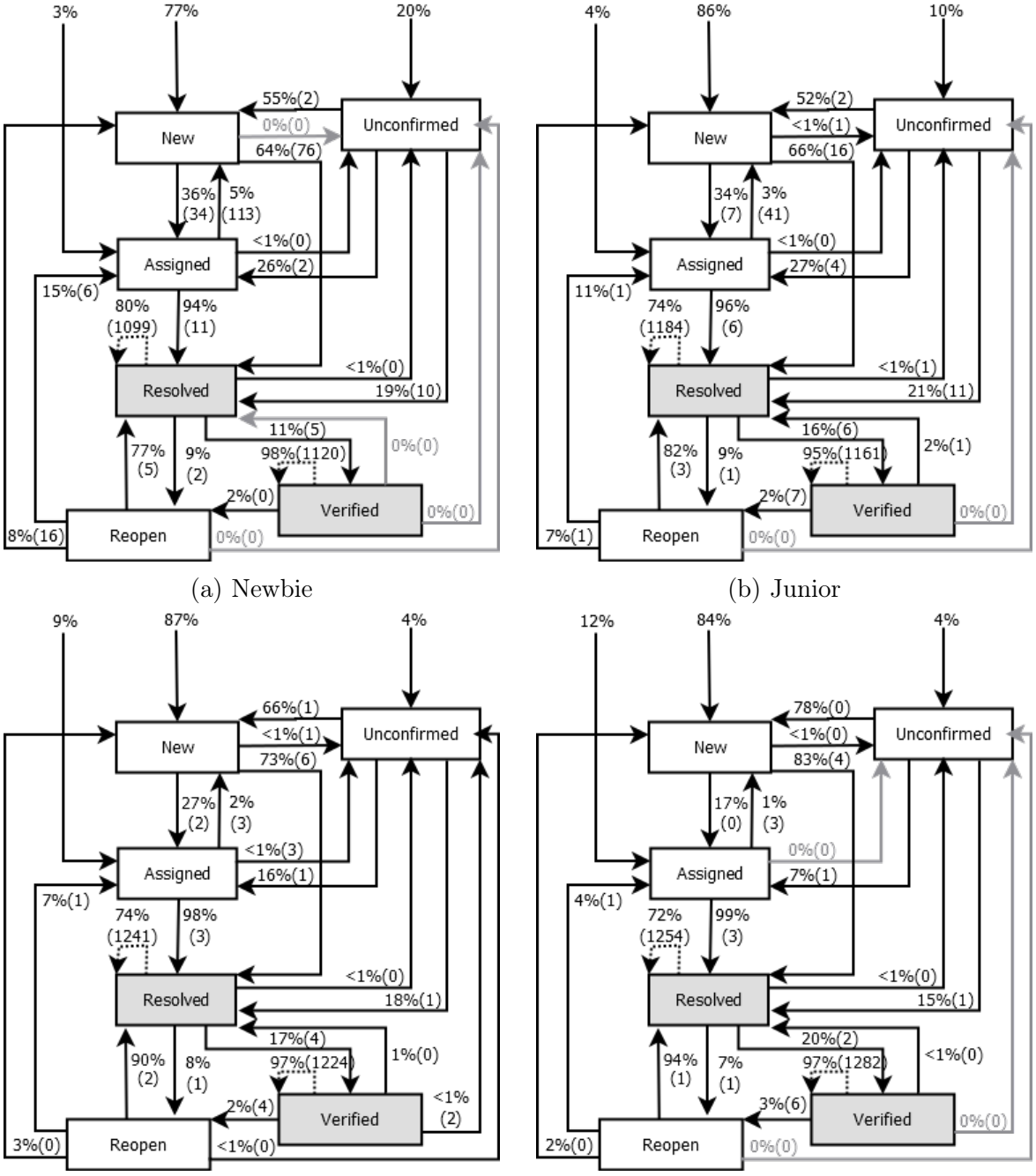
Opportunities for Improvements: The verification of bugs may be a bottleneck in the workflow, mainly because of the time. The time required for a *Firefox* bug to be verified is approximately four times greater than the time a developer working on the bug requires to fix it, i.e., five days to fix (transition *Assigned* to *Resolved*) and 21 days to verify (transition *Resolved* to *Verified*). For *Core* system, the time to verify is approximately three times greater than the time for a developer to fix (i.e., three days to fix versus ten days to verify). We could infer that the quality control team for *Core* and *Firefox* may need more people or tools to improve this verification.

3.4.4 Developers Workflow

In this section, we investigate how developers handle bugs by analyzing their workflow. We create BFGs considering only fixed bugs for each developer skill level (Figure 3.10). As we can see, the workflows for each skill category are different. We can make the following observations when comparing these BFGs:

- First, we begin by analyzing the assignment of bugs. As previously discussed (Section 3.4.1), a bug takes less time to be resolved when it is assigned to a developer. The BFGs show that skilled developers have a lower percentage of bugs being assigned to them (transition *New* to *Assigned*). Moreover, the time in days for a new bug to be assigned to a developer decreases as his skill level increases, i.e., less days are required to assign a bug to skilled developers. We also see an increase on the number of bugs that start the workflow as already assigned in accordance to the developer skill level (3% for *Newbies*, 4% for *Juniors*, 9% for *Seniors*, and 12% for *Experts*).

- Analyzing the resolution of assigned bugs (transition *Assigned* to *Resolved*), the percentage of bugs following the transition are very similar (over 94%). *Newbies*



(a) Newbie (b) Junior

(c) Senior (d) Expert

Figure 3.10: BFGs according to Developers' Skill Profiles: (a) *Newbie*, (b) *Junior*, (c) *Senior*, and (d) *Expert*.

showed a greater resolution time than the others (11 days). *Junior* require six days to fix a bug. *Seniors* and *Experts* usually take three days to fix an assigned bug. Moreover, if we sum up the assignement time (*New* to *Assigned*) and the fixing time (*Assigned* to *Resolved*) the results are 45 days for *Newbies*, 13 days

for *Juniors*, five days for *Seniors*, and three days for *Experts*.

- An interesting finding is that for *Newbies* and *Juniors* bugs spend more time waiting to be assigned than being handled by developers. For *Newbies*, a bug waits 34 days in the *New* status until being assigned to a developer, but only 11 days to get fixed after its assignment. *Juniors* wait seven days to be assigned to a bug, while taking five days to fix it. Therefore, tools to help assign bugs to developers are more helpful to less skilled developers.

- When we analyze bugs that are fixed directly following the transition from *New* to *Resolved*, developers skill seem to affect the percentage of bugs and resolution times. The percentage shows a clear trend, since the transition *New* to *Assigned* is more common than transition *New* to *Resolved*, in the case of skilled developers

- The more skilled the developer, the less likely is he/she to work on unconfirmed bugs. We can see a decrease in the number of bugs that start the workflow as unconfirmed, as the developer becomes more experienced. Moreover, the percentage of unconfirmed bugs that are directly resolved (transition *Unconfirmed* to *Resolved*) also decreases as the developer is more skilled.

- There is also a relation between the developers experience and the percentage of bugs verified by the quality control team. The more skilled the developer, the more bugs are verified.

Positive Points: Developers' skill has a positive impact on the fixing time of bugs, i.e., bugs require less time to be resolved if the developer handling them is more experienced, as expected.

Negative Points: Even though it takes more time to directly resolve bugs, skilled developers prefer this path over being properly assigned to bugs (which in the workflow requires less time to fix).

Opportunities for Improvements: When *Newbies* and *Juniors* are involved, a bug spends more time waiting to be assigned than actually being fixed. Therefore, tools to assign bugs to developers probably provide more value to less skilled developers, and could reduce the time that *Newbies* and *Juniors* demand to fix bugs.

3.5 Threats to Validity

Construct Validity: The transition times in our results are extracted from the bug tracking system. We cannot guarantee that this is the time the developer is actually working on the bug. Certainly, developers overlap bug fixing with other activities, specially in the case of volunteers.

Conclusion Validity: The findings we discussed are based on the median number of days presented by BFG transitions. Since it is a median, we can only be certain that at least half of the transition follow the indicated number of days. However, it is common for an overall analysis to account for aggregate statistics to draw lessons.

Internal Validity: We investigated Mozilla bugs reported for one year (2012). For this reason, our findings may change if different time intervals are investigated. Another threat is that we extracted all the information directly from the bug tracking system database. Therefore, if the tracking system stored faulty information that would impact in our findings.

External Validity: In our characterization study, we analyzed bugs reported for the Mozilla ecosystem. Therefore, our findings may not be valid to other other open source or closed source systems.

3.6 Final Remarks

In this chapter, we conducted a characterization study focusing on the workflow followed by Mozilla developers to handle bug reports. We begin by describing the Mozilla's maintenance process (Section 3.1). In Section 3.2, we propose the concept of Bug Flow Graphs to provide a visual representation of the overall workflow process. BFGs can be used by team leaders to identify critical points in the maintenance workflow. In Section 3.3, we present the dataset used for the characterization. We also performed an overview analysis on the dataset showing resolution times for the bugs, and the users interacting with the bugs. We also analyze the developers by classifying them into categories according to their skill. In Section 3.4, we present the characterization results by showing BFGs and discussing their positive and negative points, as well as possibilities for improvement. We also present BFGs for developers skill categories, which reveals interesting differences among them. Finally, in Section 3.5, we discuss threat to validity.

Chapter 4

Recommendations of Similar Bugs

In this chapter, we present an approach to recommend similar bugs to developers. Our goal is to extend Issue Tracking Systems (ITS) with recommendations of bugs that developers should consider to work on. Considering that developers can improve their productivity by avoiding context changes during software maintenance activities [Kersten and Murphy, 2006], we propose a recommendation system to suggest additional—and similar—software maintenance requests when a developer selects one issue to work on. The proposed recommendations consider the textual similarity of the suggested issues with the main one to foster the creation of sequences of software maintenance requests that can be implemented by the same developer. Our approach is particularly indicated for large scale open-source software systems, which have a decentralized and global community of developers who post daily many software maintenance requests in an issue tracking system [Liu et al., 2012; Mockus et al., 2002].

First, we begin by describing the main characteristics as well as the major decisions behind the design of the proposed approach for recommending similar bugs (Section 4.1). Then, we present a tool that implements this approach, called NextBug, which enhances Bugzilla standard interface with recommendations of similar bugs (Sections 4.2). Finally, we end this chapter with final remarks on the proposed approach (Section 4.3).

4.1 Proposed Approach

We assume that bugs are stored in a tracking system that has at least the following fields: short description (or summary), bug completion status (pending or completed), and the component (systems are usually divided into smaller logical components in the ITS). These fields are commonly present in existing ITS. Pending bugs are those

waiting for resolution. Completed bugs are those marked with a closed status in the ITS. In the proposed approach, we need the binary status of bug completion, i.e., if the bug is considered pending or closed. We also assume that the projects tracked by the ITS is structured into components and that bugs are allocated to particular components, during a triage process [Anvik et al., 2006].

Figure 4.1 illustrates the algorithm proposed to recommend similar bugs, which is inspired on algorithms previously proposed to detect duplicated bug reports [Wang et al., 2008]. We rely on standard information retrieval techniques for preprocessing a *query* q (representing the short description of a bug browsed by the developer) and a collection of *documents* B (representing the short descriptions of the pending bugs in the tracking system with the same component as q). Because B only includes bugs that refer to the same component of q, the recommendations share the same component with the bug queried by the developer.

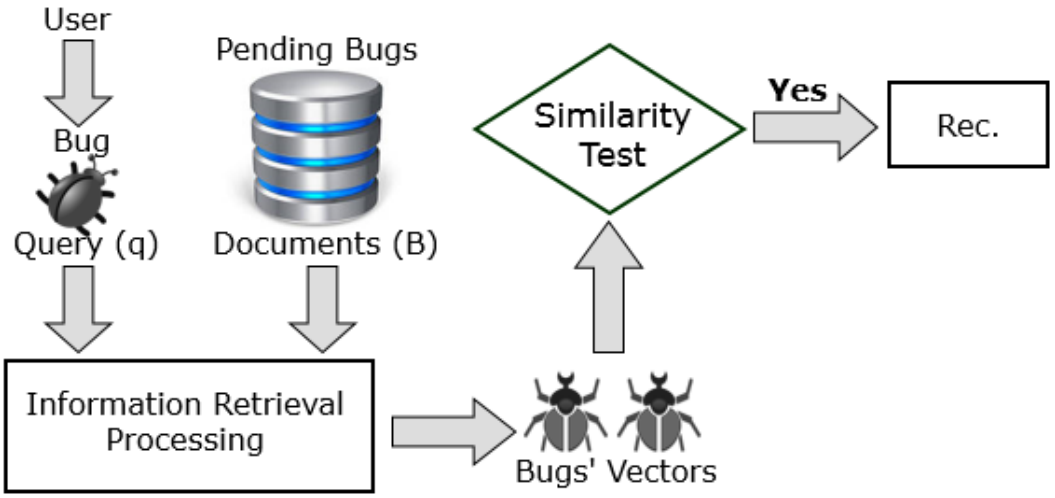


Figure 4.1: Proposed approach for recommending similar bugs

The preprocessing steps performed over q and B include (i) tokeninzation, to separate the text into a bag of words (ii) the use of stemming techniques to reduce the words to their radical form and (iii) the removal of stop-words [Wang et al., 2008; Runeson et al., 2007; Baeza-Yates and Ribeiro-Neto, 1999]. After this initial step, we calculate the cosine similarity between q and each $b \in B$. We verify whether the measured similarity passes a threshold τ to recommend b as a bug similar to q. This threshold is the only free parameter that needs to be set up prior to use, and the proposed approach does not require any training or tuning phase.

4.1.1 Rationale

In this section, we discuss the main design decision behind our approach.

Why do we use the short description (or summary) instead of full descriptions? Ko et al. [2006] and Wang et al. [2008] analyzed the use of both short and full descriptions in the detection of duplicated issues. Ko et al. [2006] argued that by using only summaries we can improve the precision and the efficiency in finding similar bugs, when compared to the use of extended bug descriptions. On the other hand, Wang et al. [2008] compared the usage of short and extended descriptions and concluded that using both descriptions is slightly better than using just the summary. Moreover, we designed our approach to compute over a large number of bugs. Therefore, using only the summary is faster than its extended description.

Why we do not use stack traces? Among the studies described in Sections 2.4, 2.5, and 2.6, only Wang et al. [2008] employ execution trace information when dealing with issue reports. Their original issue dataset did not contain execution trace information, and the authors had to extract this information. Moreover, Nguyen et al. [2012] pointed out that execution traces are not an information usually stored in issue tracking systems. Finally, as execution traces are not present in the dataset we used to evaluate our approach, we decided to not use this kind of data in our approach.

Why we do not use a classifier? There are studies applied to bugs that employ classifiers as their main component [Anvik et al., 2006; Wang et al., 2008; Jalbert and Weimer, 2008; Tian et al., 2012]. As discussed by Tian et al. [2012], bug duplication approaches can be modeled either as a classification or retrieval problem. Classification techniques have the disadvantage to require training, which hinders the usability of the technique on real world applications. On the other hand, retrieval techniques are usually simpler, although they require users to manually select the duplicates. We claim that for a recommendation system, the retrieval route is more appropriate than a classifier because the users will manually select the recommendations provided by the approach.

Why we do not use a clustering algorithm? Among the studies described in Sections 2.4, 2.5, and 2.6, only Jalbert and Weimer [2008] use a clustering algorithm to create a feature to help its classifier. Therefore, clustering is not used directly for the detection but as a component to a more complex classifier. We also tried to employ

a few clustering algorithms in the begging stages of our research to identify similar bugs. However, our preliminary results were not satisfactory when we used clusters. Therefore, we decided to not to pursue this line of work in this thesis.

Why we do not use the LSA/LSI algorithm? Some studies described in Chapters 2 employ the Latent Semantic Analysis/Index (LSA/LSI). However, the LSA algorithm is known to have high memory and performance constraints. In fact, we tried to employ LSA at the beginning stages of our research, but our hardware was unable to run LSA for our complete dataset.

Why we do not map bugs to files and use that information for recommendations? In this research, our goal is to predict similar bugs relying only on the information available at bug reports. To map files to bugs would require to mine historical commit data, as employed by Anvik et al. [2006]. This solution is outside the scope set up for this thesis, and is left as a future work idea.

Do duplicate bugs affect our approach? Our approach is designed to be used when a developer is searching for a bug to work on. In this stage of the maintenance process, duplicated bugs should be already filtered in the ITS (otherwise it is a triage mistake). Moreover, Bugzilla and other ITSs have integrated features for duplicate bug detection. Therefore, we can assume there will be few duplicate bugs going to developers, which would not have a significant impact on our results.

4.2 NextBug: A Prototype Implementation

In this section, we present NextBug [Rocha et al., 2014, 2015], an open-source tool available under the Mozilla Public License (MPL). We describe the tool's main features (Section 4.2.1) and the tool's architecture and its main components (Section 4.2.2).

4.2.1 Main Features

Currently, there are several ITSs that are used to handle software maintenance requests such as Bugzilla, Jira, Mantis, and RedMine. NextBug is implemented as a Bugzilla plug-in because this ITS is used by the Mozilla project, which is used to evaluate our approach (see Chapters 5 and 6). However, NextBug can also be extended and applied to other ITS.

When a developer is analyzing or browsing an issue, NextBug can recommend similar bugs using the standard Bugzilla web interface. NextBug uses a textual similarity algorithm to verify the similarity among bug reports registered in Bugzilla, as described in Section 4.2.2.

Figure 4.2 shows an usage example of our prototype implementation. This figure shows a real bug from the Mozilla project, which refers to a FirefoxOS application issue related to a mobile device camera (Bug 937928). As we can observe, Bugzilla shows detailed information about this bug, such as a summary description, creation date, product, component, operational system, and hardware information. NextBug extends this original interface by showing a list of bugs similar to the browsed one. In Figure 4.2, this list is shown on the lower right corner. Another important feature is that NextBug is only executed if its Ajax link is clicked and, thus, it does not cause additional overhead or hinder performance to developers who do not want to follow similar bug recommendations.

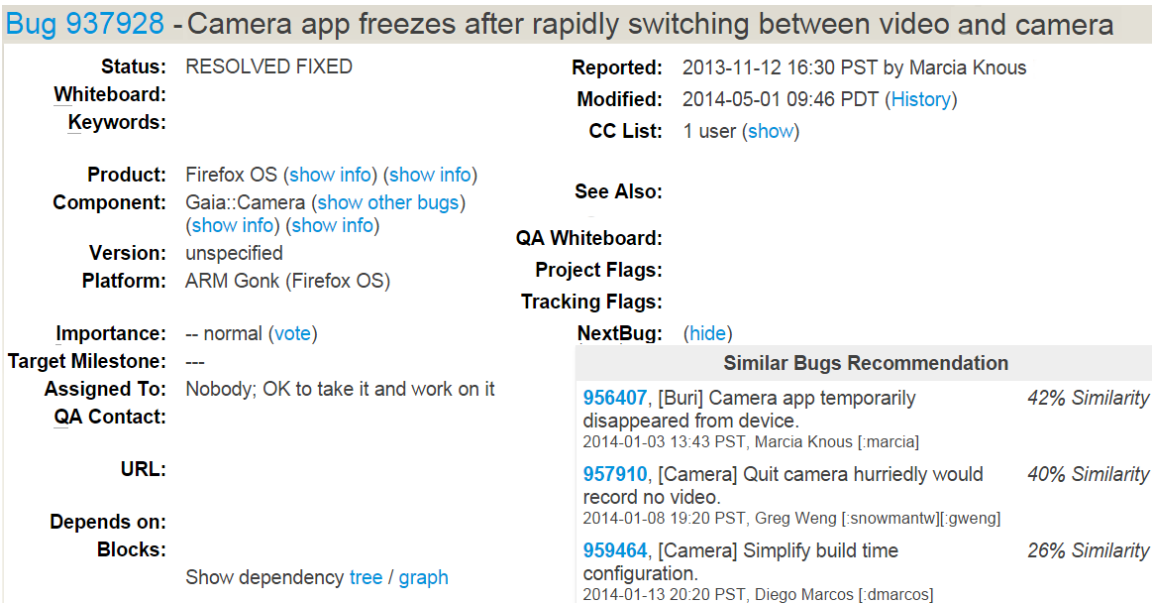


Figure 4.2: Screenshot showing the original Bugzilla interface for browsing a bug enhanced with the NextBug plug-in. Similar bugs are shown in the lower right corner.

In Figure 4.2, NextBug suggested three bugs similar to the presented one.

- *Bug 956407 - [Buri] Camera app temporarily disappeared from device, with a similarity of 42%.*
- *Bug 957910 - [Camera] Quit camera hurriedly would record no video, with a similarity of 40%.*

- *Bug 959464 - [Camera] Simplify build time configuration, with a similarity of 26%.*

As we can note, NextBug not only detects similar bugs but it also reports an index to express this similarity.

Another important feature of NextBug is the support of filters to configure the provided recommendations according to the user's preferences. We implemented this feature after conducting a survey with Mozilla developers (described in Chapter 6). Many developers requested a feature to customize the search results. The following comments show a sample feedback received from three Mozilla developers:[1]

"It would not be bad, as long as it could be configured to do the recommendations according to a set of parameters." – Subject # 13.

"If it presented similar bugs that I did not file, and which were not assigned to anyone, that might be useful." – Subject # 63

"It would probably need to be able to at least check if a bug is already assigned to the current user (or to another user, which probably makes them ineligible, too)." – Subject # 14

The recommendations produced by NextBug follow the established criteria set up by the filters. By clicking on the gear icon next to the NextBug label, a dialog popup shows the configuration options for the filters (Figure 4.3). These filters include the following options: (a) similarity threshold, i.e, the minimum similarity measure required for a recommendation to be considered valid, (b) maximum number of recommendations given by NextBug, (c) maximum and minimum bug severities considered valid for the recommendations, (d) an option to consider enhancements requests along with bugs when presenting the recommendations for similar bugs, (e) searching only for unassigned bugs, i.e., recommend only bugs that no one is currently working on, and (f) shows only bugs that are not reported by the current user; this filter is useful because the current user is probably aware of the bugs he/she reported. Except for the first and second filters, all other filters are optional, e.g., the user is not required to check for unassigned bugs if he/she does not want to.

The similarity threshold (i.e., the first filter) requires some experimentation from the user to fit his preferences.[2] We suggest an initial value of 10%, which usually

[1]The developers described here are the same subjects we presented in the field study (Chapter 6).

[2]Since our approach is unsupervised, the threshold is left for the user to configure according to its needs.

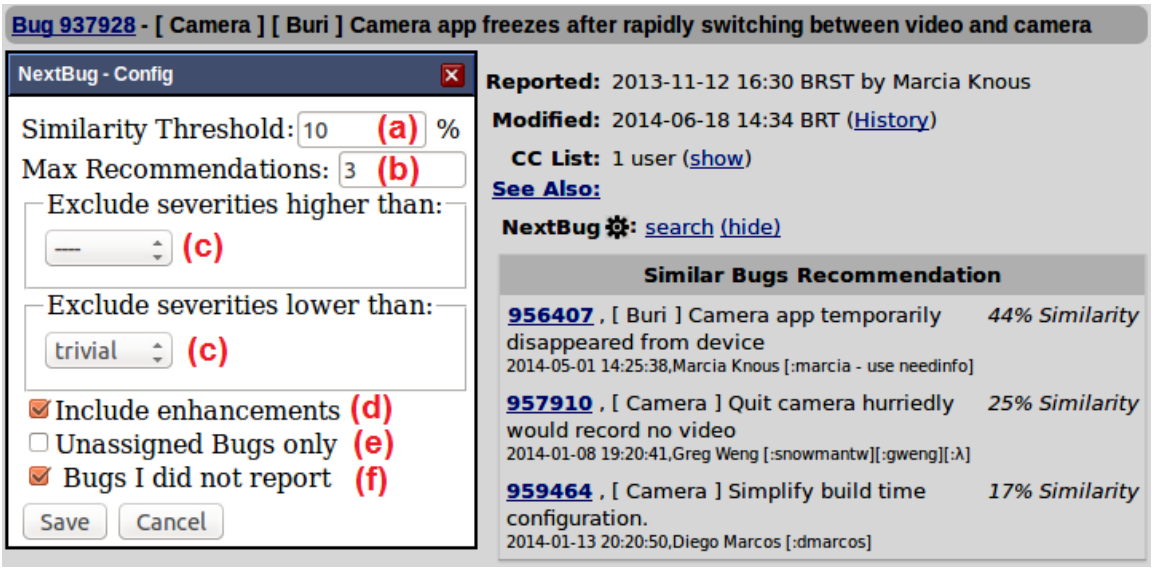


Figure 4.3: NextBug dialog with the configuration options for filtering recommendations.

results in a good number of relevant recommendations (as in the evaluation reported in Chapter 5). As the user increases this value, NextBug gives less recommendations, but more precise ones.

4.2.2 Architecture and Algorithms

Figure 4.4 shows NextBug's architecture, including the system main components and the dependencies between them. As described in Section 4.2.1, NextBug is a plug-in for Bugzilla. Therefore, it is implemented in Perl, the same programming language used in the implementation of Bugzilla. Basically, NextBug instruments the Bugzilla interface used for browsing and for selecting bugs reported for a system. NextBug registers an Ajax event in this interface that calls NextBug back by passing the browsed issue and the filter options as input parameters. We chose an asynchronous event design because it is executed only if NextBug is called by the developer and therefore it does not introduce additional overhead.

Algorithm 1 summarizes the processing of a NextBug event. It first selects the open issues that follow the criteria defined by the filter options (line 3), and then performs standard IR techniques on these issues along with the browsed one (lines 4-9). The processed issues are passed to the recommender module which selects the ones to be presented to the users (line 10). Finally, the produced recommendations are returned to the Bugzilla interface (line 14).

As presented in Figure 4.4, NextBug architecture has two central components:

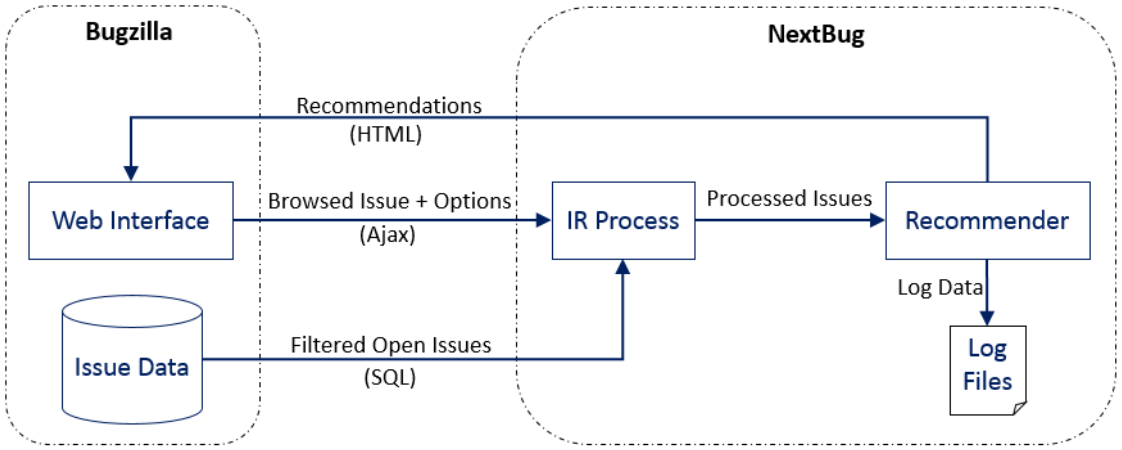


Figure 4.4: NextBug architecture showing the main components and their interactions.

Algorithm 1 Recommendation Algorithm

```
1: function NextBug-Event( BrowsedIssue, Options)
2:     StartTime = get-System-Time-Millisecs( );
3:     FilteredOpenIssues = get-Open-Issues( Options );
4:     q = IR-Processing( BrowsedIssue );
5:     D = ∅;
6:     for doeach issue d′ ∈ FilteredOpenIssues
7:         d_j = IR-Processing(d′);
8:         D = D ∪ d_j ;
9:     end for
10:    Recommendations = recommender( q, D, Options );
11:    EndTime = get-System-Time-Millisecs( );
12:    ExecutionTime = EndTime − StartTime;
13:    log( q, Recommendations, ExecutionTime );
14:    return Recommendations ;
15: end function
```

Information Retrieval (IR) Process and *Recommender*. We discuss these two components in the following subsections.

4.2.2.1 Information Retrieval Process Component

The *IR Process* component obtains the filtered open issues directly from the Bugzilla database as well as the browsed bug. Then it relies on the following standard IR techniques for natural language processing: tokenization, stemming, and stop-words removal [Wang et al., 2008; Runeson et al., 2007]. We implemented all such techniques in Perl. After this initial processing, the issues are transformed into vectors using the Vector Space Model (VSM) [Runeson et al., 2007; Baeza-Yates and Ribeiro-Neto, 1999]. We detailed VSM in Section 2.2.2.

4.2.2.2 Recommender Component

The *Recommender* component receives the processed issues and verifies the ones similar to the browsed one. The similarity is computed using the cosine similarity measure [Runeson et al., 2007; Baeza-Yates and Ribeiro-Neto, 1999]. More specifically, we employed the Equation 2.2 described in Section 2.2.2. Finally, the issues are ordered according to their similarity before being returned to Bugzilla. The recommendations are presented in the same Bugzilla interface used by developers when browsing and selecting bugs to work on (without the need of reloading the webpage because of the asynchronous event).

Logging files are also updated with anonymous information about the process before the recommendations return to the Bugzilla interface. The anonymity is important to preserve the user's privacy and to prevent NextBug being perceived as a spyware program. The logging files register execution time data, the browsed issue, and the recommendations given. The anonymous data collected can also be used in future studies to support analysis and evaluation of NextBug.

4.3 Final Remarks

In this chapter, we detailed our recommender to suggest similar bugs to developers. First, we explained our proposed approach (Section 4.1) and the rationale behind the design decisions for the approach (Section 4.1.1). Second, we presented a prototype tool, called NextBug (Section 4.2), which is implemented as a Bugzilla addon and enhances the standard Bugzilla interface with similar bugs recommendations. Finally, we described NextBug's main features (Section 4.2.1) and its internal design and algorithms (Section 4.2.2).

Chapter 5

Retrospective Study

In this chapter we present a quantitative study, by retrospectively simulating the usage of recommendations of similar bugs in the Mozilla ecosystem and in the Mylyn framework. We rely on metrics proposed to evaluate recommendation systems to assess the quality of the provided suggestions [Zimmermann et al., 2004].

This chapter is divided into four main sections: (i) we begin by comparing our recommendation approach (NextBug) against a state-of-the-art technique to detect duplicate bugs (REP) using 65K bugs from the Mozilla ecosystem (Section 5.1); (ii) we conducted another study using a smaller system (Mylyn) with 2K bugs (Section 5.2); (iii) we discuss threats to validity (Section 5.3); and (iv) we present this chapter final remarks (Section 5.4).

5.1 Comparative Study

In this section, we compare a technique for detecting similar bugs using cosine similarity (as implemented by NextBug) with REP [Sun et al., 2011], a state-of-the-art technique for detecting duplicated bug reports. Our overall aim is to check whether a duplicate bug detection technique is also appropriate for recommending similar bugs.

5.1.1 Data Collection

In this study, we use a dataset of bugs reported for the Mozilla ecosystem. This ecosystem is composed of 69 products including systems such as Firefox,[1] Thunder-

[1] `http://www.mozilla.org/firefox`, verified 2016-05-16

bird,[2] SeaMonkey,[3] and Bugzilla.[4] These systems are from different domains and are implemented in different programming languages. For the study, we initially considered fixed issues (including both bugs and enhancements) reported from January 2009 to October 2012 (130,495 bugs). Figure 5.1 shows the monthly number of issues fixed in this time frame.

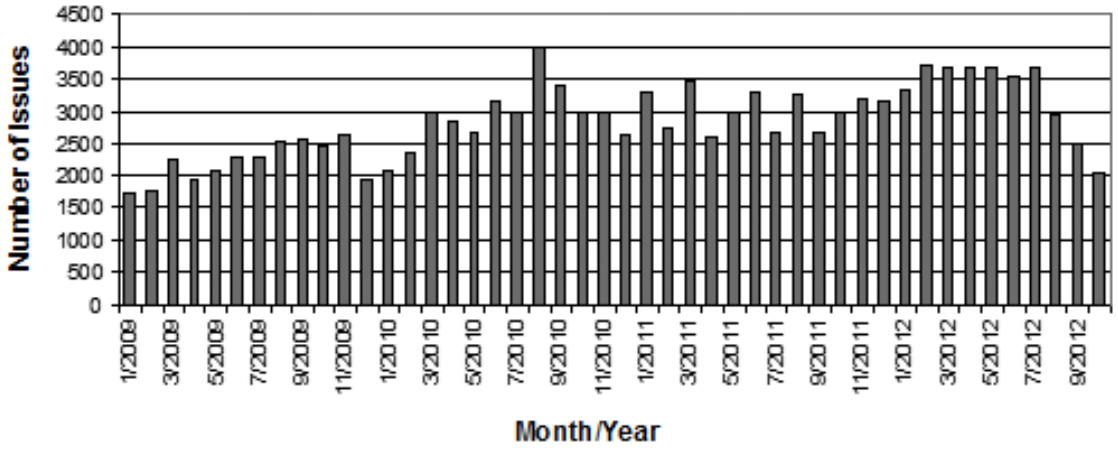


Figure 5.1: Fixed issues per month

Mozilla issues are also classified according to their severity, in the following scale: *blocker*, *critical*, *major*, *normal*, *minor*, and *trivial*. Table 5.1 shows the number and the percentage of each of these severity categories in our dataset. This scale also includes *enhancements* as a particular severity category.

Table 5.1: Issues per Severity

Severity	**Issues**		**Days to Resolve**				
	Number	**Percent.**	**Min**	**Max**	**Avg**	**Dev**	**Med**
blocker	2,720	2.08%	0	814	15.44	52.25	1
critical	7,513	5.76%	0	1258	37.87	99.52	6
major	7,508	5.75%	0	1275	41.59	109.83	5
normal	103,385	79.23%	0	1373	46.27	108.84	8
minor	3,660	2.80%	0	1355	77.05	161.72	11
trivial	2,109	1.62%	0	1288	80.84	164.74	11
enhancement	3,600	2.76%	0	1285	126.14	195.25	40
Total	130,495	100.00%	–	–	–	–	–

Table 5.1 also shows the minimum, maximum, average, standard deviation, and median number of days required to fix the issues in each category. We can observe

[2] `http://www.mozilla.org/thunderbird`, verified 2016-05-16
[3] `http://www.seamonkey-project.org`, verified 2016-05-16
[4] `http://www.bugzilla.org`, verified 2016-05-16

that *blocker* bugs are quickly corrected by developers, showing the lowest values for maximum, average, standard deviation, and median measures among the considered categories. The presented lifetimes also indicate that issues with *critical* and *major* severity are closer to each other. Finally, *enhancements* are very different from the others, showing the highest values for average, standard deviation, and median.

A recommended issue is helpful if it reduces the context switches of a developer. Thus, we evaluate the goodness of a recommendation based on whether two issues required similar files to be changed. For Mozilla, we determine the files changed to conclude a task using a heuristic introduced by Walker et al. [2012]. Normally, in open-source projects, including Mozilla, developers post a patch for a given maintenance issue, which must be approved by certified developers. Thus, patches can be considered as proxies for the files effectively changed when working on the task. By using this heuristic, we linked 65,590 issues (bugs/tasks/tickets/enhancements) to patches and after that to the files in the patches. Figure 5.2 shows a histogram with the frequency of the number of files changed by the considered issues. Table 5.2 presents descriptive statistics about the values in this histogram. As we can observe, most issues change few files (two files, on the median), as also concluded by other studies using Mozilla data [Walker et al., 2012].

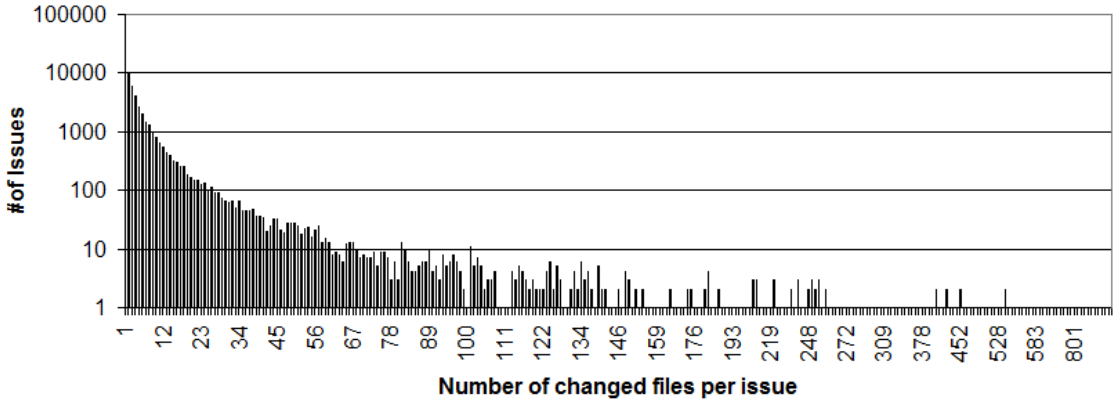


Figure 5.2: Files changed per mapped issue

Table 5.2: Files changed per mapped issue

Average	6.35
Standard Deviation	27.48
Median	2.00
Maximum	2,043.00

Table 5.3 presents the top-10 Mozilla systems by number of issues mapped to source code files. The table also shows the mean number of issues per component,

and the standard deviation. With the exception of Testing and SeaMonkey, the other systems in this table have at least 100 issues/component, but usually with relevant variations on each system, as expressed by the high standard deviation values.

Table 5.3: Top-10 Mozilla systems by issues with changed files

System	Components	Issues	Issues/Component
Core	121	28,258	233 ± 642
Firefox	38	4,732	124 ± 160
Mozilla.org	25	3,679	147 ± 514
Toolkit	30	3,044	101 ± 136
Fennec	5	2,667	533 ± 931
Firefox for Android	11	2,068	188 ± 504
Thunderbird	20	2,060	103 ± 77
Tamarin	15	1,913	127 ± 209
Testing	27	1,878	70 ± 91
SeaMonkey	32	1,764	55 ± 50

5.1.2 Technique for Comparison: REP

REP [Sun et al., 2011] is a technique to detect duplicated bugs that employs an extension on the BM25F textual similarity function. REP also uses non-textual fields (such as component, version, priority, etc.) along with a stochastic gradient and a two-round tuning to train a duplicated bug retrieval function.

The classical BM25F was designed for search engines, for this reason it handles short queries. Also, in search engines, the queries do not have repeated words. As argued by Sun et al. [2011], those characteristics do not apply to bug reports. When we use a bug report as a query for information retrieval, the report itself may be long and contain repeated words to describe the issue. The BMF25F extension takes these particular characteristics into account, and the similarity function for a query q and a bug b is defined as:

$$BM25F_{ext}(q,b) = \sum_{t \in b \bigcap q} IDF(t) \times \frac{TF_B(b,t)}{k_1 + TF_B(b,t)} \times \frac{(k_3+1) \times TF_Q(q,t)}{k_3 + TF_Q(q,t)} \quad (5.1)$$

where t is the term extracted from the textual element that appears in both the query q and the bug b, $IDF(t)$ is the inverse document frequency of the term t, TF_B is the term frequency in a bug b, TF_Q is the term frequency in the query q, and finally, k_1 and k_3 are calibration weights to adjust the similarity.

The term frequency according to a bug is defined as:

$$TF_B(b,t) = \sum_{f=1}^{K} \frac{w_f \times occurrences(b[f],t)}{1 - d_f + \frac{d_f \times length_f}{average_length_f}} \tag{5.2}$$

where f is the field, w_f is the field weight, *occurrences* is the number of occurrences of term t in the field f, $length_f$ is the size of the bag $b[f]$, $average_length_f$ is the average size of the bag $b[f]$ considering all other bugs, and d_f is a weight to scale the field.

The term weight frequency related to a query follows a different equation:

$$TF_Q(q,t) = \sum_{f=1}^{K} w_f \times occurrences(q[f],t) \tag{5.3}$$

The $BM25F_{ext}$ function is used to process the short and full descriptions together. The $BM25F_{ext}$ can be used to process single words (unigrams), two-words (bigrams), or even a set of words (n-grams). The REP retrieval function is a linear combination of seven features, where two of those features are textual information processed through $BM25F_{ext}$ (unigram and bigram), and the other five features use categorical information. The REP function between a query q and a bug b is defined as follows:

$$REP(q,b) = \sum_{i=1}^{7} w_i \times feature_i(q,b) \tag{5.4}$$

where w_i is the weight of the i-th feature, and $feature_i(q,b)$ is the returned value from computing the i-th feature. Table 5.4 describes how the features are calculated.

All of the seven weights assigned to the features require tunning. Moreover, each $BM25F_{ext}$ execution has six weights that require calibration.[5] In total, REP has 19 free parameters. Table 5.5 shows all the free parameters, their description, and their initial values before training (or tunning).

REP employs a gradient descent to optimize the parameter values, as described in Algorithm 2. The training set used by this algorithm is a collection of training instances of the form $(q, relevant, irrelevant)$, where q is the query (i.e., bug report), *relevant* is a relevant bug as a response to the query (i.e., a relevant duplicate bug in the original work), and *irrelevant* is an irrelevant bug in relation to the query (i.e., not a duplicate bug in the original work). To adapt the algorithm to handle similar bugs, a

[5]As seen in Table 5.4, features 1 and 2 require a $BM25F_{ext}$ execution on the short and long description of the bug report. Therefore, REP executes $BM25F_{ext}$ two times.

Table 5.4: Features of $REP(q, b)$

Description	Class	Equation
Unigram	Textual	$feature_1(q,b) = BM25F_{ext}(q,b)$ of unigrams
Bigram	Textual	$feature_2(q,b) = BM25F_{ext}(q,b)$ of bigrams
Product	Categorical	$feature_3(q,b) = \begin{cases} 1 & \text{if } q.product == b.product \\ 0 & \text{otherwise} \end{cases}$
Component	Categorical	$feature_4(q,b) = \begin{cases} 1 & \text{if } q.component == b.component \\ 0 & \text{otherwise} \end{cases}$
Type	Categorical	$feature_5(q,b) = \begin{cases} 1 & \text{if } q.type == b.type \\ 0 & \text{otherwise} \end{cases}$
Priority	Categorical	$feature_6(q,b) = \frac{1}{1 + \lvert b.priority - q.priority \rvert}$
Version	Categorical	$feature_7(q,b) = \frac{1}{1 + \lvert b.version - q.version \rvert}$

similar bug is used to compose *relevant* (using an oracle as described in Section 5.1.3) and the *irrelevant* with a non-similar bug. In this way, we trained the technique to optimize the parameter to find similar bugs.

Algorithm 2 REP Parameter Tuning Algorithm

```
1: function REP-PAR-TUNING( TrainingSet, TuningRate, MaxIterations)
2:     for n = 1 to MaxIterations do
3:         for each instance I ∈ TrainingSet in random order do
4:             for each free parameter par in instance I do
5:                 par = par − TuningRate × ∂RNC/∂par (I)
6:             end for
7:         end for
8:     end for
9: end function
```

In Algorithm 2, $\frac{\partial RNC}{\partial par}(I)$ is the partial derivative of the cost function RNC with respect to the parameter par. The RNC cost function uses the training instance $I(q, relevant, irrelevant)$ to compute the similarity score between the query q and the *irrelevant* bug against the similarity between the query and the *relevant*. Low RNC values tend to increase the accuracy of the similarity calculation. Throughout each iteration, the free parameters are adjusted towards a minimum RNC.

Algorithm 3 describes the steps for the two-round tuning, basically it runs the tuning algorithm (Algorithm 2) twice. A fixed parameter is not adjusted by the tuning algorithm, which is performed to avoid redundant tunning.

Table 5.5: Parameters of REP

Par	Description	Init.
w_1	weight of $feature_1$ unigram	0.9
w_2	weight of $feature_2$ bigram	0.2
w_3	weight of $feature_3$ product	2.0
w_4	weight of $feature_4$ component	0.0
w_5	weight of $feature_5$ type (bug or enhancement)	0.7
w_6	weight of $feature_6$ priority	0.0
w_7	weight of $feature_7$ version	0.0
w_8	weight w_f of short description ($feature_1$)	3.0
w_9	weight w_f of full description ($feature_1$)	1.0
w_{10}	weight d_f for term frequency (TF_B) in short description ($feature_1$)	0.5
w_{11}	weight d_f for term frequency (TF_B) in full description ($feature_1$)	1.0
w_{12}	k_1 weight used in $BM25F_{ext}$ ($feature_1$)	2.0
w_{13}	k_3 weight used in $BM25F_{ext}$ ($feature_1$)	0.0
w_{14}	weight w_f of short description ($feature_2$)	3.0
w_{15}	weight w_f of full description ($feature_2$)	1.0
w_{16}	weight d_f for term frequency (TF_B) in short description ($feature_2$)	0.5
w_{17}	weight d_f for term frequency (TF_B) in full description ($feature_2$)	1.0
w_{18}	k_1 weight used in $BM25F_{ext}$ ($feature_2$)	2.0
w_{19}	k_3 weight used in $BM25F_{ext}$ ($feature_2$)	0.0

Algorithm 3 REP Two-Round Tuning Algorithm

1: initialize free parameters in REP with default values
2: fix parameters w_{12},w_{13},w_{18}, and w_{19} (k_1 and k_3 weights used for $BM25F_{ext}$ in $feature_1$ and $feature_2$).
3: execute Algorithm 2 with $|TrainingSet| = 30$, $TuningRate = 0.001$, $MaxIterations = 24$
4: unfix w_{13} and w_{19} (k_3 weights).
5: fix w_8, w_9, w_{10}, w_{11}, w_{14}, w_{15}, w_{16}, w_{17} (d_f for term frequency, w_f short and full description weights used for $BM25F_{ext}$ in $feature_1$ and $feature_2$).
6: execute again Algorithm 2 (same configuration as first execution)

5.1.3 Study Design

For the comparison between NextBug and REP, we use the same algorithms and parameters reported in the original work about REP [Sun et al., 2011]. For example, in the parameter tuning phase, we use 30 training instances, 24 iterations, and an adjustment coefficient of 0.001. The difference is that we use similar bugs as training set, rather than duplicated ones. To compose the training set we randomly selected bugs from our dataset and each bug was paired with two others: a similar bug and a non-similar one. We select similar bugs from an oracle, as described next. We also normalize the results provided by the REP function. Basically, we divide the results of $REP(q, b)$, where q is a query and b is a document (or bug), by the maximal pos-

sible result, i.e., $REP(q, q)$. In this way, REP normalized results range from 0 to 1. When evaluating and comparing the techniques, the bugs similar to a query q are the ones with a similarity measure (NextBug) and a normalized result (REP) greater than a threshold τ. Our implementation of REP and NextBug is publicly available on GitHub.[6]

We simulate NextBug and REP by retrospectively computing recommendations for the bugs in our dataset. Our goal is to reconstruct a scenario where similar bug recommendation lists would be in place when each bug in our dataset was marked as closed. To reconstruct this scenario, we call B_q the set of pending bugs at the exact moment that each bug q was fixed. For each bug $b \in B_q$, the similarity measure (or REP function result) is then used to decide whether b is similar to q or not. A lower bound threshold τ is used in this check. Finally, we call A_q the set of similar bugs that would be recommended for q, $A_q \subseteq B_q$.

After producing the recommendations, we evaluate whether each bug $r \in A_q$ is a useful recommendation by checking if both r and q changed similar files. Suppose that bugs q and r require changes in the sets of files F_q and F_r, respectively. The similarity of F_q and F_r is calculated using the Overlap coefficient [Rijsbergen, 1979]:

$$Overlap(F_q, F_r) = \frac{|F_q \cap F_r|}{min(|F_q|, |F_r|)} \tag{5.5}$$

The Overlap result is equal to one (maximal value) in the situations illustrated in Figure 5.3. First, when $F_q \subseteq F_r$, i.e., the "context" established by the developer when working on the bug q is reused when working on the recommended bug r. Second, when $Fr \subseteq Fq$, i.e., to work on the recommended bug r the developer does not need to set up new "context" items. In both cases, the developer concludes two bugs and one of the required contexts is completely reused.

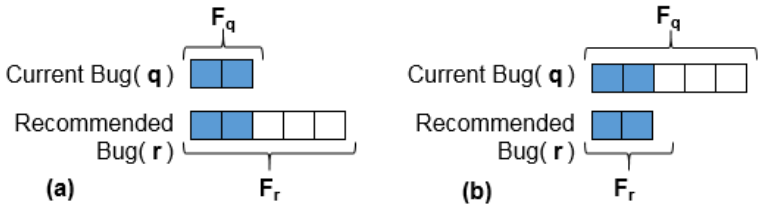


Figure 5.3: Maximal overlap scenarios (boxes represent files changed to conclude a bug): (a) the context of the first bug is reused by the recommended bug; (b) no new files are included in the context of the recommended bug.

[6] `https://github.com/hscrocha/NextBug-REP-Comparative-Study-Impl`, verified 2016-05-16.

Overlap similarity is used to create an oracle O_q with the relevant bugs that must be recommended with q, as follows:

$$O_q = \{\ b \in B_q \mid Overlap(F_q, F_b) \geq 0.5\} \tag{5.6}$$

This oracle includes the pending bugs b at the moment that q was concluded (set B_q) and whose overlap coefficient calculated using the files changed by q and b is greater than 0.5. We are assuming that 50% of "context reuse" between bugs is enough to convince a developer to fix the bugs consecutively (or concurrently).

The described oracle is used only to evaluate the quality of the provided recommendations. Both techniques, NextBug and REP, do not employ change file information for their recommendations.

5.1.4 Evaluation Metrics

We evaluate both techniques using four metrics for recommendation systems: feedback, precision, likelihood, and recall. These metrics are inspired by the evaluation followed by the ROSE recommendation system [Zimmermann et al., 2004]. Although, ROSE targets a different context, their metrics are appropriate to evaluate other recommendation systems. We also calculated F-measure, which combines both precision and recall into an averaged weighted result.

5.1.4.1 Feedback

Assuming that Z is the set of queries and that Z_k is the set of queries with at least k recommendations, feedback is the ratio of queries with at least k recommendations:

$$Fb(k) = \frac{|Z_k|}{|Z|} \tag{5.7}$$

For example, suppose a recommendation system that executed 100 queries ($|Z| = 100$). If all those queries returned at least one recommendation each, then Fb(1) is 100%. On the other hand, if only 40 queries returned at least 3 recommendations, then $Fb(3) = 40\%$.

Feedback is a useful metric for recommendation systems, because a recommender that rarely gives recommendations is not practical. However, feedback is not the only important point to evaluate, as a recommender that gives too many imprecise recommendations is not trustworthy.

5.1.4.2 Precision

Precision is the ratio of relevant recommendations provided by the recommender. Assuming that $R_q(k)$ are the top-k recommendations more similar to the query q (measured in terms of the similarity measure or the normalized REP results), precision is defined as follows:

$$P_q(k) = \frac{|R_q(k) \cap O_q|}{|R_q(k)|} \tag{5.8}$$

The overall precision is the average of the precisions calculated for each query:

$$P(k) = \frac{1}{|Z_k|} \sum_{q \in Z_k} P_q(k) \tag{5.9}$$

Suppose that a query returned four recommendations. If the first recommendation is a relevant one, then $P_q(1) = 100\%$. Otherwise, we have $P_q(1) = 0\%$ (for a non-relevant first recommendation). Moreover, if among the top four recommendations only the second one is relevant, then the precision values would be $P_q(2) = 50\%$, $P_q(3) = 33\%$, and $P_q(4) = 25\%$.

5.1.4.3 Likelihood

Likelihood checks whether there is at least one relevant suggestion (i.e., included in the proposed oracle) among the provided recommendations. Likelihood provides an indication of how often a query to produce recommendations provides at least one potentially useful result. Likelihood is defined as follows:

$$L_q(k) = \begin{cases} 1 & \text{if } R_q(k) \cap O_q \neq \emptyset \\ 0 & \text{otherwise} \end{cases} \tag{5.10}$$

Therefore, $L_q(k)$ is a binary measure. If there is at least one useful recommendation among the top-k recommendations, it returns one; otherwise, it returns zero. It is worth to mention that likelihood(1) is always equal to precision(1), by definition. The overall likelihood is the average of the likelihood computed for each query:

$$L(k) = \frac{1}{|Z_k|} \sum_{q \in Z_k} L_q(k) \tag{5.11}$$

For example, suppose that a query returned four recommendations but only the

third one is relevant. Then the likelihood values would be $L_q(1) = 0\%$, $L_q(2) = 0\%$, $L_q(3) = 100\%$, and $L_q(4) = 100\%$.

We compute likelihood because a recommender might be useful even when it provides one or two incorrect recommendations plus one clearly useful result. In this case, humans with some experience in the domain can rapidly discard the incorrect recommendations and focus on the useful one [Zimmermann et al., 2004; Shani and Gunawardana, 2011].

5.1.4.4 Recall

Recall is the ratio of recommendations in the oracle among the provided recommendations. The recall up to k recommendations is defined as follows:

$$Rc_q(k) = \frac{|R_q(k) \cap O_q|}{|O_q|} \tag{5.12}$$

The overall recall is the average of the recall calculated for each query:

$$Rc(k) = \frac{1}{|Z_k|} \sum_{q \in Z_k} Rc_q(k) \tag{5.13}$$

For example, suppose a query returned three recommendations and all of them are relevant. Suppose also that the oracle size is ten ($|O_q| = 10$) , i.e., the oracle includes ten relevant bugs for this query. In this case, the recall values are $Rc_q(1) = 10\%$, $Rc_q(2) = 20\%$, and $Rc_q(3) = 30\%$.

Since the number of recommendations is limited by the top-k, the maximum possible recall for a given query may not be 100%. In the last example, all recommendations are relevant however if we compute recall(3), the maximum possible result considering the oracle ($|O_q| = 10$) is 30%. For this reason, recall values are usually lower than precision and we also decided to compute the maximum possible recall for a more fair comparison. The maximum possible recall for k recommendations is defined as follows:

$$Max\ Rc_q(k) = \frac{k}{|O_q|} \tag{5.14}$$

In our case, recall can be perceived as less important than feedback. Indeed, a recommender that gives results for most queries (feedback) is still useful, even if it does not cover all relevant recommendations (recall).

5.1.4.5 F-score

F-score is a measure of accuracy that considers both precision and recall. The classical F-score (also known as F-measure or F_1-score) is a balanced weighted average between precision and recall. Since we compute precision and recall for a list of k recommendations, F-score also considers k recommendations as follows:

$$F_1(k) = 2 \times \frac{P(k) \times Rc(k)}{P(k) + Rc(k)} \tag{5.15}$$

Summary: Feedback is the ratio of queries with at least a given number of recommendations. Precision is the ratio of relevant recommendations, i.e., recommendations found in the oracle. Likelihood checks whether there is at least one relevant recommendations among the provided suggestions. Recall is the ratio of recommendations in the oracle among the provided recommendations. Finally, F-score is a weighted average between precision and recall.

5.1.5 Comparison Results

Figures 5.4 to 5.8 show results of the evaluation metrics, for thresholds ranging from 0.0 to 0.8 (τ parameter). When the threshold is zero, both techniques act as a ranking function, as the similarity test considers any pending bug.

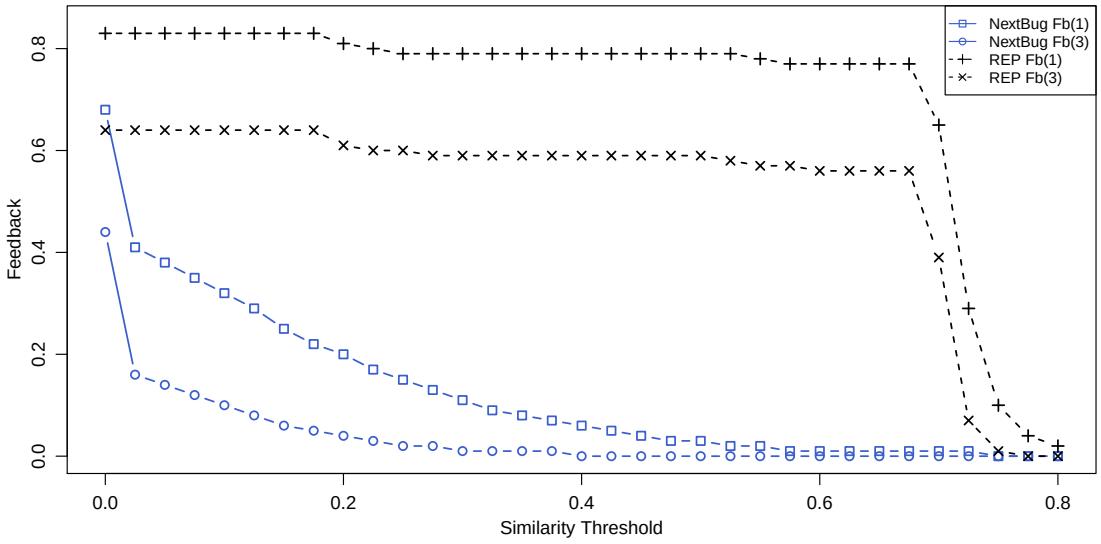


Figure 5.4: Feedback (Fb)

Figure 5.4 shows the feedback results. REP provides more recommendations

(maximum Feedback of 83%) than NextBug (maximum Feedback of 68%). NextBug feedback reveals how the technique is sensible to the threshold parameter. When we increase the similarity threshold, NextBug feedback decreases because the similarity test becomes more strict. When the threshold is zero, NextBug recommends less bugs than REP due to its component filter, i.e., NextBug only recommends bugs that share the same component as the query. On the other hand, there is a small impact on REP feedback as we increase the threshold. The main reason is that REP similarity function scores are usually high due to the use of many features (e.g., product, version, priority, etc.). Thus, it is very likely that it exists a pending bug sharing at least some features with the query. For example, the recommendations' average similarity for NextBug is 0.17, and for REP is 0.73 (considering a threshold of zero).

Regarding precision (Figure 5.5), both techniques show precision(1) values around 70%, and precision(3) values greater than 62%. One technique may perform slightly better than REP depending on the similarity threshold. For example, if we consider a threshold of zero, then REP has a better precision(1) than NextBug (71% and 70%, respectively). However, for a threshold of 0.2, REP precision(1) is lower than NextBug (71% and 73%, respectively). Moreover, REP precision slightly varies for thresholds lower than 0.6 because the feedback values also show minor variation in these cases.

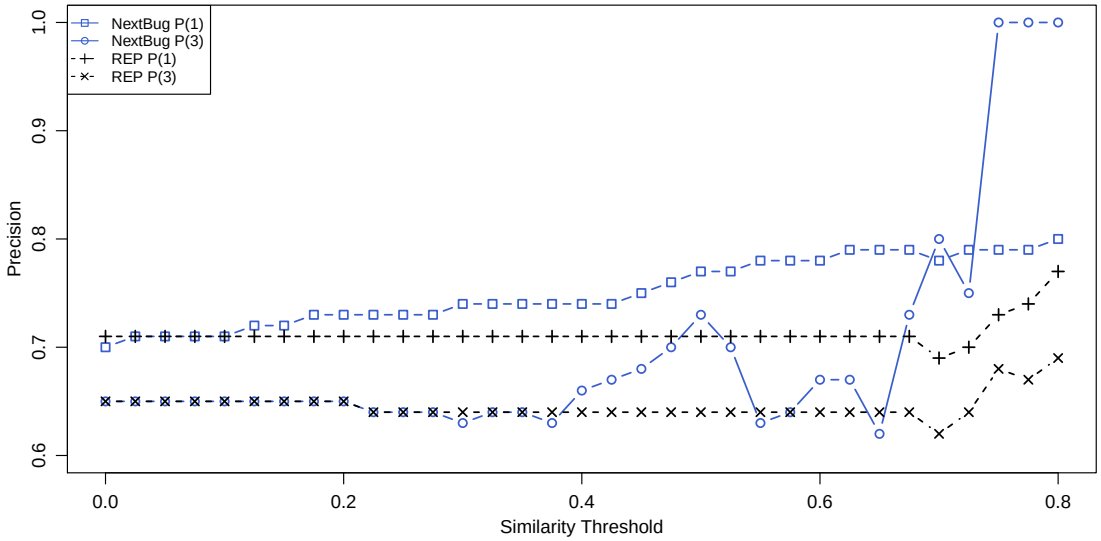


Figure 5.5: Precision (P)

Figure 5.6 shows that both techniques perform well for likelihood. Since by definition, likelihood(1) is equal to precision(1), we focus the likelihood analysis on a list of three recommendations. NextBug has a minimum likelihood(3) of 79% for a similarity threshold of 0.65, and REP has a minimum likelihood(3) of 80% for a

similarity threshold of 0.7.

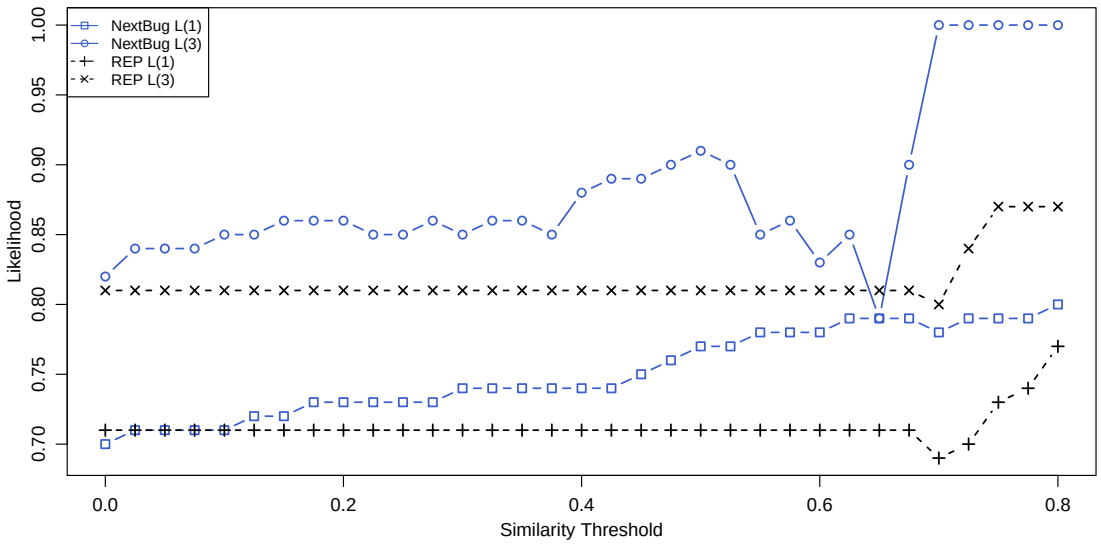
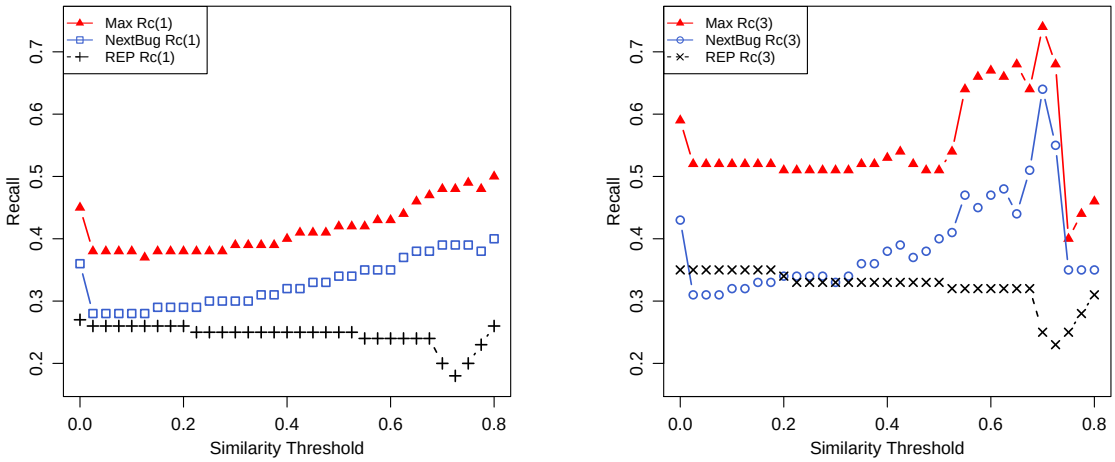


Figure 5.6: Likelihood (L)

Considering recall, NextBug performs better than REP as shown in Figure 5.7. For example, assuming a similarity threshold of zero, the recall(1) values for NextBug and REP are 36% and 27%, respectively. For the same threshold, recall(3) values are 43% for NextBug and 35% for REP. Although recall results may seem low, they are limited by the maximum possible recall. The max recall(1) for threshold of zero is 45%, and max recall(3) is 59%. Moreover, the recall results are comparable with other recommendation systems. For example, ROSE (a system that recommends classes that usually change together) has an average recall of 33% (in the fine-grained navigation scenario) [Zimmermann et al., 2004]. Another recommendation system that suggests the most suitable developer to fix a bug has a recall of 10% (highest average recall) [Anvik et al., 2006].

Finally, Figure 5.8 shows the F_1-scores. NextBug shows higher F_1 values considering top-1 recommendations. For example, considering a threshold of zero and 0.4, NextBug $F_1(1)$ scores are 48% and 45%, while REP scores are 39% and 37%, respectively.

Summary: The reason for NextBug (a simple technique) to perform just as well as REP (a more complex function) is because the textual description field of a bug report appears to be the most relevant field to predict similar bugs. As such, the extra fields processed by REP do not contribute significantly to better recommendations although they do contribute to an increase in the number of suggestions.



(a) Top-1 recommendations (b) Top-3 recommendations

Figure 5.7: Recall (Rc) and Maximum Recall ($Max\ Rc$)

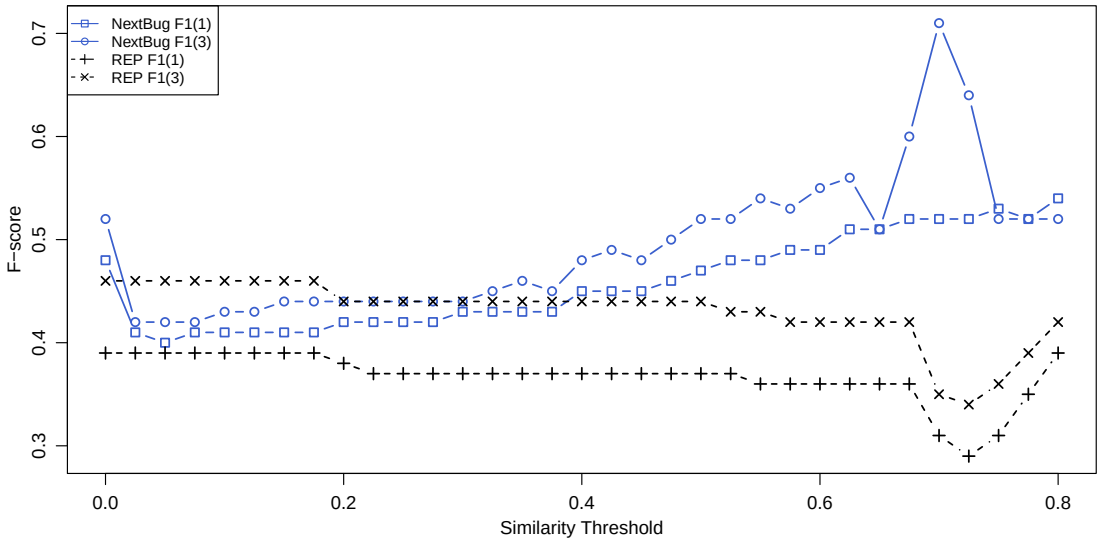
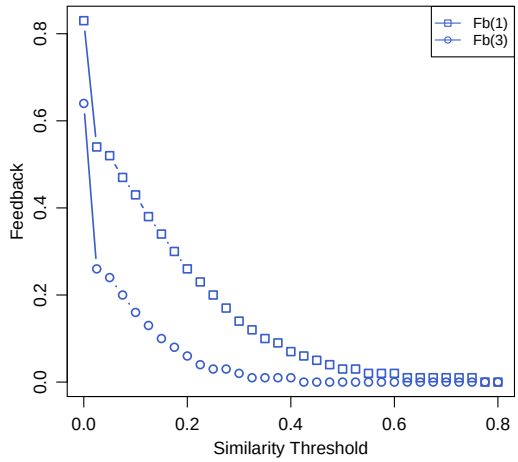


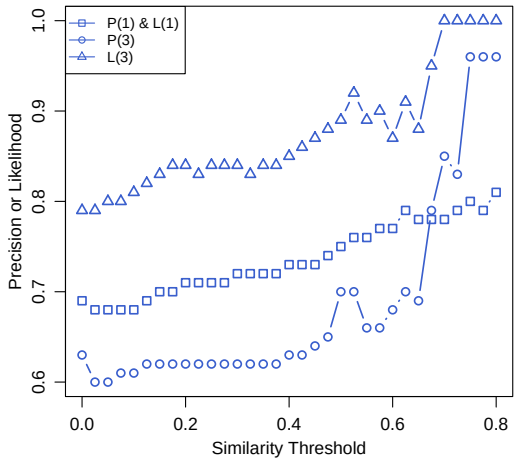
Figure 5.8: F-score (F_1)

5.1.6 Configuration Without the Component Filter

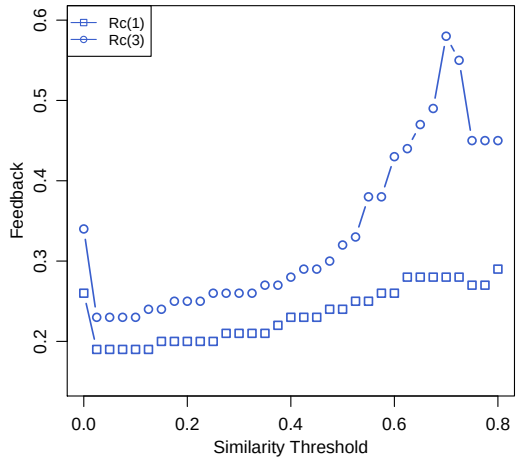
We also investigate the relevance of the component filter in NextBug results. As mentioned, NextBug only recommends bugs that share the same component with the query. For this reason, it shows a lower feedback than REP, which does not have this filter. Figure 5.9a shows the feedback results when NextBug is configured without this component filter. As we can see, the results are very similar to the ones presented earlier by REP earlier (Figure 5.4). For example, for a similarity threshold of zero, feedback(1) is 83% for both NextBug and REP. However, even without the component filter, the similarity threshold (τ) still has a relevant impact on NextBug's feedback values.



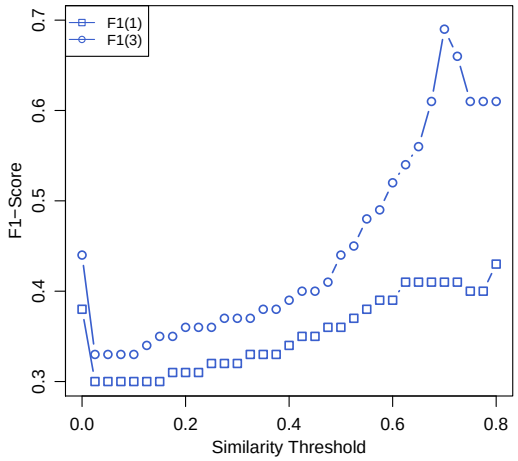
(a) Feedback (Fb)



(b) Precision (P) and Likelihood (L)



(c) Recall (Rc)



(d) F-score (F_1)

Figure 5.9: NextBug Feedback (Fb), Precision (P), Likelihood (L), Recall (Rc), and F-score (F_1) without the component filter

Although it increases feedback, removing the component filter has an impact on the remaining metrics. Precision and likelihood (Figure 5.9b) shows slightly lower values without the filter. For example, without the filter, NextBug precision(1), precision(3), and likelihood(3) are respectively 69%, 63%, and 79% (considering a similarity threshold of zero). Using a filter, the same metrics values are 70%, 65%, and 82%. The results without the filter are lower than those provided by REP (71%, 65%, and 81%).

Removing the filter has a more noticeable impact on recall. As we can see in Figure 5.9c, the recall values without the filter are lower than those with the filter (Figure 5.7). For example, NextBug recall(1) is 36% with and 26% without the filter, considering a similarity threshold of zero. Recall(3) values are 43% with and 34% without the filter. The results without the filter are closer to the ones showed by

REP (recall(1) of 27% and recall(3) of 35%). The maximum recall is not affected by removing the filter, and the results remain the same as presented earlier in Figure 5.7.

Since both precision and recall results are affect by disabling (or not) the component filter, F-score shows a similar variation. For example, Figure 5.9d shows the F_1 scores for a NextBug instance without component filtering. For a similarity threshold of zero, NextBug's $F_1(1)$ score decreases from 48% (with filter) to 38% (without filter). However, it is very close to the results provided by REP (39%).

Summary: Removing the component filter increases the feedback values for NextBug but has a negative impact on the remaining metrics. Therefore, we claim that NextBug performs better when its component filter is turned on.

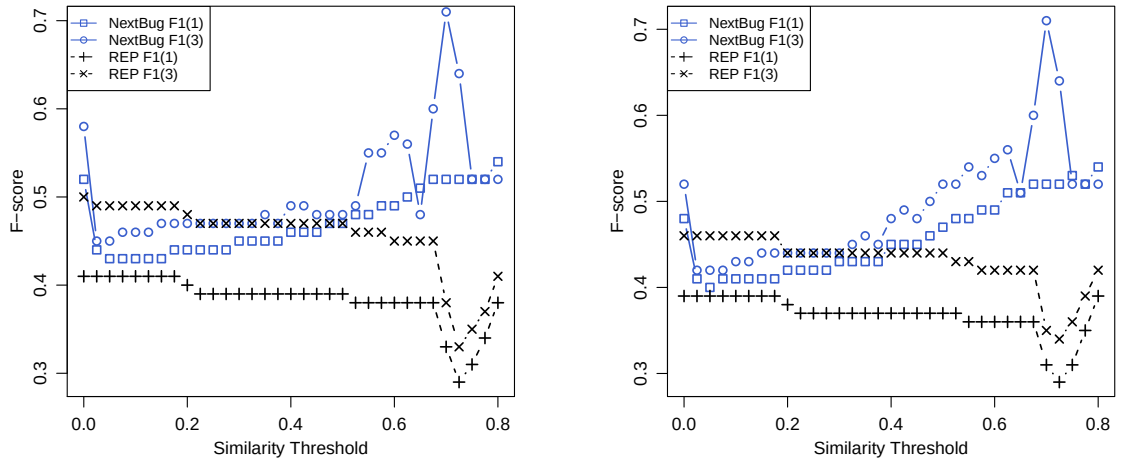
5.1.7 Oracle Sensibility Testing

Most of the evaluation metrics (precision, likelihood, recall, and F_1-score) use the defined Oracle O_q to verify whether a recommendation is relevant. We assumed that an overlap coefficient of 0.5 or higher, i.e., 50% of "context reuse" in terms of changed files would be good enough to convince developers to fix these bugs. However, the selection of 0.5 was arbitrary (it was based on user experience in the context) and different results are possible by varying the overlap criteria. For this reason, we changed the oracle to account for different overlap values:

$$O_q = \{\ b \in B_q \mid Overlap(F_q, F_b) \geq \Omega\} \tag{5.16}$$

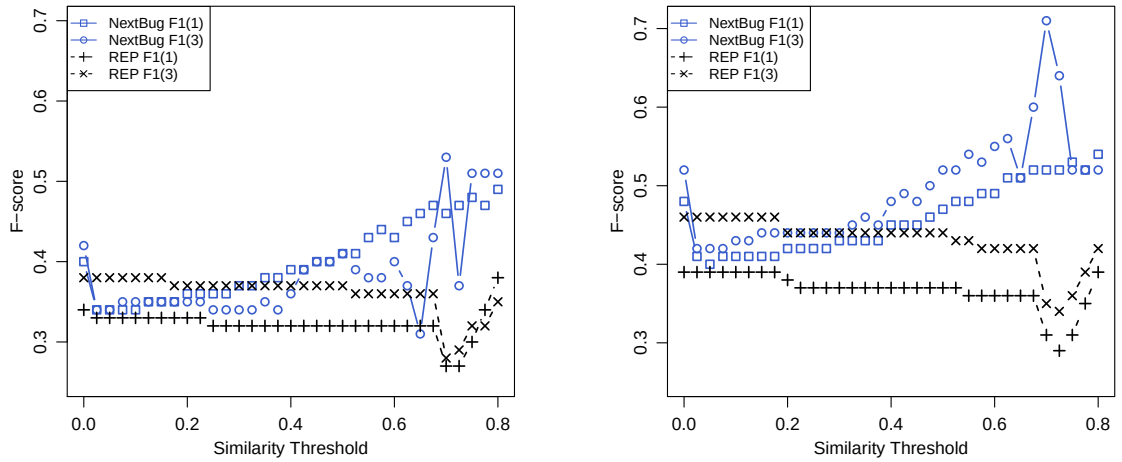
First, we present results when we set Ω=0.25. Since we are setting Ω to a lower value, the Oracle becomes less strict, because only a 25% of "context reuse" is necessary for a bug to be considered relevant. As a consequence, we expected the metrics to show higher results. We did not present feedback because its results are unaffected by the Oracle.

We show only the F-score results to summarize the metrics (Figure 5.10). F-scores increase for $\Omega = 0.25$ for similarity thresholds of 0.6 and lower. For example, for a similarity threshold of zero and $\Omega = 0.25$, NextBug and REP scores are 52% and 41% for $F_1(1)$, 58% and 50% for $F_1(3)$. Considering the same similarity and $\Omega = 0.5$ (original Oracle criteria), F-score results for NextBug and REP respectively are: $F_1(1)$ of 48% and 39%, $F_1(3)$ of 52% and 46%.



(a) $\Omega = 0.25$ (b) $\Omega = 0.5$

Figure 5.10: F-score (F_1): (a) $\Omega = 0.25$, (b) $\Omega = 0.5$



(a) $\Omega = 0.75$ (b) $\Omega = 0.5$

Figure 5.11: F-score (F_1): (a) $\Omega = 0.75$, (b) $\Omega = 0.5$

Figure 5.11 presents the F-score results for $\Omega = 0.75$ and $\Omega = 0.5$. We expect an opposite impact as the oracle becomes more strict, i.e., we anticipate the results to show lower values when compared to the original Oracle ($\Omega = 0.5$). For instance, NextBug and REP scores are respectively, $F_1(1)$ of 40% and 34%, and $F_1(3)$ of 42% and 38% (for a similarity of zero and $\Omega = 0.75$). Considering the original Oracle ($\Omega = 0.5$) and the same similarity, F-scores results are $F_1(1)$ of 48% and 39%, and $F_1(3)$ of 52% and 46% (for NextBug and REP, respectively).

Summary: When we change the overlap criteria (Ω) used to define the Oracle, it affects the metrics results (excluding feedback). Setting $\Omega = 0.25$ increase these metrics, because the Oracle becomes less strict and more recommendations are considered relevant. When we set $\Omega = 0.75$ it has an opposite effect, because the Oracle becomes more strict the metrics' results decrease.

5.1.8 Alternative Evaluation: Assigned Developers

The previous measures are centered on the following external similarity criteria: two bug reports are similar whenever their implementation require changes to similar source code files. In this section, we consider an alternative measure of similarity: a bug q is similar to an opened issue when they were assigned to the same developer. The assumption in this case is that a recommendation is useful when it matches a developer that fixed the bug.

Following this measure of relevance, we can redefine the Oracle O_q as follows (where $developer(x)$ is the developer that fixed an issue x).

$$O_q = \{\ b \in B_q \mid developer(q) = developer(b)\ \} \tag{5.17}$$

Using this redefinition of O_q it is possible to recalculate likelihood and precision using the same formulas described earlier (Section 5.1.4). As we mentioned before, feedback results are unaffected by how the Oracle is defined, and for this reason we do not present feedback results. We do not compute recall using this new Oracle because using it in the recall formula would give the same results as precision. One can argue that it is possible to change the developer's Oracle to contain every developer in the dataset, in a similar way as done by Anvik et al. [2006]; Anvik and Murphy [2011]. However, such change would not provide reasonable results for the Mozilla ecosystem which has thousands of developers working on similar bugs.

Figure 5.12 shows precision results for this alternative relevance measure. As we can observe, the results are very different from those presented in Figure 5.5. NextBug shows better precision results than REP using this alternative Oracle. As the similarity threshold increases so does the precision of NextBug, and the difference between NextBug and REP results. REP precision varies slightly for similarities of 0.7 and lower. For example, considering a similarity threshold of zero, NextBug shows precision(1) = 24% and precision(3) = 20%, while REP shows precision(1) = 23% and precision(3) = 18%. Increasing the similarity threshold to 0.2, NextBug results increases

to precision(1) = 35% and precision(3) = 34%, while REP results are precision(1) = 23% and precision(3) = 18%.

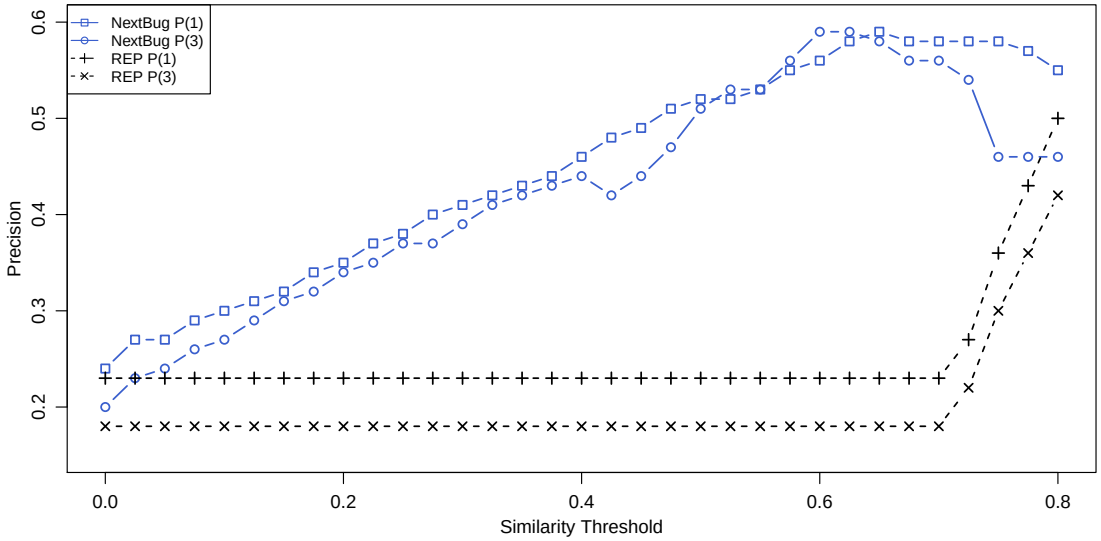


Figure 5.12: Precision considering assigned developers

Figure 5.13 shows likelihood results considering the alternative measure. The likelihood results also show lower results when compared to the ones in Figure 5.6. For instance, likelihood(3) results for NextBug and REP respectively are, 40% and 38% (considering a similarity threshold of zero). Increasing the similarity threshold to 0.2, likelihood(3) results are 58% and 38% for NextBug and REP, respectively.

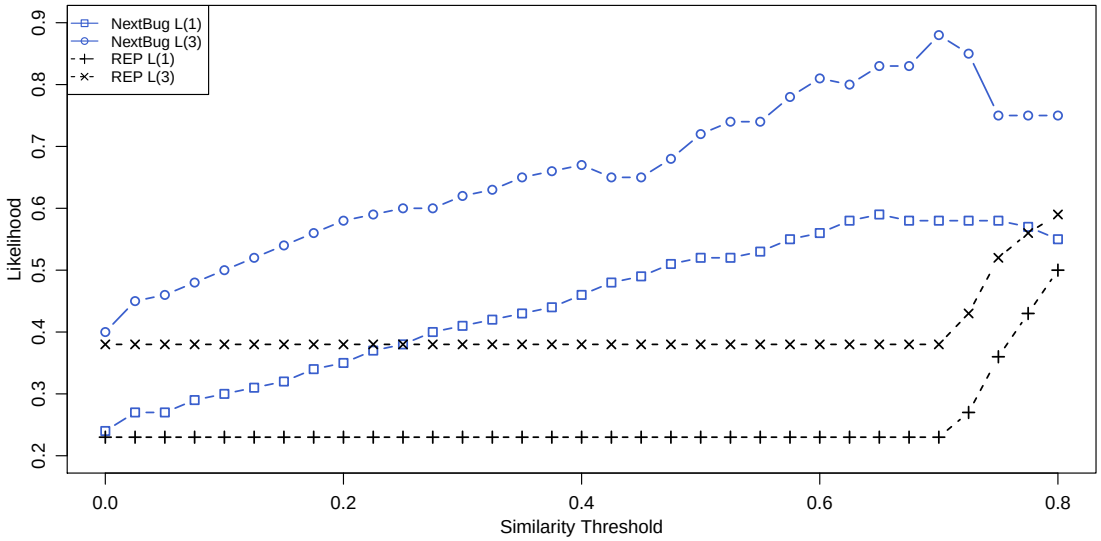


Figure 5.13: Likelihood considering assigned developers

The results indicates that NextBug recommends more issues than the ones that were handled by the same the developer.

Summary: When we analyze the similar bugs suggestions regarding if the same developer handled both the query and the recommendations, NextBug outperforms REP by showing better precision and likelihood.

5.1.9 Execution Time

Table 5.6 shows the time to compute the recommendations evaluated in this study. The execution time refers to an HP Server, CPU Xeon Six-Core E5-2430 2.20 GHz, 64 GB RAM, operating system Ubuntu 12.04, 64 bits. Both techniques (NextBug and REP) are implemented in Java and executed in a Java Virtual Machine (JVM) 1.7.0 80. The execution time only considers the information retrieval processing steps, the similarity test (according to each technique), and the time to return the recommendation list. Particularly, we do not consider the time to process the SQL queries that retrieve the pending bugs reported in a given date (since in our study all bugs are initially loaded in main memory, to optimize performance). For REP, the execution time shown in Table 5.6 does not include the training stage (or parameter tuning), since it can be performed off-line. We execute the experiments three times, using the threshold zero (the one with the highest feedback). We report the average values for the execution time results.

Table 5.6: Execution Time (ms)

	Min	**Max**	**Avg**	**Med**	**Std Dev**
NextBug	0.0	163.6	3.8	3.7	2.1
REP	0.0	1,117.3	11.3	11.4	6.9

As can be observed in Table 5.6, the maximum time to provide a recommendation is 163 milliseconds for NextBug and 1.1 seconds for REP. NextBug average, median, and standard deviation time are also lower than REP.

Summary: The reported execution times show that is feasible to (re-)compute on-the-fly the recommendations for both techniques each time a developer requests a web page with a bug report.

5.1.10 Summary of Findings

For detecting similar bugs (i.e., bugs requiring changes in the same parts of a system), a technique that considers just two bug report fields (component and short description) performs just as well as REP, a technique optimized for detecting duplicate bug reports. Furthermore, both techniques have a runtime performance that supports their online integration with a issue tracker system.

5.2 Second Study: Mylyn

The goal of this study is to analyze NextBug when recommending similar bugs for a smaller system (Mylyn), with few bugs. The Mozilla ecosystem has an overwhelming amount of reported issues, while smaller and less popular open source projects may face a different scenario. Considering such scenario is important to verify if smaller projects can also benefit from similar bug recommendations.

For this study, we do not compare NextBug results with REP because the Mylyn dataset for this study does not have all the fields and information required by REP (e.g., product, priority, version, etc.).

5.2.1 Mylyn Dataset

Mylyn is a task management framework for Eclipse. For this study, we initially consider 2,682 issue reports fixed between January 2009 and December 2012. Figure 5.14 shows the monthly number of fixed issues in this time frame.

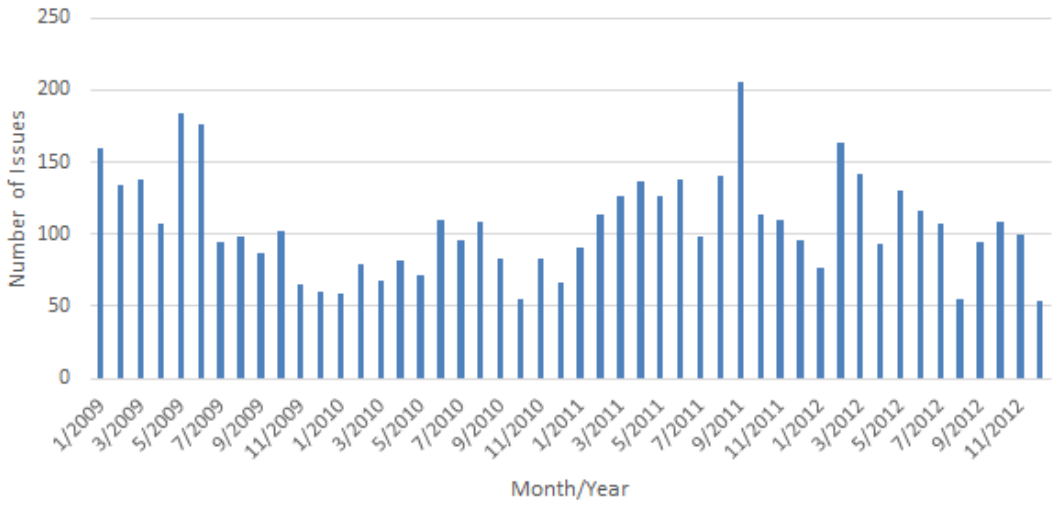
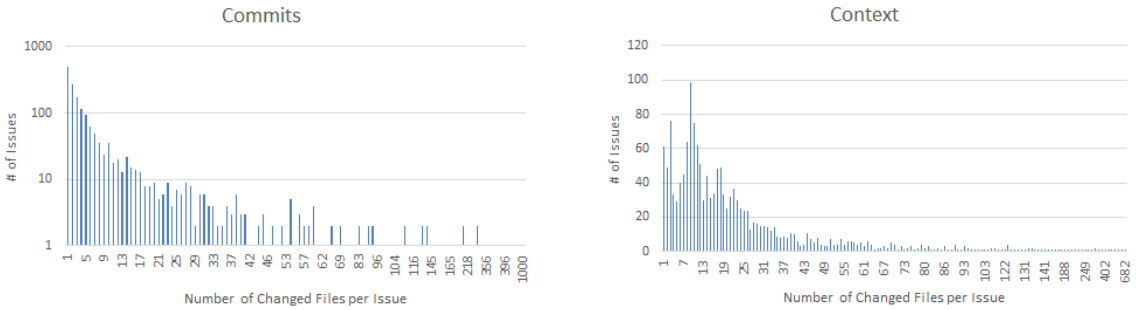


Figure 5.14: Fixed issues per month



(a) Commit (b) Context

Figure 5.15: Files changed/browsed per mapped issue

As we previously stated for the Mozilla data (Section 5.1.1), a recommendation is relevant if two issues changed similar files. For Mylyn, there are two sets of files that can be related to an issue. First, Mylyn has the files effectively changed when working on 1,756 issues, which we call the *Commit* dataset. Second, the system also provides information on the files browsed in Eclipse by Mylyn's developers when working on 1,573 issues, which we call the *Context* dataset. This last dataset was collected using the Mylyn plugin itself. Figure 5.15 shows a histogram with the frequency of files changed (Figure 5.15a for the *commits* dataset) or files browsed (Figure 5.15b for the *context* dataset). When considered together the *Commit* and *Context* datasets include 2,682 unique issues, distributed over 43 components.

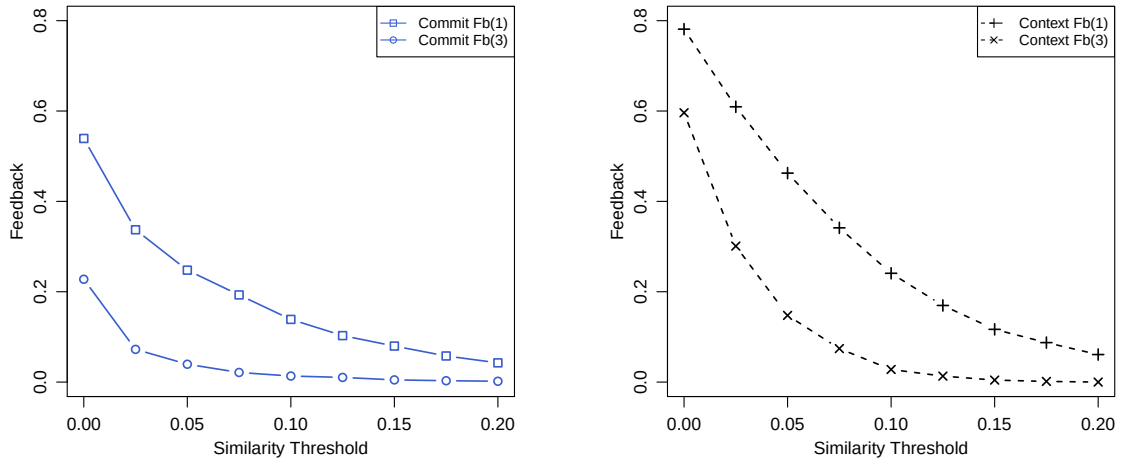
5.2.2 Study Design

For this study, we retrospectively simulated NextBug providing recommendations for the bugs in the Mylyn dataset. The design is similar to the comparative study (Section 5.1), but using a different dataset. Moreover, we present the results for both the *Commit* and *Context* datasets. For the evaluation we will also use the same metrics described in Section 5.1.4: feedback, precision, likelihood, recall, and f-score.

5.2.3 Results

Figures 5.16 to 5.20 show results for the evaluation metrics, for similarity thresholds ranging from 0.0 to 0.2 (τ parameter). We decided to analyze a lower threshold range than in the Comparative Study (Section 5.1) because Mylyn has fewer issues when compared to the Mozilla dataset (less than 5% of the analyzed Mozilla issues). For this reason, most metrics do not show results for higher thresholds.

Figure 5.16 shows the feedback results. The *Commit* dataset shows lower feedback values than *Context*. For instance, considering a similarity threshold of zero,



(a) Commit (b) Context

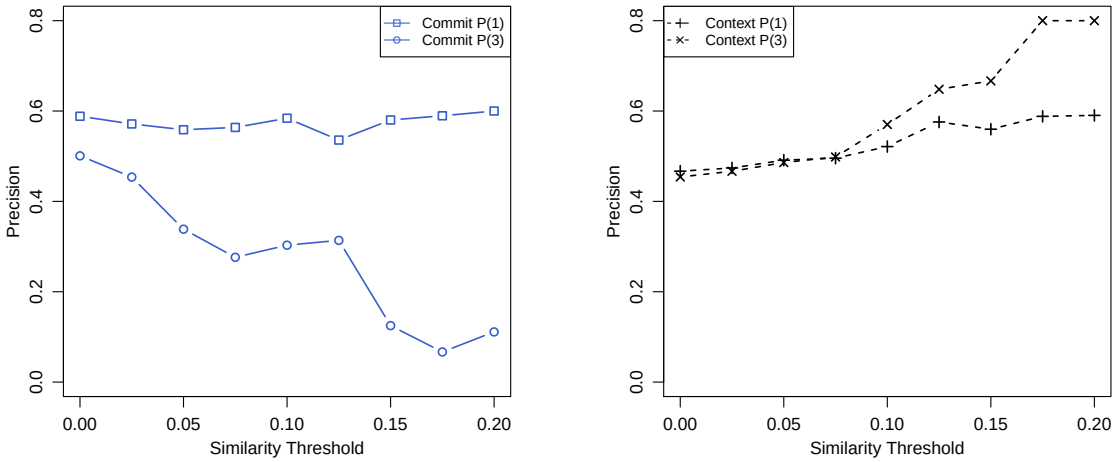
Figure 5.16: Feedback (Fb) results: (a) Commits, (b) Context

feedback(1) results are 54% and 78%, and feedback(3) results are 23% and 60%, respectively for *Commit* and *Context*. Moreover, if we compare to the Mozilla dataset, the feedback results for Mylyn *Commit* are very close for low thresholds. For example, NextBug showed feedback(1) = 68% and feedback(3) = 44% (similarity threshold of zero). This indicates that NextBug is able to give recommendations even when a system has few issues (such as Mylyn).

Considering precision (Figure 5.17), both *Commit* and *Context* present a slightly upward trend for top-1 recommendations. Regarding top-3 recommendations, *Commit* shows a downward trend while *Context* shows an upward trend. These trends contrast with the Mozilla dataset which showed a more stable precision values on thresholds of 0.2 and lower. For example, for precision(1) and precision(3), *Commit* results are 59% and 50%, and *Context* results are 47% and 45%, respectively. For Mozilla these results are 70% and 65% (considering a similarity threshold of zero). If we increase the threshold to 0.2, precision(1) and precision(3) results are 60% and 11% for *Commit*, 59% and 80% for *Context*, and 72% and 65% for Mozilla.

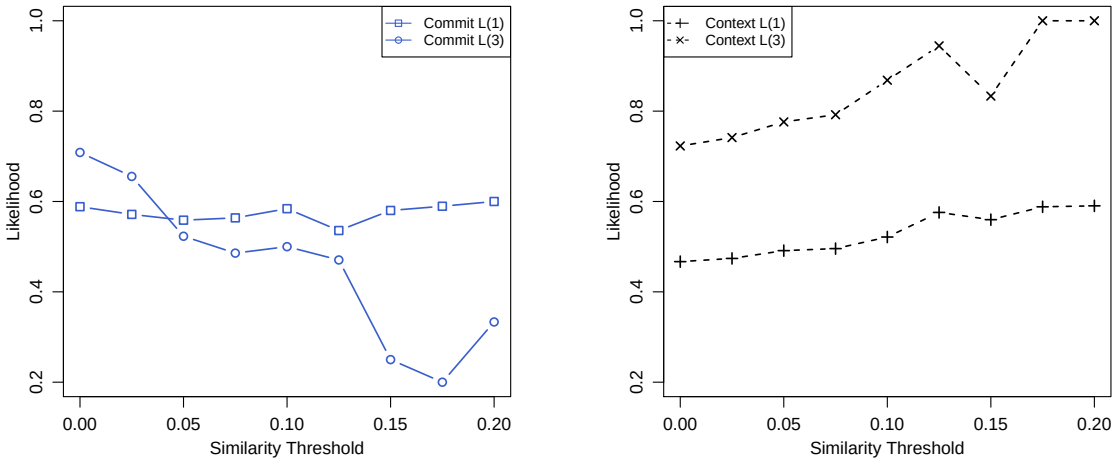
Regarding likelihood (Figure 5.18), the *Context* dataset performs again better than the *Commit*. Because likelihood(1) is equal to precision(1) by definition, we only discuss likelihood(3) results. For a threshold of zero, *Commit* likelihood(3) is 71%, and *Context* likelihood(3) is 72%. As the similarity threshold increases, *Commit* likelihood(3) decreases and *Context* likelihood(3) increases. For instance, considering a similarity threshold of 0.1, *Commit* and *Context* likelihood(3) results are 50% and 87%, respectively.

Figures 5.19 shows the recall and maximum possible recall measurements. Both



(a) Commit (b) Context

Figure 5.17: Precision (P) results: (a) Commits, (b) Context

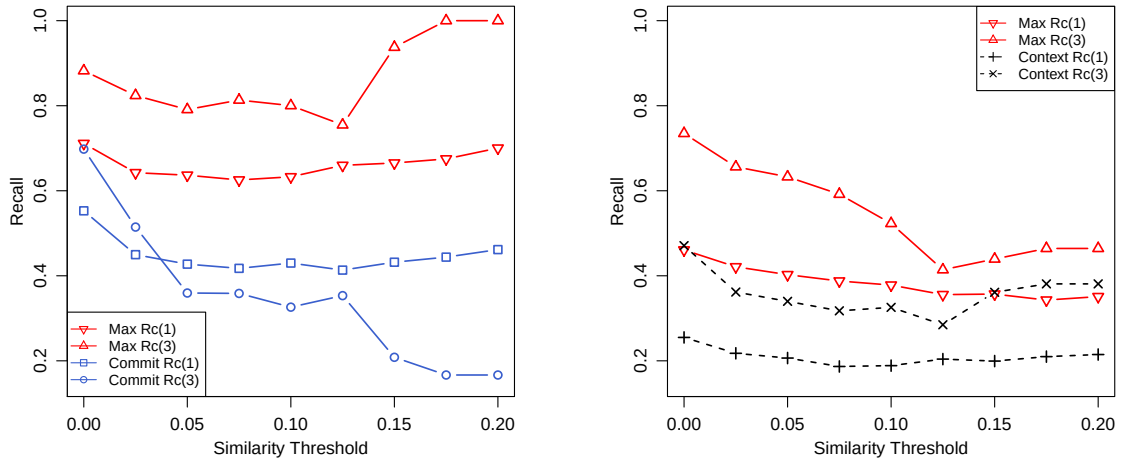


(a) Commit (b) Context

Figure 5.18: Likelihood (L) results: (a) Commits, (b) Context

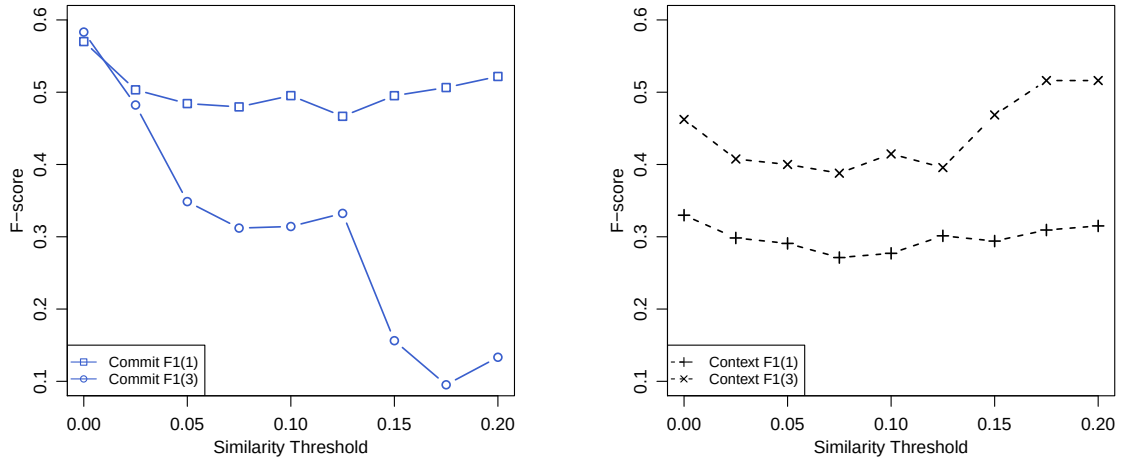
Commit and *Context* shows more slightly downward trend for recall results. When we compare the results, *Commit* recall are usually better than *Context*. For example, considering a threshold of zero, recall(1) and recall(3) results are respectively 55% and 70% for *Commit*, and 25% and 47% for *Context*. These results are expected because the Oracle for *Commit* is smaller than *Context*.

Considering the maximum recall, both *Commit* and *Context* show a downward trend for thresholds lower than 0.15. If we consider a similarity threshold of zero, max recall(1) values are 71% and 46% respectively for *Commit* and *Context*. For the same threshold, max recall(3) values are 88% and 73% respectively for *Commit* and *Context*.



(a) Commit (b) Context

Figure 5.19: Recall (Rc) results: (a) Commits, (b) Context



(a) Commit (b) Context

Figure 5.20: F-score (F_1) results: (a) Commits, (b) Context

Figures 5.20 shows the F-scores. Since *Commit* shows better results for both precision(1) and recall(1) than *Context*, its $F_1(1)$-scores are also higher than *Context*. For example, considering a threshold of zero, $F_1(1)$-scores are 57% and 33%, respectively for *Commit* and *Context*. Considering a threshold of zero, $F_1(3)$-scores are 58% *Commit* and 46% *Context*.

Summary: NextBug can provide similar bugs recommendations for small projects (such as Mylyn), once it is configured with low similarity thresholds.

5.3 Threats to Validity

Internal Validity: Initially, we assumed that the bugs recommendations are relevant if they share similar changed files with the query. There are at least three threats to this assumption. First, one study discuss that developers sometimes include extra files in their patches that are not necessarily changed [Walker et al., 2012]; these files are included in the patches to help understand the proposed changes. Therefore, we can consider these extra files as part of the bug fixing context, i.e., they do not in fact represent noisy data. Second, two bugs are considered similar even when they require changes in different parts of the same source code files. Therefore, handling these bugs sequentially might not necessarily save effort.

Regarding Mylyn, these threats are mitigated in two ways. First, the *Commit* dataset are the real files changed when working on Mylyn's issues (and not information extracted from patches). Second, the *Context* dataset includes the files accessed in the IDE when working on the issues, as captured by the Mylyn tool itself.

External Validity: The study reported in this chapter considered systems with different characteristics. Mozilla is a complex ecosystem, including well-known Internet software. Mylyn is a task management framework for the Eclipse ecosystem. Although the studied systems represent different cases, our results may not generalize to other systems and domains.

5.4 Final Remarks

In this chapter, we evaluated our approach, NextBug, in a quantitative manner by retrospectively computing recommendations over bug datasets. First, we compare NextBug with REP—a state of art technique for detecting duplicate bugs (Section 5.1). We employed five evaluation metrics design compare both techniques. The comparative study showed that, for similar bug recommendations, NextBug performs just as well as REP. Moreover, both techniques are feasible to be computed online by Issue Tracking Systems.

To conclude the chapter, we present a study using bugs from a smaller project, Mylyn (Section 5.2). The results showed that NextBug small systems with few reported bugs. We also discuss the threat to validity for the experiments conducted in this chapter (Section 5.3).

Chapter 6

Field Study

Retrospective studies take an optimistic view; any relevant recommendation will be recognized by the recipient of the recommendation. To address this issue, in this chapter we report a field study with Mozilla developers to determine if developers recognize relevant recommendations and if they would consider taking a recommendation. First, we present the study design for the experiment (Section 6.1). Second, we discuss the results and analyze the surveyed answers (Section 6.2). Then, we present the threats to validity for the field study (Section 6.3). Finally, we present the final remarks of this chapter (Section 6.4).

6.1 Study Design

For this study, we monitored bugs handled on Mozilla projects for a week (from May 22 to May 28, 2014). On each day, we collected data on bugs fixed in the previous day and computed off-line recommendations of similar bugs, as provided by NextBug. We used $\tau = 0.3$ (similarity threshold) because it showed an interesting balance between precision and recall in the comparative study results (Section 5.1.5). Additionally, it does not generate too many recommendations to manage manually.

We analyzed 1,412 bugs that were handled during the experiment's week and we were able to compute recommendations for 421 bugs, which corresponds to a feedback of 30% (using the feedback definition presented in Section 5.1.4). For each bug with at least one recommendation, we sent an email, using information from the tracking system, to the developer responsible for its resolution (i.e., the developer assigned to handle the bug).[1] Figure 6.1 presents one of the emails we sent. In each email, the

[1]More specifically, we did not selected which bugs to send. We send the top-3 recommendations as provided by NextBug to every developer in the study.

developer is presented with a list of up to three bugs and asked whether he/she would consider to work on one of them (Question #1). Each recommendation includes the ID and short description of the bug, as well as a link to its page at Bugzilla to facilitate the developer's inspection. We also asked the developer why (or why not) they would work on the recommendations (Question #1a and #1b). Finally, we asked the developers whether they would consider it useful to extend the issue tracking system (Bugzilla in this particularly case) with a list of similar bugs (Question #2).

Hello Mr./Ms. [XXX],

The following bug assigned to you was resolved on Tuesday:

[789261] - WebIDL bindings for Window

We found that you might next consider to work on one of the following open bugs:

I. [976307] - ES objects created by WebIDL bindings should be created in the compartment of the callee
II. [986455] - Support implementing part of C++ WebIDL interfaces in JS
III. [979835] - Port BoxObject to WebIDL

Looking at these suggestions, would you:

1. Consider working on one of these suggested bugs next?
1a. If so, which of the bugs would you select to work on?
1b. If not, why are these bugs not of interest to work on?

2. Consider useful a Bugzilla extension presenting bugs similar to a browsed one (i.e., bugs that would probably require changes similar to the ones performed when fixing a given bug)?

Figure 6.1: E-mail sent to a Mozilla developer

We sent only one email to any given developer, regardless of how many bugs he/she concluded in the week we studied. The intention was to avoid a perception of our mails as spam messages. We discarded the usage of a control group due to potential ethical issues. Using a control group would require sending meaningless recommendations to real developers, which could impact negatively their daily work and therefore contribute to a negative image of software engineering researchers among practitioners.

The study involved sending emails to 176 developers. We received 66 answers, which represents a response ratio of approximately 37%. We classify the developers that answered our survey according to the number of handled bugs. We employed the same skill classification based on the number of assigned bugs from our developers' profiles in the characterization study (Section 3.3.3). More specifically, we classified as *Newbies* developers who are assigned to work on three bugs or less; *Juniors* developers

who work on 22 bugs or less; and *Seniors* developers who work on more than 22 bugs. These groups correspond to 3% *Newbies*, 8% *Juniors*, and 89% *Seniors* of the developers we received answers.

6.2 Results

Table 6.1 summarizes the field study results. Both questions were initially proposed to receive a positive (yes) or negative (no) answer. However, in some cases the developers did not answer the questions (blank) or answered in a not clear way (unclear). The percentage of valid answers (yes or no) is 97% and 79% for Question #1 and Question #2, respectively. In the following subsections, we analyze the valid answers received for each question.

Table 6.1: Field study results

	Answer			
	Yes	**No**	**Blank**	**Unclear**
Question #1	39 (59%)	25 (38%)	–	2 (3%)
Question #2	44 (67%)	8 (12%)	10 (15%)	4 (6%)

6.2.1 Question #1: Would you consider working on one of these bugs next?

For this first question, 59% (39 out of 66 developers) answered they would indeed consider taking one of the recommendations. Moreover, 86% of these answers (57 out of 66 answers) were provided by *Senior* developers, which increases their confidence. Table 6.2 presents detailed information on two recommendations that received a positive feedback from Mozilla developers. We can observe that the cosine similarity between the query and the recommendations is high (greater than 0.33) and the bugs seems to be semantically related to the queries. For example, the first query and associated recommendations denote problems in the library for Firefox marketplace payments. The second query and recommendations are related to the airplane mode feature, from Firefox OS. The developer who worked on the second query answered that he replied to Rec. #2.1 thanks to our email.

For the bugs in Table 6.2, we received these comments:

"The suggestions seem pretty accurate (...) the #3 bug is new to me — I didn't know about that one" (Subject #47 on Recs. for Query #1)

Table 6.2: Examples of useful recommendations, according to Mozilla developers (including the textual similarity between the query and the provided recommendations)

	Short Description	**Sim.**
Query #1	fxpay: make a payment with the example app (ID 989136)	
Rec. #1.1	fxpay: save a receipt to device on purchase (ID 991994)	0.35
Rec. #1.2	fxpay: fixup transaction state (ID 987758)	0.34
Rec. #1.3	'Payment Cancelled' message is displayed after 'Payment Complete' message (ID 972108)	0.33
Query #2	Airplane mode icon in status bar is not responsive (ID 1014262)	
Rec. #2.1	Airplane mode icon can co-exist with the wifi icon (ID 1008945)	0.49
Rec. #2.2	Intermittently the user is unable to turn/off Airplane mode on (ID 1003528)	0.48
Rec. #2.3	Airplane mode does not display an 'on' message to user (ID 1010551)	0.39

"The suggested bugs you present are definitely bugs I'd be likely to pick up after the one I solved" (Subject #40 on Recs. for Query #2)

Table 6.3 presents two examples of incorrect recommendations. In the first example, both the query and the recommendation are related to a specific component, called *Compositor*. However, the query denotes a performance bug and the recommendation denotes a crash. In the second example, Query #4 denotes a fairly simple bug in the JavaScript engine and Rec. #4.1 is a more complex but not critical bug, according to the developer who completed the query.

Table 6.3: Examples of incorrect recommendations, according to Mozilla developers (including the textual similarity between the query and the provided recommendations)

	Short Description	**Sim.**
Query #3	Add compositor benchmark (ID 1014042)	
Rec. #3.1	Compositor crash during shutdown crash while debugging (ID 977641)	0.36
Query #4	Use a magic number to identify crashes related to any stack traversal during bailouts (ID 1015145)	
Rec. #4.1	IonMonkey bailouts should forward bailout reason string to bailout handler (ID 1015323)	0.35

We also analyzed the comments related to the incorrect answers, trying to extract small phrases and sentences that can contribute to organize the developers' reasons in categories. This analysis resulted in four categories, as follows:

- Recommendations that do not make sense (e.g., *"one bug is a performance bug, the other is a stability bug", Subject #9*).
- Recommendations that might be correct, but they are not a priority at this moment (e.g., *"bugs may interest me but are definitely not my current focus", Subject #1*).
- Recommendations to developers who are not an expert on the component (e.g., *"I fixed that bug because someone broke the build, I'm not interested in the sandbox", Subject #38*).
- Recommendations to paid Mozilla developers, who follow a work schedule (e.g., *"Our manager determines the next bugs we work on", Subject #42*).

The percentage of answers in each category is presented in Table 6.4. Only 20% of the recommendations were ranked as denoting a non-similar or unrelated bug. In fact, 40% of the negative answers are not exactly because the recommendations are meaningless, but because the developers decided to focus on another bug, which he/she judged as having a higher priority. Moreover, 16% of the developers confirmed they did not have the expertise to evaluate if the recommendation were relevant or not. Only 12% of the developers stated they follow a work schedule. This is expected since most Mozilla contributors are volunteers with the liberty to choose their own work.

Table 6.4: Reasons for not following a recommendation

Recommendations that do not make sense	20%
Recommendations that might be correct, but are not a priority	40%
Recommendations to non-expert developers	16%
Recommendations to developers with a work schedule	12%
Other reasons	12%

6.2.2 Question #2: Would you consider useful a Bugzilla extension with recommendations?

For this second question, 44 developers (67%) answered that a Bugzilla extension including similar bug recommendations would be useful. All *Newbie* and *Junior* developers answered this second question positively. Among the *Senior*, 82% answered the

question positively. We analyzed the comments related to the positive answers which resulted in the following categories:

- The extension is specially good for new contributors (e.g., *"would be immensely helpful for new contributors as they don't know the project very well", Subject #10*).
- The extension would increase developers' productivity (e.g., *"we could do more work in less time", Subject #27*).
- The extension would be useful as a customized bug search engine (e.g., *"might be useful for keeping relevant bugs that I might miss on my radar", Subject #39*).

The percentage of answers in each category is presented in Table 6.5. We can observe that 11 developers (25%) highlighted the benefits of the recommendations to new Mozilla contributors. Other 14% of the answers mention the increase in productivity and 13% envisioned using NextBug as an advanced bug search engine.

Table 6.5: Usefulness of a Bugzilla extension

Support new contributors when searching for bugs	25%
Increase productivity	14%
Support to customized bug searches	13%
No reason given	25%
Other reasons	23%

Only 8 developers (12%) answered that a Bugzilla extension including bug recommendations would *not* be useful (all of them are *Senior* developers). We organized the negative answers in two categories:

- The extension is not useful for developers who follow a well defined work schedule (e.g., *"in my case a product process is driving the prioritization", Subject #48*).
- The extension is not useful for developers that work on projects with few and usually well-known bugs (e.g., *"I don't manage a lot of open bugs at the same time, so I don't need a list of suggestions", Subject #22*).

Each of these categories received 25% of the answers. Furthermore, 50% of the respondents did not give a clear reason or did not provide an answer at all.

6.3 Threats to Validity

Internal Validity: We based the study on a single textual similarity threshold. Differently from the retrospective study, we did not test other thresholds, which would be a particularly difficult task in a study with real developers. However, we at least tried to select a threshold showing an interesting balance between precision and feedback. We did not send multiple mails to the same developer, to avoid a perception of our messages as spam. Due to this decision, we in fact reduced our sample to 42% of the bugs with recommendations completed in the week monitored in the study. Despite this fact, we were able to receive feedback from 66 unique developers, which is a good response rate (≈37%).

External Validity: The participants might not be representative of the whole population of Mozilla developers and, in more general terms, of general software developers.

6.4 Final Remarks

In this chapter, we presented a qualitative study by surveying real developers about similar bugs recommendations. We received 66 responses (37%) from unique developers working on Mozilla bugs. First, we described how the experiment was designed and conducted, the survey questions, and the experimental subjects (Section 6.1). Then, we presented an overview of the answers and we discussed the comments we received (Section 6.2). For instance, 59% of the developers considered taking a recommendation provided by NextBug, and 67% of the developers considered useful a Mozilla extension to recommend similar bugs. Finally, we presented the threats to validity (Section 6.3).

Chapter 7

Conclusion

In this chapter we present our closing points and arguments. We begin with a discussion on our main findings and some possible questions (Section 7.1). Then, we describe the main contributions from our research (Section 7.2). Finally, we outline possible ideas for future work (Section 7.3).

7.1 Discussion

In this section, we discuss and put our findings in perspective. We start by discussing why it is not always possible to avoid context changes (Section 7.1.1). We also discuss the increase on productivity possible by reordering tasks (Section 7.1.2), the systems that most benefit of task recommendations (Section 7.1.3), and alternatives to improve our evaluation metrics (Section 7.1.4).

7.1.1 Can we avoid context changes on issue handling?

It is not possible to completely avoid context changes because a similar pending issue may not exist in the tracking system at the time a given issue is concluded. In such cases, a developer who wishes to continue working on the system must choose a bug requiring a different context. For example, in the retrospective study, NextBug achieved a feedback of 78% and a top-1 precision of 47% for Mylyn's *Context* dataset (considering a similarity threshold of zero). Therefore, we can infer that $1-(0.78*0.47) = 63\%$ of the issues do not have a similar pending issue in the tracking system (at least, as predicted by our approach). For Mozilla, we achieved a feedback of 68% and a top-1 precision of 70%. Therefore, for $1 - (0.68 * 0.70) = 48\%$ of the bugs a context change is required.

For both cases we assume only a top-1 recommendation list to be conservative. If we consider top-3 recommendations the percentage of context change would be lower.

Based on these results, one can argue that NextBug is effective for a portion of issues. However, for systems with an intense flow of new issues, a positive impact on approximately 35-50% of the issues is relevant. For example, our recommendations can avoid context changes for 67K bugs (52% of 130,495) in Mozilla and 992 bugs (37% of 2,682) in the Mylyn plugin. Moreover, these recommendations are provided at low cost in feasible execution time (3.8 ms on average). Finally, they are reported to be of interest to end-users. For example, 67% of the Mozilla developers in the field study indicated interest in a Bugzilla extension with recommendations.

7.1.2 Does our approach increase developer's productivity?

In the field study, six developers spontaneously mentioned that a Bugzilla extension with recommendations would contribute to an increase in productivity. We received answers like this one:

"It would be useful for developers, since they'll be able to be much more productive in their contributions", (Subject #7)

Additionally, issue recommendations can keep contributors working on the system, by promptly indicating new issues they could work on. In this way, gains of productivity may be achieved by expanding the workforce, which is especially important for the long-term survival of complex open-source systems [Mockus et al., 2002]. For example, 2,221 developers were responsible for the 130,495 issues we initially considered in the retrospective study with Mozilla data. However, 928 developers (42%) worked on at most two issues and did not return to work on the system. It is therefore reasonable to assume that recommendations could help to keep such developers involved with the project. In fact, 11 developers mentioned this fact in the field study, as in the following answer:

"A recommendation engine would be immensely helpful for new contributors to the project as they often don't know the project very well and find skulking through the bug tracker tedious and energy-sucking.", (Subject #10)

The previous comment reinforces the finding revealed in the characterization study, where less skilled developers (i.e., *Newbies* and *Juniors*) take more time to be assigned to a bug than actually fixing it (Section 3.4.4). In summary, tools like

NextBug, can help new contributors to find and, consequently, handle more bugs.

In this thesis, our experiments (Chapters 5 and 6) showed indirect evidence that NextBug can help to increase productivity. However, we acknowledge we did not conduct specific experiments to measure the productivity gains achieved from NextBug. These experiments are left as a future line of work.

7.1.3 Applicability

The benefits of the approach discussed in this thesis tend to be clearer in systems with a large base of maintenance tasks. For example, in 2011, on average 6,593 maintenance issues were reported monthly for Mozilla projects and on average 3,043 valid issues were waiting to be handled in the first day of each month. On the other hand, on systems with limited maintenance activity the advantages of an automated approach for recommending similar bugs are less clear. First, because the space for searching for similar tasks is naturally smaller. In other words, less tasks means less similar tasks and therefore less effort saved on avoiding context changes. Second, if a system has few pending tasks, it is easier for developers to discover by themselves the next tasks they should work on, without the support of a recommendation engine.

We evaluated NextBug with 70 open-source systems, including 69 systems from Mozilla and with the Mylyn plugin from the Eclipse ecosystem. Open-source systems are natural candidates for task recommendations due to the uncoordinated and decentralized nature of their development process, which depends on a large base of contributors, most of them working as volunteers. Therefore, as previously discussed in Section 7.1.2, such volunteers constitute an important target of the recommendations proposed in the thesis. On the other hand, even systems like Mozilla include paid developers, who have a well-defined workflow established by project managers. In the field study, these developers usually ranked our recommendations as not useful, as stated in the following comment:

*"I'm a paid contributor to Mozilla, and as such, we have processes used by ourselves and our manager which determine the next bugs we work on. So while your approach looks like it did indeed select bugs somewhat similar to the one you mentioned, the process used to choose which bug to *actually* work on next is quite different than simply "it is similar". Your approach may work well for volunteers who have full control over exactly what they choose to do.", (Subject #42)*

However, even in such cases we envision that task recommendations might have an application, not to developers, but to support the project managers when creating

and assigning working units to paid developers. For example, recommendations can be used to complement these working units with related tasks that could be easily implemented by the assigned developers and that were not initially considered by the managers responsible for their creation. A similar scenario applies to closed systems, where the workforce typically follows a well-defined task schedule.

We also received suggestions of slightly different applications of NextBug. For example, one of the Mozilla's project manager commented that NextBug could be adapted to recommend mentors for pending bugs:

"To be able to say, based on [a previous] contribution, you would be a good mentor for these bugs over here, that would be quite valuable to us.", (Mozilla's Project Manager)

Another surveyed subject suggested that we could apply our approach to find bugs that are appropriate to beginner contributors:

"I spend some time looking through the bugs and marking them as good first bugs for contributors, but that requires a lot of work. An engine would probably be easier, plus it might connect bugs it didn't occur to me to be connected.", (Subject #10)

7.1.4 Can we improve the measured evaluation results?

In this section, we discuss whether it is possible to improve results, considering the evaluation metrics used in the retrospective study (Chapter 5). We employed four metrics proposed to evaluate recommendation systems (feedback, likelihood, precision, and recall) [Zimmermann et al., 2004] and a f-score (a harmonic mean between precision and recall). On NextBug, we use only two fields from issue reports: short description and component. At first glance, it may seem that our results can be improved by considering other fields, such as priority, severity, full description, etc. In fact, work on duplicated bug reports followed this strategy over the years to present better results. However, in the comparative study (Section 5.1) we used REP [Sun et al., 2011], a state-of-art technique of duplicate bugs detection, to find similar bugs. Even tough REP uses several fields (e.g., full description, severity, etc.) to compute similarity among bugs, its results are just as good as the ones achieved with NextBug. Therefore, by just adding more fields to the similarity computation does not necessary impact on the evaluated results.

We also argue that the use of a thesaurus with common and domain-specific words can contribute to increase the evaluation metrics. By using a thesaurus, our

approach could detect similar bugs using related semantic words that otherwise would be undetected, and consequently, recommend bugs more accurately.

Another possibility of improvement would be to employ a clustering algorithm and recommend the bugs from the same cluster as similar. However, we claim this technique is more appropriate to create maintenance projects. In such case, each cluster would be a maintenance project composed of similar issues. Nevertheless, to apply clustering on-the-fly in the issue tracking system to give recommendations, requires a cluster algorithm with a extremely fast computing time. Whether or not clustering would improve our results, it is still an open question (and a future work idea).

7.2 Contributions

Popular open source systems may face a high number of reported issues. When developers work on issues that touch different parts of a software system, they must spend time between each task handling a context change, i.e., they must spend time finding the appropriate code and understanding it. In this thesis, to reduce the number and costs of context changes, we proposed a new approach, called NextBug, to identify and to organize a set of similar programming tasks using a recommendation mechanism. The recommendations provided by NextBug can help a developer to work in a more productive manner.

First, we conducted a characterization study focusing on the workflow followed by developers to handle bug reports to better understand the bug handling process. To help on this characterization, we propose the concept of Bug Flow Graphs (BFG), which provides a visual representation of the overall workflow process in a more simple way than retrieving this information from the issue tracking system. The BFG for all Mozilla bugs showed that a bug usually takes five days longer to be resolved when it is not properly assigned to a developer. When analyzing the developers' profile BFGs, we discovered that less skilled Mozilla developers take a longer time to be assigned to a bug than actually fixing it. Therefore, tools and techniques to help developers to find more bugs can improve the productivity of the bug handling process.

We performed a comparative study to verify whether NextBug is more suited for similar bug recommendations than REP (a state-of-the-art technique for duplicate bug retrieval). Although REP provides more recommendations, NextBug shows comparable results for the remaining metrics. On Mozilla bugs, NextBug precision for top-1 recommendation is 70%, and REP is 71%. Considering likelihood, for the top-3 recommendations, the values are 82% and 81% for NextBug and REP, respectively. These

results indicate that for detecting similar bugs a technique that considers just the bug components and short descriptions (like NextBug) performs as well as a technique optimized for identifying identical bug reports, which usually considers other fields and compute their recommendations using complex ranking functions.

We also conducted a retrospective study on the Mylyn framework to verify NextBug applicability in a system with a smaller dataset of issues. The results show that NextBug can effectively provide recommendations even for smaller systems.

Finally, we performed a week-long field study by sending e-mails with recommended bugs produced by NextBug to Mozilla developers. We received a feedback of 66 developers (37%) and most of them (59%) confirmed that NextBug recommendations express meaningful next bugs for developers to work on. Moreover, most of the surveyed subjects (67%) confirmed interest in an extension to recommend similar bugs on their issue tracking system (Bugzilla).

7.3 Future Work

Future work includes an investigation on alternatives to improve NextBug results, like building a thesaurus with common words and also with domain-specific words which can contribute to increase NextBug accuracy. Another possibility to improve our approach is to employ clustering algorithms and verify their efficacy for similar bug recommendations. Moreover, we plan to implement and compare other close related techniques (proposed for example to duplicate bugs or to assign bugs to developers).

We also plan to investigate proactive recommendation styles, e.g., notifying developers when a new bug similar to one he/she is currently working on appear in the tracking platform. Another route for future work is to create user profiles to store and identify the developer's behavior when handling bugs, and therefore, recommend bugs based on his personal preferences. We are planning a second field study, when we intend to install NextBug in a real bug tracking system to investigate whether developers really follow similar bug recommendations. Regarding the characterization study, we have plans to implement an online tool to create BFGs (Bug Flow Graphs), which can be used by project managers to reason about bug fixing workflows.

Another line of future work is to perform experiments to measure the gains of productivity achieved by NextBug. Although, we did show indirect evidence that NextBug can increase the productivity, these new experiments can provide more robust evidences about this benefit.

Bibliography

Ahn, Y., Suh, J., Kim, S., and Kim, H. (2003). The software maintenance project effort estimation model based on function points. *Journal of Software Maintenance*, 15(2):71--85.

Alipour, A., Hindle, A., and Stroulia, E. (2013). A contextual approach towards more accurate duplicate bug report detection. In *10th Working Conference on Mining Software Repositories (MSR)*, pages 183--192.

Alkhatib, G. (1992). The maintenance problem of application software: an empirical analysis. *Journal of Software Maintenance*, 4(2):83--104.

Anvik, J., Hiew, L., and Murphy, G. C. (2006). Who should fix this bug? In *28th International Conference on Software engineering (ICSE)*, pages 361--370.

Anvik, J. and Murphy, G. C. (2011). Reducing the effort of bug report triage: recommenders for development-oriented decisions. *ACM Transactions on Software Engineering Methodology (TOSEM)*, 20(3):10:1--10:35.

Aziz, J., Ahmed, F., and Laghari, M. (2009). Empirical analysis of team and application size on software maintenance and support activities. In *1st International Conference on Information Management and Engineering (ICIME)*, pages 47–51.

Baeza-Yates, R. A. and Ribeiro-Neto, B. (1999). *Modern information retrieval*. Addison-Wesley, 2nd edition.

Banker, R. D. and Slaughter, S. A. (1997). A field study of scale economies in software maintenance. *Management Science*, 43:1709--1725.

Bugzilla Team (2015). *Bugzilla Documentation - 5.0.2+ Release*. Mozilla Foundation.

Cavalcanti, Y. C., Mota Silveira Neto, P. A., Lucrédio, D., Vale, T., Almeida, E. S., and Lemos Meira, S. R. (2013). The bug report duplication problem: an exploratory study. *Software Quality Journal*, 21(1):39–66.

Chambers, J., Cleveland, W., Kleiner, B., and Tukey, P. (1983). *Graphical Methods for Data Analysis*. Wadsworth & Brooks/Cole statistics/probability series. Chapman & Hall Boston.

Chan, T., Chung, S.-L., and Ho, T. H. (1996). An economic model to estimate software rewriting and replacement times. *IEEE Transactions on Software Engineering*, 22(8):580--598.

Coleman, D., Ash, D., Lowther, B., and Oman, P. (1994). Using metrics to evaluate software system maintainability. *Computer*, 27(8):44--49.

Dal Sassc, T. and Lanza, M. (2013). A closer look at bugs. In *1st IEEE Working Conference on Software Visualization (VISSOFT)*, pages 1–4.

D'Ambros, M., Lanza, M., and Pinzger, M. (2007). A Bug's life Visualizing a bug database. In *4th IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*, pages 113--120.

Erlikh, L. (2000). Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17--23.

Guo, P. J., Zimmermann, T., Nagappan, N., and Murphy, B. (2010). Characterizing and predicting which bugs get fixed: An empirical study of Microsoft Windows. In *32th International Conference on Software Engineering (ICSE)*, pages 495--504.

Heales, J. (2002). A model of factors affecting an information system's change in state. *Journal of Software Maintenance*, 14(6):409--427.

Holmes, R. and Murphy, G. C. (2005). Using structural context to recommend source code examples. In *27th International Conference on Software engineering (ICSE)*, pages 117--125.

Holmes, R., Walker, R. J., and Murphy, G. C. (2005). Strathcona example recommendation tool. In *13th Symposium on Foundations of Software Engineering (FSE)*, pages 237--240.

Ihara, A., Ohira, M., and Matsumoto, K. (2009). An analysis method for improving a bug modification process in open source software development. In *7th Joint International Workshop Principles of Software Evolution and Software Evolution (IWPSE-Evol)*, pages 135--144.

ISO/IEC 12207 (2008). *Systems and software engineering – Software life cycle processes*. International Organization for Standardization, 2 edition.

ISO/IEC 14764 (2006). *Software engineering – Software life cycle processes – Maintenance*. International Organization for Standardization, 2 edition.

Jalbert, N. and Weimer, W. (2008). Automated duplicate detection for bug tracking systems. In *38th International Conference on Dependable Systems and Networks (DSN)*, pages 52--61.

Jeong, G., Kim, S., and Zimmermann, T. (2009). Improving bug triage with bug tossing graphs. In *7th ACM SIGSOFT Symposium on The Foundations of Software Engineering (FSE)*, pages 111--120, New York, NY, USA. ACM.

Joorabchi, M. E., Mirzaaghaei, M., and Mesbah, A. (2014). Works for me! characterizing non-reproducible bug reports. In *11th Working Conference on Mining Software Repositories (MSR)*, pages 62--71.

Junio, G. A., Malta, M. N., Mossri, H., Marques-Neto, H. T., and Valente, M. T. (2011). On the benefits of planning and grouping software maintenance requests. In *15th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 55–64.

Kagdi, H., Gethers, M., Poshyvanyk, D., and Hammad, M. (2012). Assigning change requests to software developers. *Journal of Software: Evolution and Process*, 24(1):3--33.

Kagdi, H., Hammad, M., and Maletic, J. (2008). Who can help me with this source code change? In *24th IEEE International Conference on Software Maintenance (ICSM)*, pages 157--166.

Kasi, B. K. and Sarma, A. (2013). Cassandra: proactive conflict minimization through optimized task scheduling. In *35th International Conference on Software Engineering (ICSE)*, pages 732–741.

Kersten, M. and Murphy, G. C. (2006). Using task context to improve programmer productivity. In *14th Symposium on Foundations of Software Engineering (FSE)*, pages 1--11.

Ko, A. J., Aung, H., and Myers, B. A. (2005). Eliciting design requirements for maintenance-oriented IDEs: A detailed study of corrective and perfective mainte-

nance tasks. In *27th International Conference on Software Engineering (ICSE)*, pages 126--135.

Ko, A. J., Myers, B. A., and Chau, D. H. (2006). A linguistic analysis of how people describe software problems. In *17th IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 127--134.

Lehman, M. M. and Belady, L. A., editors (1985). *Program evolution: processes of software change*. Academic Press Professional, San Diego, CA, USA.

Lehman, M. M. and Ramil, J. F. (2001). Rules and tools for software evolution planning and management. *Ann. Softw. Eng.*, 11(1):15--44.

Liu, K., Tan, H. B. K., and Chandramohan, M. (2012). Has this bug been reported? In *20th International Symposium on the Foundations of Software Engineering (FSE)*, pages 28:1--28:4.

Luijten, B., Visser, J., and Zaidman, A. (2010). Assessment of issue handling efficiency. In *7th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 94–97.

Marcus, A. and Menzies, T. (2010). Software is data too. In *18th Workshop on Future of Software Engineering Research (FoSER)*, pages 229--232.

Marques-Neto, H., Aparecido, G. J., and Valente, M. T. (2013). A quantitative approach for evaluating software maintenance services. In *28th Annual ACM Symposium on Applied Computing (SAC)*, pages 1068--1073.

Meyer, A. N., Fritz, T., Murphy, G. C., and Zimmermann, T. (2014). Software developers' perceptions of productivity. In *22th International Symposium on Foundations of Software Engineering (FSE)*, pages 26--36.

Minelli, R. and Lanza, M. (2013). Visualizing the workflow of developers. In *1st IEEE Working Conference on Software Visualization (VISSOFT)*, pages 1–4.

Mockus, A., Fielding, R. T., and Herbsleb, J. D. (2002). Two case studies of open source software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology*, 11(3):309--346.

Mookerjee, R. (2005). Maintaining enterprise software applications. *Communications of the ACM*, 48(11):75--79.

Murphy, G. C. (2014). Getting to flow in software development. In *1st ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward!)*, pages 269--281, New York, NY, USA. ACM.

Naguib, H., Narayan, N., Brugge, B., and Helal, D. (2013). Bug report assignee recommendation using activity profiles. In *10th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 22–30.

Nguyen, A. T., Nguyen, T. T., Nguyen, T. N., Lo, D., and Sun, C. (2012). Duplicate bug report detection with a combination of information retrieval and topic modelling. In *27th International Conference on Automated Software Engineering (ASE)*, pages 70--79.

Parnin, C. and Rugaber, S. (2011). Resumption strategies for interrupted programming tasks. *Software Quality Journal*, 19(1):5--34.

Poshyvanyk, D., Dang, H., Hossen, K., Kagdi, H., Gethers, M., and Linares-Vasquez, M. (2012). Triaging incoming change requests: bug or commit history, or code authorship? In *28th IEEE International Conference on Software Maintenance (ICSM)*, pages 451--460. IEEE Computer Society.

Raghavan, V. V. and Wong, S. K. M. (1986). A critical analysis of vector space model for information retrieval. *Journal of the American Society for Information Science*, 37(5):279--287.

Ricci, F., Rokach, L., Shapira, B., and Kantor, P. B., editors (2011). *Recommender systems handbook*. Springer.

Rijsbergen, C. J. V. (1979). *Information retrieval*. Butterworth-Heinemann, 2nd edition.

Robertson, S., Zaragoza, H., and Taylor, M. (2004). Simple BM25 extension to multiple weighted fields. In *13th ACM International Conference on Information and Knowledge Management (CIKM)*, pages 42--49.

Robillard, M., Walker, R., and Zimmermann, T. (2010). Recommendation systems for software engineering. *IEEE Software*, 27(4):80–86.

Rocha, H., Maques-Neto, H., and Valente, M. T. (2013). Agrupamento automático de solicitações de manutenção. In *1a Escola Latino Americana de Engenharia de Software (ELA-ES)*, pages 1--1.

Rocha, H., Oliveira, G., Maques-Neto, H., and Valente, M. T. (2014). Nextbug: A tool for recommending similar bugs in open-source systems. In *5th Brazilian Conference on Software: Theory and Practice (CBSoft), Tools Demonstration Track*, pages 1--6.

Rocha, H., Oliveira, G., Marques-Neto, H., and Valente, M. (2015). NextBug: a Bugzilla extension for recommending similar bugs. *Journal of Software Engineering Research and Development (JSERD)*, 3(1):14.

Rocha, H., Valente, M. T., Maques-Neto, H., and Murphy, G. C. (2016). An empirical study on recommendations of similar bugs. In *23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 1--11.

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

Shani, G. and Gunawardana, A. (2011). Evaluating recommendation systems. In *Recommender Systems Handbook*, pages 257--297.

Shokripour, R., Anvik, J., Kasirun, Z. M., and Zamani, S. (2013). Why so complicated? simple term filtering and weighting for location-based bug report assignment recommendation. In *10th Working Conference on Mining Software Repositories (MSR)*, pages 2--11.

Sun, C., Lo, D., Khoo, S.-C., and Jiang, J. (2011). Towards more accurate retrieval of duplicate bug reports. In *26th International Conference on Automated Software Engineering (ASE)*, pages 253--262.

Sun, C., Lo, D., Wang, X., Jiang, J., and Khoo, S.-C. (2010). A discriminative model approach for accurate duplicate bug report retrieval. In *32nd International Conference on Software Engineering (ICSE)*, pages 45--54.

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

Swanson, E. B. and Beath, C. M. (1989). *Maintaining information systems in organizations*. Wiley & Sons, New York, NY, USA.

Tamrawi, A., Nguyen, T. T., Al-Kofahi, J. M., and Nguyen, T. N. (2011). Fuzzy set and cache-based approach for bug triaging. In *13th Conference on Foundations of Software Engineering (FSE)*, pages 365--375.

Tan, Y. and Mookerjee, V. (2005). Comparing uniform and flexible policies for software maintenance and replacement. *IEEE Transactions on Software Engineering*, 31(3):238--255.

Tian, Y., Sun, C., and Lo, D. (2012). Improved duplicate bug report identification. In *16th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 385–390.

Walker, R. J., Rawal, S., and Sillito, J. (2012). Do crosscutting concerns cause modularity problems? In *20th Symposium on the Foundations of Software Engineering (FSE)*, pages 1--11.

Wang, D., Lin, M., Zhang, H., and Hu, H. (2010). Detect related bugs from source code using bug information. In *34th IEEE Computer Software and Applications Conference (COMPSAC)*, pages 228–237.

Wang, X., Zhang, L., Xie, T., Anvik, J., and Sun, J. (2008). An approach to detecting duplicate bug reports using natural language and execution information. In *30th International Conference on Software Engineering (ICSE)*, pages 461--470.

Weiß, C., Premraj, R., Zimmermann, T., and Zeller, A. (2007). How long will it take to fix this bug? In *4th Workshop on Mining Software Repositories (MSR)*, pages 1–8.

Zimmermann, T., Nagappan, N., Guo, P. J., and Murphy, B. (2012). Characterizing and predicting which bugs get reopened. In *34th International Conference on Software Engineering (ICSE)*, pages 1074--1083.

Zimmermann, T., Weisgerber, P., Diehl, S., and Zeller, A. (2004). Mining version histories to guide software changes. In *26th International Conference on Software Engineering (ICSE)*, pages 563--572.

Appendix A

Field Study Answers

In this appendix we supply the summarized field study answers. To preserve the subject's anonymity, we withhold personal information from our surveyed subject as well as any other data that could be used to track them. Table A.1 show the subject's number, the date we sent the survey email, the bug ids recommended to them, and the yes/no answers to the surveyed questions.

Table A.1: Survey Answers

Subject	**Survey Date**	**Recommendations**			**Answers**	
		I	**II**	**III**	**Q#1**	**Q#2**
1	2014-05-22	956294			N	Y
2	2014-05-22	979946	957830		Y	Y
3	2014-05-22	971044	976571		unclear	Y
4	2014-05-22	973133			N	Y
5	2014-05-22	965435	965454		Y	Y
6	2014-05-22	967621			N	Y
7	2014-05-22	977245	977178	978220	Y	Y
8	2014-05-22	983930			N	Y
9	2014-05-22	977641			N	Y
10	2014-05-22	975141			Y	Y
11	2014-05-22	956159	984865		Y	
12	2014-05-23	1007772	999091	1005296	Y	unclear
13	2014-05-23	1015208			N	Y
14	2014-05-23	1015205	1015144		Y	Y
15	2014-05-23	1014242			Y	N
16	2014-05-23	983041	957046		N	

Continued on next page ⇓

Subject	Survey Date	Recommendations			Answers	
		I	II	III	Q#1	Q#2
17	2014-05-23	1003570			Y	Y
18	2014-05-23	979835			unclear	unclear
19	2014-05-23	960462	980066	1007117	N	Y
20	2014-05-23	1012462	988366		Y	Y
21	2014-05-23	1015162	1007336	1015157	Y	Y
22	2014-05-26	967893	978296	978298	Y	N
23	2014-05-26	999778	999779	994520	Y	Y
24	2014-05-26	982019			Y	Y
25	2014-05-26	1011831	1014932		N	Y
26	2014-05-26	1001540	989889	979158	Y	N
27	2014-05-26	1015688	991553	989292	Y	Y
28	2014-05-26	979888			N	
29	2014-05-26	980922			N	Y
30	2014-05-26	973503			Y	Y
31	2014-05-26	983018			Y	Y
32	2014-05-26	1011708			N	
33	2014-05-27	1011019			Y	N
34	2014-05-27	1016577			Y	Y
35	2014-05-27	993555	994410	973732	Y	Y
36	2014-05-27	1012985			Y	N
37	2014-05-27	1013398	1008991	1006775	N	
38	2014-05-27	1012584	995071	1011491	N	Y
39	2014-05-27	1014280	970094		Y	Y
40	2014-05-27	1008945	1003528	1010551	Y	Y
41	2014-05-27	979879	977484	996208	N	Y
42	2014-05-27	1014957	978010	987719	N	Y
43	2014-05-27	1015323			N	Y
44	2014-05-27	1009198	1009205	977219	Y	Y
45	2014-05-27	987838			Y	Y
46	2014-05-27	1013217			Y	Y
47	2014-05-27	991994	987758	972108	Y	unclear
48	2014-05-27	1016072	987418	1016069	Y	N
49	2014-05-28	956529			N	Y
50	2014-05-28	991148	1014244		Y	
51	2014-05-28	989768			Y	
52	2014-05-28	1016227	1016221	1016233	Y	

Continued on next page ⇓

Subject	Survey Date	Recommendations			Answers	
		I	II	III	Q#1	Q#2
53	2014-05-28	976521	971856	1016973	Y	Y
54	2014-05-28	1011439	1013008	1014354	N	Y
55	2014-05-28	1004153			N	Y
56	2014-05-28	1010732			N	
57	2014-05-28	1016730			Y	unclear
58	2014-05-28	992401			Y	Y
59	2014-05-28	990354			N	Y
60	2014-05-28	1004154	987924		N	N
61	2014-05-28	1013821	1011541	1012632	Y	Y
62	2014-05-28	991342			N	N
63	2014-05-28	1004365	1004377		Y	Y
64	2014-05-28	987111	976148	1014991	Y	
65	2014-05-28	958098	988434	1016155	Y	Y
66	2014-05-28	999125	999119		N	Y