Bug Assignment: Insights on Methods, Data and Evaluation

by

Ali Sajedi Badashian

A thesis submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Computing Science

University of Alberta

Abstract

The bug-assignment problem is prevalently defined as ranking developers based on their competence to fix a given bug. Previous methods in the area used machine-learning or information-retrieval techniques and considered textual elements of bug reports as evidence of expertise of developers to give each of the developers a score and sort the developers for the given bug report.

Despite the importance of the subject and the substantial attention it has received from researchers during last 15 years, still it is a challenging, time-consuming task in large software projects. Even there is still no unanimity on how to validate and comparatively evaluate bug-assignment methods and, often times, methods reported in the literature are not reproducible. In this thesis, we make the following contributions.

1) We investigate the effect of three important experimental-design parameters in the previous research; the evaluation metric(s) they report, their definition of who the real assignee is, and the community of developers they consider as candidate assignees. Supported by our experiment on a comprehensive data set of bugs we collected from Github, we propose a systematic framework for evaluation of bug-assignment research. Addressing those aspects supports better evaluation, enables replication of the study and promotes its usage in other research or industrial applications.

2) We propose a new bug-assignment approach relying on the set of Stack Overflow tags as the thesaurus of programming keywords. Our approach, called Thesaurus and Time based Bug Assignment (TTBA), weights the relevance of a developer's expertise based on how recently they have fixed a bug with keywords similar to the bug at hand. In spite of its

simplicity, our method predicts the assignee with high accuracy, outperforming state-of-the-art methods.

3) We extend TTBA to consider a broader record of the developer's expertise, considering multiple sources of evidence of expertise. Then we investigate the information value of these information sources, considering various technical contributions to the project and contribution to social software platforms. We show that in addition to bug-fixing contributions, other technical and even social contributions of the developers within version control system are useful for bug-assignment. We also show that extending the sources of expertise can improve the accuracy of assignee recommendations.

4) We study the impact and usefulness of the above contributions using a comprehensive data set of bugs we collected from 13 long-term open-source projects in Github. In addition to the technical work by developers, this data set includes social contributions of developers in the version control system. This is one of the biggest data sets made available online for further studies and research.

Preface

This thesis is an original contribution by Ali Sajedi-Badashian. Parts of this thesis has been published before or submitted for publication.

Chapter 2 of this thesis has been submitted for publication:

- Ali Sajedi-Badashian and Eleni Stroulia, "A Systematic Framework for Evaluating Bug-assignment Research", ACM Computing Surveys (CSUR), 2018 Sajedi-Badashian and Stroulia 2018a.

Chapter 3 of this thesis has been submitted for publication:

- Ali Sajedi-Badashian and Eleni Stroulia, "TTBA: Thesaurus and Time Based Bug-Assignment", Journal of Systems and Software, 2018 Sajedi-Badashian and Stroulia 2018c.

Chapter 4 of this thesis has been submitted for publication:

- Ali Sajedi-Badashian and Eleni Stroulia, "The Information Value of Different Sources of Evidence of Developers' Expertise for Bug Assignment", International Conference on Computer Science and Software Engineering (CASCON), 2018 Sajedi-Badashian and Stroulia 2018b.

Chapter 5 of this thesis has been published in:

- Ali Sajedi-Badashian, Abram Hindle and Eleni Stroulia, "Crowdsourced Bug Triaging", International Conference on Software Maintenance and Evolution (ICSME), 2015 Sajedi-Badashian and Stroulia 2018b.
- Ali Sajedi-Badashian, Abram Hindle and Eleni Stroulia, "Crowdsourced Bug Triaging: Leveraging Q&A Platforms for Bug Assignment", Fundamental Aspects of Software Engineering (FASE), 2016 Sajedi-Badashian and Stroulia 2018b.

There are other research that performed during the development of this thesis but are not discussed in this manuscript:

- Ali Sajedi-Badashian, Afsaneh Esteki, Ameneh Gholipour, Abram Hindle and Eleni Stroulia, "Involvement, Contribution and Influence in GitHub and Stack Overflow", International Conference on Computer Science and Software Engineering (CASCON), 2014 Sajedi-Badashian et al. 2014.

- Ali Sajedi-Badashian and Eleni Stroulia, "Measuring user influence in github: the million follower fallacy", CrowdSourcing in Software Engineering (CSI-SE), 2016 Sajedi-Badashian and Stroulia 2016.

- Ali Sajedi-Badashian, Vraj Shah and Eleni Stroulia, "GitHub's big data adaptor: an eclipse plugin", International Conference on Computer Science and Software Engineering, 2015 Sajedi-Badashian, Shah, et al. 2015.

- Ali Sajedi-Badashian and Eleni Stroulia, "Realistic bug triaging", Software Engineering Companion (ICSE-C), IEEE/ACM International Conference on, 2016 Sajedi-Badashian 2016.

To my mother
For teaching me "life",

To my father, who is not between us anymore,
For inspiring me "honesty",

To my wife, Zahra, and my daughter, Elham,
For their love, patience and dedication in helping me during my studies.

Acknowledgements

I hereby thank Professor Eleni Stroulia for her supervision, advice and support during my studies.

I also thank the financial supporters. Parts of this work were supported by the following grants:

- Queen Elizabeth II Graduate Scholarship[1] funded by Faculty of Graduate Studies and Research (FGSR)[2] at University of Alberta.
- Graduate Student Scholarship[3] funded by Alberta Innovates - Technology Futures (AITF)[4]
- Natural Sciences and Engineering Research Council of Canada (NSERC) and the GRAND NCE.
- European Joint Conferences on Theory and Practice of Software (ETAPS) 2016 student scholarship (travel award)
- Faculty of Graduate Studies and Research (FGSR) Travel Award
- Two GSA Academic Travel Awards
- GRAND Postgraduate Scholar Award

[1] https://www.ualberta.ca/graduate-studies/awards-and-funding/scholarships/queen-elizabeth-ii
[2] https://www.ualberta.ca/graduate-studies
[3] https://fund.albertainnovates.ca/Fund/BasicResearch/GraduateStudentScholarships.aspx
[4] https://innotechalberta.ca

Contents

1 Introduction **1**
- 1.1 Problem statement . . . 2
- 1.2 Contributions . . . 4
- 1.3 Thesis Outline . . . 6

2 A Systematic Framework for Evaluating Bug Assignment Research **7**
- 2.1 Introduction . . . 10
- 2.2 A survey on Bug-assignment Research . . . 13
  - 2.2.1 Survey protocols and process . . . 13
  - 2.2.2 Bug-assignment Tasks and Objectives . . . 15
  - 2.2.3 Bug-Assignment Methodologies . . . 19
- 2.3 Select Empirical Bug-assignment Studies . . . 21
  - 2.3.1 Knowledge Assumptions of Bug-assignment Methods . . . 24
  - 2.3.2 Metrics for Evaluation of Bug-assignment Research . . . 25
- 2.4 A Discussion of Bug-assignment Evaluation Metrics . . . 27
- 2.5 Dimensions of Variability in Bug-assignment Empirical Studies . . . 31
  - 2.5.1 Ground-truth assignee . . . 32
  - 2.5.2 Developer community . . . 37
- 2.6 Experiment setup and Data Set . . . 38
  - 2.6.1 The Data Set . . . 39
- 2.7 Findings . . . 41
  - 2.7.1 Comparing Different Types of "Ground-truth Assignee" . . . 41
  - 2.7.2 The effect of "Developer Community" . . . 44
- 2.8 Discussion . . . 48
  - 2.8.1 The proposed evaluation framework . . . 49
  - 2.8.2 Threats to validity . . . 51
- 2.9 Conclusions . . . 52

3 *TTBA*: Thesaurus and Time Based Bug Assignment **54**
- 3.1 Introduction . . . 57
- 3.2 Background and Related Research . . . 58
- 3.3 Recognizing Developers with Relevant Expertise: The *TTBA* Metric . . . 61
  - 3.3.1 *TF-IDF* . . . 64
  - 3.3.2 Focusing on a Thesaurus of Terms . . . 65
  - 3.3.3 Recency-aware Term Weighting . . . 67

3.4 The Data Set . . . 68
3.5 Evaluation . . . 69
3.5.1 Evaluation Metrics . . . 70
3.5.2 Tuning and Optimization . . . 70
3.6 Results . . . 72
3.6.1 Comparisons Against Implemented Baseline Methods . . . 72
3.6.2 Comparison Against Results Reported in the Literature . . . 73
3.7 Discussion . . . 74
3.7.1 Intuitions About Comparison of the Implemented Approaches . . . 76
3.7.2 Intuitions About Comparison of Reported Results . . . 77
3.7.3 Threats to Validity . . . 78
3.8 Conclusions and Future Work . . . 82
Appendix 3.A Appendix: Details of mutation experiment for tuning 6 parameters of our method . . . 83

4 An Investigation Into the Information Value of Different Sources of Evidence on the Developers' Expertise for Bug Assignment 89
4.1 Introduction . . . 92
4.2 Background and Related Research . . . 94
4.2.1 Knowledge assumptions of Bug-Assignment . . . 94
4.3 Bug Assignment Based on Multiple Sources of Evidence of the Developers' Expertise . . . 98
4.3.1 TTBA: A Compositional Similarity Metric for Bug Assignment . . . 98
4.4 Experimental Design . . . 99
4.4.1 Data set . . . 100
4.5 Value of Various Sources of Expertise . . . 101
4.5.1 Considering additional textual information as evidence of expertise . 102
4.5.2 Considering References to the developers' names . . . 104
4.6 Validating *Multisource* Approach on the Whole Data Set . . . 106
4.7 Threats to Validity . . . 109
4.8 Conclusions and Future works . . . 111

5 Utilizing Beyond-project Sources of Expertise for Bug Assignment 113
5.1 Introduction . . . 116
5.2 Literature Review . . . 117
5.3 A Social Bug-Triaging Model . . . 120
5.3.1 Social Metrics of Expertise . . . 121
5.3.2 A Bug-Specific Social Metric of Expertise . . . 122
5.4 Evaluation . . . 125
5.4.1 Experiment Setup . . . 126
5.4.2 Comparison to State of the Art . . . 126
5.4.3 Implementation . . . 128
5.4.4 Performance of Variant Social Metrics of Expertise . . . 128
5.4.5 Performance of the RA_SSA_Z_score$_{u,b}$ . . . 130
5.5 Analysis . . . 131

5.6 Conclusions and Future Work 133
Appendix 5.A Appendix: A Preliminary study for Usage of External Beyond-project Sources of Expertise in Isolated Settings 135
5.A.1 Introduction and Background 137
5.A.2 A Social Bug-Triaging Method 138
5.A.3 Evaluation 141
5.A.4 Conclusions and Future Work 146
Appendix 5.B Appendix: Can External Beyond-project Sources of Expertise be Useful in General Settings? 147
5.B.1 Information From Project Families 147
5.B.2 Information from Other Technical Networks 149

6 Conclusions and Future Works **152**
6.1 Future Works 154

References **155**

List of Tables

2.1 A summary of the methods and techniques used in the literature 16
2.1 A summary of the methods and techniques used in the literature 17
2.2 A review of the used information for bug assignment in selected BA studies 25
2.3 The evaluation metrics and other design choices of the selected research 26
2.4 The types of assignment used in previous studies, varied from T1 to T4 36
2.5 The data set including T1 to T5 bug-assignments in each project 40
2.6 MAP (%) for 13 projects using different assignee types 42

3.1 Obtaining best configurations for the main factors affecting the accuracy of *TTBA* 71
3.2 Comparing *TTBA* results against *TF-IDF* and *Time-TF-IDF* (Shokripour et al. 2015) 73
3.3 The evaluation measures and other design choices of the selected research 75
3.4 The results of optimizing secondary factors affecting the accuracy of *TTBA* 87

4.1 A review of the used information for bug assignment in previous research 95
4.2 Data set different evidence of expertise in 13 projects 101
4.3 Determining context weights for additional pieces of information in test projects 103
4.4 Statistics of the references to the developers' names in three test projects . . 104
4.5 Determining whether to prioritize the referenced developers or not 106
4.6 Comparison of the results of our approach against *TTBA* 107
4.7 Available multi-source data in different projects 108

5.1 An example of scoring regarding the bug shown in Figure 5.1 based on the users' activities in Stack Overflow 121
5.2 Accuracy results for preliminary approaches and tuning 130
5.3 Accuracy results for different simulated approaches compared with ours . . . 131
5.4 Recent Bug-Triaging Methods 138
5.5 Example of different scores for users 140
5.6 Results of different simulated approaches compared with ours 144
5.7 Statistics of the shared users with sub-projects or Stack Overflow 148
5.8 Results of using external sources of expertise 149
5.9 Statistics of the answers posted by the developers in Stack Overflow 151

List of Figures

2.1 Methods used for bug-assignment (overall) 18
2.2 Methods used for bug-assignment (details) 19
2.3 An example of use of MAP for bug-assignment over three bugs 32
2.4 A sample bug report (#257) in *Travis-ci* 33
2.5 Distribution of MAP over 13 projects using five different assignment types. T5 has the lowest variance 43
2.6 The effect of filtering less-active developers, and the bug reports they fixed, on accuracy of results 47

3.1 A sample bug report (#92) in *www.html5rocks.com* 62

4.1 A sample bug report and information sources available around it 97
4.2 The data model of our Github data set 102

5.1 An example bug report (selected fields) 120

Nomenclature

abs_recency: absolute recency which is obtained by considering the time passed from an evidence of expertise, ... p. 67

A_score: the score of answers provided by a Stack Overflow user, ... p. 122

BA: Bug Assignment, ... p. 15

bug: an error, flaw, failure, problem or needed change in the software[5], ... p. 1

Bug Assignment: Ranking developers based on their expertise for fixing a given bug, ... p. 10

bug description: the descriptive words of a bug which is written by a developer to explain that bug, ... p. 55

bug report: the report written by a developer regarding an existing bug (might include stack traces or reproduction steps), ... p. 1

context: the "context" or "subject" score related to the type of a sub-document (e.g., evidence of expertise), which is considered as a coefficient in the *Multisource* scoring function, ... p. 98

cross validation: an out-of-sample testing technique in which the result of a new bug-assignment method is validated on a data set with known developers as ground-truth assignees, ... p. 12

D: the set of all the bugs –up to a specific point in time when the new query (bug) is assigned to a developer, ... p. 38

d: a developer that might be assigned any number of bugs, ... p. 38

developer's expertise record: the tracks of a developer in the VCS in the form of bug-fix or other programming-related contributions, ... p. 1

developer community: the list of developers who might be considered as assignee for any new bug in a project (this is a set of developers which is sorted for each bug reports in evaluation and cross-validation of a method against ground-truth assignees), p. 37

[5]The actual notation of *bug* mostly indicates a problem in developer's code and is more restricted than *issue*. The usage of *bug* in this study is the same as *issue*.

distinctiveness in Stack Overflow: the extent that a Stack Overflow tag is specific, noticeable and explicit; the less common Stack Overflow tags are more specific and distinct than the common tags, ... p. 5

efficiency of BA: the ability to recommend desirable developers for a set of given bugs so that the bugs can be fixed correctly and there will be no need to toss to other developers, ... p. 27

freq: frequency (number of repeats) of a specific term in a query, ... p. 64

golden standard: see "ground truth", ... p. 11

ground truth: the real bug-assignment data against which researchers validate their bug-assignment methods (showing who was the correct assignee for each real bug) and generally evaluate their methods, ... p. 2

ground truth assignee: the developer(s) who actually worked towards fixing a given bug in the project; the researchers validate their bug-assignment methods by comparing the developers recommended by their approach against these developers, ... p. 32

idf: inverse document frequency which measures how important a term is, regarding all the documents, ... p. 64

issue: see "bug", ... p. 1

issue report: see "bug report", ... p. 1

multisource: multi-source variant of *TTBA* including various sources of expertise, .. p. 98

project member: any developer in the project who can do any type of bug-fix (including committing and referencing the bug as "fixed", reviewing and closing the bug or generally being tagged as assignee at the time of closing the bug), ... p. 45

q: a query (bug) that needs to be assigned to a developer for fixing, ... p. 38

Q_score: the score of a Stack Overflow user related to the questions he/she asked, . p. 123

RA_SSA_Z_score: recency-aware *SSA_Z_score*, ... p. 125

recency: a factor injected in our expertise metrics to emphasize on newer evidence, . p. 68

rel_recency: relative recency which is obtained by considering the amount of worked after an evidence of expertise, ... p. 67

sd: sub-document, ... p. 66

SSA_Z_score: the social and subject-aware score of a Stack Overflow user obtained from his/her posts (i.e., questions and answers), ... p. 123

technical keyword space: all the Stack Overflow tags, ... p. 5

technicality: the level or state a term is technical regarding programming aspects, ... p. 3

tf: term frequency (we consider a normalized version by dividing the number of times a term is mentioned in a document by the document length), ... p. 64

$tf - idf$: Term Frequency - Inverse Document Frequency, ... p. 38

$TTBA$: Thesaurus and Time Based Bug-Assignment, ... p. 58

u: a Stack Overflow user who is also a Github developer, ... p. 123

VCS: Version Control System, ... p. 4

w: weight of a term (obtained from its appearance in Stack Overflow), ... p. 65

Z_score: the score of a Stack Overflow user related to the answers he/she provided, p. 123

α: the coefficient for "social" section of $RA\_SSA\_z\_score$, ... p. 125

β: the coefficient for "recent activity" section of $RA\_SSA\_z\_score$, ... p. 125

μ: a normalization factor in the SSA_Z_score to adjust the number of questions with number of answers, ... p. 123

Chapter 1

Introduction

In big software projects, many *bugs* (*issues*) are reported every day which need to be fixed in a reasonable time. One of the main triaging steps, after receiving a bug report (issue report), is to assign it to an appropriate developer. In big projects, addressing bugs properly needs effort from the project's management team, which is too expensive. As a result, researchers devised semi-automatic solutions to help managers assign bugs to the developers. Bug assignment is the process of ranking developers in terms of the relevance of their expertise to fix a new bug report.

The problem has already received substantial attention over the past 15 years (Aljarah et al. 2011; Bhattacharya and Neamtiu 2010; Jeong et al. 2009; Linares-Vásquez et al. 2012; Liu et al. 2016; Nguyen et al. 2014; Shokripour et al. 2013). Despite the vast attention of researchers, bug-assignment is still an expensive, time-consuming task in software projects (Saha et al. 2015). More than half (and up to 90%) of software development cost is regarding maintenance (Bhattacharya et al. 2012; Seacord et al. 2003), and a great deal of maintenance work is dealing with bugs. Large projects receive hundreds of bug reports daily (Tian et al. 2016), which get recorded in their issue-tracking tools. As an example, in Eclipse and Mozilla, it takes about 40 and 180 days respectively to assign a bug to a developer (Bhattacharya et al. 2012). In ArgoUML and PostgreSQL, the median time-to-fix for bugs in some projects is around 200 days (Kim and Whitehead Jr 2006). In Eclipse, 24% of the bugs are re-assigned to another developer, before getting fixed (Baysal et al. 2009). The reason for re-assignment can be for determining the ownership of the bug or other reasons (Guo et al. 2011). Even the fixed bugs, in both Eclipse and Mozilla, have been re-assigned at least once in more than 90% of the cases (Bhattacharya et al. 2012; Jonsson et al. 2016).

This indicates the need for more accurate bug-assignment methods for big projects (T. Zhang et al. 2016; W. Zhang et al. 2016b) to help the project managers pick the right assignees for each received bug.

In this thesis, we develop a new method for bug-assignment, and validate it through a series of experiments on a substantial data set.

1.1 Problem statement

To help large software teams like open-source projects handle bugs effectively, it is needed to enhance the current methods and develop more accurate assignee recommendations. This includes revisiting the problem and design choices used for experimentation in the field, developing more accurate methods and utilizing the most practical and useful data. We review these problems in three categories;

1) Despite the importance of the subject and the substantial attention it has received from researchers during last 15 years, still it is a challenging, time-consuming task in large software projects. On the other hand, validation and evaluation of the bug-assignment methods differs a lot from case to case, and, often times, methods reported in the literature are not reproducible. Most research in the field involve a number of challenging questions about ***how*** to implement a more accurate bug-assignment method, including "how to find and relate different pieces of information to a bug report to assign it to a developer", "how to utilize similarity measures to match a bug report with a developer?", "how to use other clues or heuristics to connect a bug report to a candidate developer as the potential assignee", "how to take into account the developers' workload?" and so on. However, there is no previous study aiming at addressing the following ***what*** questions:

- What is the best metric for evaluating a bug-assignment research? What are the criteria for choosing evaluation metric in a bug-assignment experiment? Can the choice of evaluation metric bias the evaluation process?

- What is the best definition of "ground-truth assignee"? What is the best "ground truth" for evaluation (cross-validation of a developer recommendation against some real data)? How can we judge if a recommended developer is a good fit for fixing a given bug or not? And what are the exact criteria for this judgment (to identify the ground-truth assignees for a given bug) in a project?

- What is the best definition for "developer community" from which the bug-assignment methods recommend appropriate developers? Which developers in the project should be considered as "developer community" (considered as the pool of candidate assignees), in order to have a fair evaluation? And does *filtering* this community bias the results?

To the best of our knowledge, there is no previously published survey aiming at discussing the main objectives, methods, metrics and information used for bug-developer matching. In addition to addressing the above questions, providing a comprehensive survey helps us identify the pitfalls and challenges in the field and pinpoint best practices to make future bug-assignment research more useful and reproducible.

2) Despite all the previous work in the area of bug-assignment, still more accurate developer recommendation methods are needed to facilitate bug-fix in large projects with hundreds or even thousands of developers. In order to develop an automated method for assigning a bug to the developer best qualified to fix it, the first question is to decide on "how to match the information available on the bug with the information available on the developer's prior experience and contributions".

The most prevalent arrangement used for finding the best developer to fix a given bug is "matching" between the text of the new bug report and the profile of each developer (i.e., text of previously fixed bugs or other contributions made by a developer). There are two important elements affecting this matching.

- The *technical terms* are very important. Some previous studies considered specificity of the terms, by measuring the statistics of the terms in the corpus (whole bug reports). This mitigates the need for emphasizing on technical programming terms. However, no previous research applied "technicality" of the terms as indication of importance of the keywords.

- The *time of usage* is another important aspect of text matching. Developers' interests and areas of work change during time. While old evidence of usage of a term by a developer is still indicating some knowledge for that developer, its importance diminishes with passage of time. A few previous studies considered the time of evidence but only as a high-level estimate or decay factor.

The above two cases make the recognition of the needed expertise different from textual-similarity assessment, which has been the prevalent paradigm for this task to date. So further

investigation is needed to obtain a better bug-developer matching.

3) The most prevalent types of data used for bug-developer matching is the text of bug reports and there are only limited studies on addition of some other information (e.g., text of commits). However, there are many other technical evidence of expertise of developers in the open-source projects which can be useful for this matching. Furthermore, the utilization of social contributions of developers in the project is another possible source of expertise which should be investigated. Out of all the available sources of expertise in the Version Control System (VCS), some sources are fruitful in connecting a bug to a developer, while the others might be noisy. So, an investigation is needed to highlight the proper information for utilization in further bug-assignment research.

1.2 Contributions

In this thesis, we focus on revisiting the problem and design choices used for experimentation in the bug-assignment field, developing more accurate methods and identifying and utilizing the most practical and useful data. This thesis makes the following contributions.

1) A Systematic Framework for Evaluating Bug Assignment Research

We first perform a systematic review of the broad bug-assignment research field, including methods, evaluation metrics and information used for previous bug-assignment research. Then, we review three important experimental-design parameters, namely the evaluation metric(s) they report, their definition of who the ground-truth assignee is, and the community of developers they consider as candidate assignees. Next, due to the substantial variability on these criteria, we formulate a systematic experiment to explore the impact of these choices, and, argue that Mean Average Precision (MAP) is the most informative evaluation metric, the ground-truth assignee should be defined as "any developer who worked toward fixing a bug" and the developer community should be defined as "all the project members".

The comprehensive survey and proposed systematic framework are presented in Chapter 2.

2) A New Bug Assignment Method Relying on Technical Terms and Considering the Information Recency

We propose a new bug-developer similarity metric which is originally based on *TF-IDF*.

It includes two important enhancements regarding technical aspects of keywords and time of usage of them by the developers. Our similarity metric is a **t**hesaurus and **t**ime-aware **b**ug-**a**ssignment, henceforth TTBA; 1) The metric deals with the relevance of a developer's expertise to a given bug by considering the **technical-keyword space**. Ignoring all words that do not belong to the technical vocabulary of Stack Overflow tags, which are curated by the software-engineering community, our method weighs the importance of the keywords based on their distinctiveness in Stack Overflow. 2) Furthermore, since the developers' expertise shifts as their tasks evolve over time, our similarity metric takes into account the recency of the technical-keyword appearance in the developer's record. It benefits from **high granularity of the time** of usage of the terms in previous bug-assignments. We show that our model notably enhances the assignee recommendation accuracy and outperforms state-of-the-art methods.

The TTBA approach is presented in Chapter 3.

3) Investigation of Information Value of Social and Technical Contributions of Developers in Open-source Projects

Finally, we study the utilization of various data in bug-assignment with two goals; first, to increase the accuracy of our assignee recommendation. Second, to investigate the information value of different pieces of available data for bug-assignment. This helps further researchers to focus on appropriate fields of data effectively in their different methods.

We show that using this *Multisource* approach, the overall accuracy is increased. Also, we show that most textual bug-related elements (e.g., title and description of bugs as well as their comments) are effective sources while some other available information are not that helpful.

We also check the usefulness of external beyond-project sources of expertise. First, we investigate the isolated settings in which only the developers with external evidence of expertise (i.e., Stack Overflow) participate. Then, we generalize this to all the developers. We show that the approach works in isolated settings, but in general, with existence of inside-project data, the external evidence cannot make the predictions any better. In either case, we discuss that the external evidence will be useful in specific situations (e.g., in new projects).

The basic *Multisource* approach and the investigation of internal sources are discussed in Chapter 4. The further investigation of external beyond-project sources is discussed in

Chapter 5

4) Data sets for further research

We support our arguments and validate our new proposed methods by several experiments using an extensive data set of bugs, developers and their technical and social contributions we have curated from Github as the most popular version control system. Our latest data set is regarding 13 big open-source projects during +5 years. This data set is one of the most comprehensive and recent data sets which is publicly available for further bug-assignment research.

1.3 Thesis Outline

The structure of this thesis is as follows; In Chapter 2, after a survey on bug-assignment methods and design choices, we propose our bug-assignment evaluation framework. Then, in Chapter 3, we propose our new bug-assignment method and discuss about its evaluation. Chapter 4 is devoted to the *Multisource* approach and its validation and Chapter 5 discusses the usefulness of external sources. Finally, Chapter 6 concludes the thesis with some avenues for future work.

Chapter 2

A Systematic Framework for Evaluating Bug Assignment Research

Preface

In this chapter, we first perform a systematic review of previous bug-assignment approaches and discuss three design choices of their experiments; the utilized metrics, the adopted definition of ground-truth assignee and definition of developer community. Then, we introduce our evaluation framework which includes guidelines regarding those three aspects.

We show that variability on the above criteria can affect the results of bug-assignment experiments. We argue that the best evaluation metric is MAP, the developer community should be defined as "all project members" and that the best definition of ground-truth assignee should include author of any work toward fixing the bugs.

This section has been submitted to the Journal of Software: Evolution and Process.

Abstract

Bug assignment is the task of ranking candidate developers in terms of their potential competence to fix a bug report. Numerous methods have been developed to address this task, relying on different methodological assumptions and demonstrating their effectiveness with a variety of empirical studies with numerous data sets and evaluation criteria. Despite the importance of the subject and the attention it has received from researchers, there is still no unanimity on how to validate and comparatively evaluate bug-assignment methods and, often times, methods reported in the literature are not reproducible.

In this chapter, we first report on our systematic review of the broad bug-assignment research field. Next, we focus on a few key empirical studies and review their choices with respect to three important experimental-design parameters, namely the evaluation metric(s) they report, their definition of who the ground-truth assignee is, and the community of developers they consider as candidate assignees.

The substantial variability on these criteria led us to formulate a systematic experiment to explore the impact of these choices. We conducted our experiment on a comprehensive data set of bugs[1] we collected from 13 long-term open-source projects, using a simple Tf-IDf similarity metric. Based on our experiment, we argue that MAP is the most informative evaluation metric, the developer community should be defined as "all the project members", and the ground-truth assignee should be defined as "any developer who worked toward fixing a bug".

[1] The data set, source code, documentations and detailed output results are available at: https://github.com/TaskAssignment/MSBA-outline

2.1 Introduction

Bug-assignment (BA) is an important problem for the software-engineering industry. As a key task of software development as well as quality-assurance process, it aims at identifying the most appropriate developer(s) to fix a given bug. Typically, BA methods consider the developers' previous bug assignments and other activities as indicators of their expertise and rank the developers' relevance to the bug in question using a variety of heuristics.

Previous BA research involves a number of challenging but well-addressed questions, related to ***how*** to implement a more accurate BA method, including "how to gather evidence for a developer's expertise in software projects?", "how to relate different pieces of information to a bug report to assign it to a developer", "how to utilize similarity measures to match a bug report with a developer?", "how to use other clues or heuristics to connect a bug report to a candidate developer as the potential assignee", "how to take into account the developers' workload?" and so on. These types of questions are addressed in almost all the previous BA studies (Aljarah et al. 2011; Bhattacharya and Neamtiu 2010; Jeong et al. 2009; Linares-Vásquez et al. 2012; Liu et al. 2016; Nguyen et al. 2014; Shokripour et al. 2013).

Despite the vast attention of researchers, BA is still an expensive, time-consuming task in software projects (Saha et al. 2015). Between 50% to 90% of software development cost is regarding maintenance (Bhattacharya et al. 2012; Seacord et al. 2003), and a great deal of maintenance work is dealing with bugs. Large projects receive hundreds of bug reports daily (Tian et al. 2016), which get recorded in their issue-tracking tools. In Eclipse and Mozilla, it takes about 40 and 180 days respectively to assign a bug to a developer (Bhattacharya et al. 2012). In ArgoUML and PostgreSQL, the median time-to-fix for bugs in some projects is around 200 days (Kim and Whitehead Jr 2006). In Eclipse, 24% of the bugs are re-assigned to another developer, before getting fixed (Baysal et al. 2009). Even the fixed bugs, in both Eclipse and Mozilla, have been re-assigned at least once in more than 90% of the cases (Bhattacharya et al. 2012; Jonsson et al. 2016).

This indicates the need for more accurate BA methods for big projects (T. Zhang et al. 2016; W. Zhang et al. 2016b) to help automate the BA task in the issue-tracking tools. Currently, no issue-tracking tool automates this task, which is still manual or, at best, semi-automated. Despite the long record of related research in the field, it is hard to apply the proposed BA research in practice (Jie et al. 2015). Even it is a question whether the previously introduced BA approaches can be applied in large proprietary projects with acceptable

performance or not (Jonsson et al. 2016). Still the industry needs more practical methods, with higher standards (Bhattacharya et al. 2012).

A study can be useful in industry if it is assessed using a realistic validation, and, is reproducible. The term "Reproducible research", introduced by Jan Claerbout, tries to infuse standards for publications in computing science (Fomel and Claerbout 2009) to make the research results more useful. The idea is that the main product of a research is not only the paper, but also the full contents and materials that may be used to build upon the research and reproduce the results. These materials include but are not limited to source code and data sets used in computational science experiments (Fomel and Claerbout 2009; Schwab et al. 2000). In *Science*, Roger Peng mentioned the potential of serving as "*a minimum standard for judging scientific claims*" an important characteristic of reproducibility (Peng 2011). He indicated the need for *data*, *meta-data* and *code* being linked to each other and to the corresponding publications as prerequisites for *full reproducible* research. Posing specific assumptions, conditions or dependencies in some of the previous researches hinders this.

Regarding BA research, one study may depend on very detailed data or sophisticated meta-data that rarely are accessible. Or it may pose biased conditions or filtering (e.g., to remain few number of developers or small number of bug reports) in favor of itself. These make the publication of future high-quality studies harder since the comparison against those studies cannot be fair. To summarize, we found the following evaluation-related problems;

1. There is no generally agreed-upon definition of the "ground-truth assignee" (i.e., who the best developer for a given bug really is) against which to validate BA methods (i.e., the notion of golden standard). The broader the definition of ground-truth assignee is, the easier the prediction will be.

2. The rule of thumb for defining the group of developers who are considered as candidate assignees differs a lot from case to case. The bigger the community is considered, the more difficult the BA task becomes.

In addition, there is no unanimity in evaluation and reporting metrics which are used in assessment of the proposed methods. Needless to say, many of the previously used evaluation measures do not reflect the effectiveness of the proposed method properly. Compounded by the fact that many research publications do not share their data or code, and the experimental data sets vary substantially in their size and complexity, the above issues can become

critical reproducibility problems. And if these problems are not addressed correctly in a BA study, the usefulness of the study can be questioned, or even disproved in further research.

In this work, we systematically examine the above-mentioned evaluation-related factors. We propose a practical framework to be able to fairly evaluate goodness of a method and compare it against as many methods reported in the literature as possible. Rather than discussing ***how*** to establish a new BA method, we cover two "***what*** *questions*". These questions are the main research questions of this study:

Research Questions:

1. **What is the most practical definition of "ground-truth assignee"?**

   What is the best "ground truth" for evaluation (cross-validating against a developer recommendation)? How can we judge if a recommended developer is a good fit for fixing a given bug or not? And what are the exact criteria for this judgment (to identify the ground-truth assignees for a given bug) in a project?

2. **What is the best definition for "developer community" from which the bug-assignment methods recommend appropriate developers?**

   Which developers in the project should be considered as "developer community" (considered as the pool of candidate assignees), in order to have a fair evaluation? Are the members of this community normally affected by the definition of ground-truth assignee? Does the definition of this community affect the results (reported based on evaluation measures)? And does *filtering* this community bias the results?

Answering the above questions helps to establish a baseline for fair comparison of the results in further BA research. To answer to the above questions, we have curated an extensive data set, including bug reports from thirteen open-source projects, their meta-data and textual information and their assignee(s), according to the different definitions of assignee discussed in this study. We extracted the information of these projects using Github APIs. This data set is used for an experiment in support of the arguments we provide. We also publish this data set online for further research.

The **contributions** of this study are as follows: First, in a systematic review, we summarize key methods in the literature and discuss them from different aspects. Then, we investigate the current BA evaluation measures, and provide arguments towards selecting the best evaluation measure for BA research. After that, we identify two important dimensions of variability in the evaluation of BA methods and put forward a framework that argues for special choices in these dimensions. We motivate our framework by demonstrating how these aspects affect the reported results with a study, a big data set and a standard similarity

metric, *tf-idf*.

The remainder of this chapter is organized as follows. We perform the systematic review in Sections 2.2 and 2.3. Section 2.4 discusses the evaluation metrics used for evaluating BA research. Section 2.5 introduces the common definitions of the two dimensions or variability, "ground-truth assignee" and "developer community". After Section 2.6 which describes the experiment setup, in Sections 2.7, we discuss the two research questions of the study regarding those two dimensions of variability. Section 2.8 reflects on some implications and discussions about the provided framework. Finally, Section 2.9 concludes the chapter.

2.2 A survey on Bug-assignment Research

In the context of our own research in the area of BA, we found that there is no comprehensive review on the field investigating different aspects of the problem. In this section, we discuss such a review. We first explain our survey methodology and discuss the main objectives, methods, metrics and information used for bug-developer matching in previous research.

2.2.1 Survey protocols and process

This section presents the process we followed for doing the survey. We conducted the systematic review according to the guidelines mentioned in (Budgen and Brereton 2006; Kitchenham and Charters 2007; Weidt and Silva 2016).

The first goal is to select all the papers that propose a new BA method to get a proper intuition about the area and proposed solutions. We study the BA objectives, formulations and methods. Later, we will focus on a subset of these research and discuss their methods more in-depth.

We performed a publication search on four research databases; ACM[2], IEEE [3], Sciencedirect[4] and Springer[5]. We searched for *bug assignment*, *"bug assignment"*, *bug triaging* and *"bug triaging"* in each of the databases. For each search, we captured the top 100 returned results (which were usually returned in the first 4 or 5 pages). We reviewed all the 100 results of each search and considered them in our survey if they were introducing a BA approach. The selection criteria were as follows:

[2]https://dl.acm.org/dl.cfm
[3]http://ieeexplore.ieee.org/Xplore/home.jsp
[4]https://www.sciencedirect.com/
[5]https://link.springer.com/

Inclusion criteria:

1. Papers must be published in peer-reviewed conferences or journals;
2. Papers must describe a new BA-related methodology; and
3. Papers must follow the formulation of "ranking developers for the given bugs", or similar formulations.

Exclusion criteria:

1. Papers that are only tool-development or poster papers are excluded;
2. Papers that are about bug-triaging rather than BA are removed[6]; and
3. Papers that investigate "challenges of bug-assignment and reassignment", rather than BA are excluded.

Considering the above criteria, we reviewed all the mentioned results. First, we analyzed the title and abstract. If we found the paper related to the topic, then, we surveyed the paper in more detail to make sure it fits with inclusion and exclusion criteria. We did not consider any filtering on the venue (conference or journal) since all the selected databases support legitimate scientific conferences and journals.

Examples of the eliminated studies[7] included general purpose task-assignment (Helming et al. 2010), applying BA methods in code reviewer recommendation (Yu et al. 2016), bug fix time prediction (Akbarinasaji et al. 2017; H. Zhang et al. 2013), tool-development for BA with no reported accuracy (Bortis and Hoek 2013) and bug-triaging related studies other than BA, including bug report de-duplication (Banerjee et al. 2016; X. Wang et al. 2008), bug localization (Chaparro 2017; Lukins et al. 2010), component-assignment (recommending a component for a given bug report) (Somasundaram and Murphy 2012; Yan et al. 2016) and qualitative studies investigating problems of assignment and re-assignment of bugs (Baysal et al. 2012; Cavalcanti et al. 2014a).

Finally, we found 74 BA-related studies, dating from 2004 to the end of 2017, some of which were extensions or journal versions of the other ones. The 74 extracted papers

[6] Note that our exclusion criteria is about the subjects of the papers (as we investigated in their abstract or even text) not just their keywords. For example, there are some papers called with keyword "bug-triaging" (or similar), but are in fact bug-assignment (Nasim et al. 2011). So we did not exclude those papers.

[7] There were tens of those cases, but we did not count the exact number of those eliminated studies.

targeted the problem using different wordings; bug-triaging, change request assignment, anomaly request assignment, issue-assignment, and more prevalently bug-assignment (BA). We summarized all these papers and extracted the methods each study utilizes.

Table 2.1 shows these papers and the methods they used and Figures 2.1 and 2.2 show a summary of the number of studies using each method. We categorized the methods in six groups; First, different Machine Learning (ML) classifiers as well as Stacked Generalization (which combines several classifiers) are categorized in ML. Then, we put the general Information retrieval (IR) methods that consider the notation of query-document for BA problem as well as IR-based methods like Topic Modeling and Cosine similarity into IR category. The third category is Natural Language Processing (NLP) including the general NLP techniques as well as Information Extraction (IE) methods. Then, methods based on Markov chains (in which a state diagram determines the probability of events), Tossing Graphs (in which a network of developers represent the probability of substitute assignees) and Social Network Analysis (SNA) are represented in Network and Graph based category. The Statistical models including smoothed Unigram Model (in which each bug is represented as an n-dimensional probability vector for the n terms available in the corpus, and the probabilities are smoothed based on their occurrence in the whole corpus) and Kullback–Leibler (KL) Divergence (as a measure of difference between two probability distributions) are shown in the Statistical Models category. Finally, other methods like Fuzzy, Bug Localization (in which the related files to the new bug is estimated and the developers related to those files are considered as possible assignees), Rule-based (in which rules are extracted based on meta-data of the old bugs and used for deciding about the new bugs), Collaborative Filtering (CF) (substituting developers and files / bug reports instead of users and items in the regular usages of CF) and Genetic Algorithm (in which different combinations are built and tested adaptively, representing different assignment objectives) are categorized in the last category.

2.2.2 Bug-assignment Tasks and Objectives

The most prevalent formulation of BA is as follows: "Given a new bug report, identify a ranked list of developers, whose expertise (based on their record of contributions to the project) qualifies them to fix the bug" (Bhattacharya and Neamtiu 2010; Hu et al. 2014; Khatun and Sakib 2016; Matter et al. 2009; Shokripour et al. 2015); this is the formulation most researchers used in their studies. In this study, we consider this formulation. However, there are other formulations that we briefly mention them here.

Table 2.1: A summary of the methods and techniques used in the literature

Study	Machine Learning (ML)							Information Retrieval (IR)							NLP		Network and Graph based			Statistical models				Other methods						
	Naïve Bayes (NB)	Bayesian Network (BN)	Support Vector Machine (SVM)	C4.5	K-Nearest Neighbor (KNN)	Convoluted Neural Network (CNN)	Stacked Generalization (SG) ensemble learner	IR (general)	Latent Semantic Indexing (LSI)	Latent Dirichlet Allocation (LDA)	Topic Modeling (general)	Vector Space Model (VSM)	Term Frequency – Inverse Document Freq. (TF-IDF)	Cosine similarity	NLP (general)	IE (Information Extraction)	Social Network Analysis (SNA)	Tossing Graph (TG)	Markov chains	smoothed Unigram Model (UM)	Kullback–Leibler (KL) Divergence	Developers' vocabulary profile (as expertise matrix)	Term selection / Term weighting	Fuzzy	Bug localization	Rule based	Collaborative Filtering (CF)	Genetic Algorithm (GA)	Others
(Čubranić and Murphy 2004)	✓																												
(Anvik 2006)			✓																										
(Canfora and Cerulo 2006)								✓														✓							
(Anvik et al. 2006)			✓																										
(Kagdi et al. 2008)																									✓				✓
(Baysal et al. 2009)												✓																	
(Lin et al. 2009)			✓										✓																
(Jeong et al. 2009)	✓	✓																✓	✓										
(Matter et al. 2009)																						✓							
(Ahsan et al. 2009)			✓						✓				✓									✓							
(M. M. Rahman et al. 2009)																													✓
(Kagdi and Poshyvanyk 2009)								✓	✓																✓				
(Bhattacharya and Neamtiu 2010)	✓	✓																✓	✓										
(Chen et al. 2010)												✓						✓	✓										
(Nasim et al. 2011)	✓	✓																											✓
(Park et al. 2011)			✓							✓																	✓		
(Wu et al. 2011)					✓								✓				✓												
(Tamrawi et al. 2011b)																						✓		✓					
(Tamrawi et al. 2011a)																						✓		✓					
(Anvik and Murphy 2011)	✓			✓																									
(Aljarah et al. 2011)		✓																					✓						✓
(Kagdi et al. 2012)								✓	✓																✓				
(Bhattacharya et al. 2012)	✓	✓	✓	✓														✓	✓										
(Shokripour et al. 2012)															✓	✓									✓				
(Xuan et al. 2012)	✓		✓														✓												
(Xie et al. 2012)										✓	✓																		
(V. Jain et al. 2012)																						✓							
(T. Zhang and B. Lee 2012)											✓						✓												✓
(Jonsson et al. 2012)	✓		✓		✓		✓																						✓
(Linares-Vásquez et al. 2012)								✓	✓																✓				
(Servant and Jones 2012)																									✓				✓
(Rahmana et al. 2012)																												✓	✓
(Hosseini et al. 2012)	✓																												✓
(T. Zhang and B. Lee 2013)																	✓			✓	✓								✓
(Shokripour et al. 2013)															✓								✓		✓	✓			✓
(Naguib et al. 2013)										✓	✓																		
(Kevic et al. 2013)														✓															✓
(Jonsson 2013)	✓	✓	✓		✓		✓																						✓
(Banitaan and Alenezi 2013)	✓										✓												✓						✓
(Nguyen et al. 2014)										✓	✓																		✓
(Yang et al. 2014)																	✓			✓	✓								
(Hossen et al. 2014)									✓																✓				
(Hu et al. 2014)												✓					✓												
(Cavalcanti et al. 2014b)									✓			✓	✓													✓			
(Borg 2014)							✓																						✓

continued . . .

continued ...

Table 2.1: A summary of the methods and techniques used in the literature

Study	Machine Learning (ML)							Information Retrieval (IR)							NLP		Network and Graph based			Statistical models				Other methods					
	Naïve Bayes (NB)	Bayesian Network (BN)	Support Vector Machine (SVM)	C4.5	K-Nearest Neighbor (KNN)	Convoluted Neural Network (CNN)	Stacked Generalization (SG) ensemble learner	IR (general)	Latent Semantic Indexing (LSI)	Latent Dirichlet Allocation (LDA)	Topic Modeling (general)	Vector Space Model (VSM)	Term Frequency – Inverse Document Freq. (TF-IDF)	Cosine similarity	NLP (general)	IE (Information Extraction)	Social Network Analysis (SNA)	Tossing Graph (TG)	Markov chains	smoothed Unigram Model (UM)	Kullback–Leibler (KL) Divergence	Developers' vocabulary profile (as expertise matrix)	Term selection / Term weighting	Fuzzy	Bug localization	Rule based	Collaborative Filtering (CF)	Genetic Algorithm (GA)	Others
(Song Wang et al. 2014)																													✓
(Shokripour et al. 2014)															✓								✓						
(Shokripour et al. 2015)													✓		✓														
(Sharma et al. 2015)															✓											✓			✓
(Sajedi-Badashian et al. 2015)																							✓						✓
(Zanjani et al. 2015)					✓								✓	✓											✓				
(S. Jain and Wilson 2016)	✓																												✓
(Dedík and Rossi 2016)	✓		✓										✓																
(W. Zhang et al. 2016a)										✓	✓						✓												
(W. Zhang et al. 2016b)					✓								✓				✓												
(Tian et al. 2016)													✓	✓											✓				✓
(Jonsson et al. 2016)	✓		✓				✓						✓																✓
(T. Zhang et al. 2016)					✓					✓	✓		✓		✓		✓												✓
(Cavalcanti et al. 2016)			✓						✓				✓													✓			
(Anjali et al. 2016)																						✓	✓						
(Zanjani 2016)					✓																				✓				
(Khatun and Sakib 2016)													✓																✓
(Sajedi-Badashian et al. 2016)																							✓						✓
(Park et al. 2016)			✓							✓	✓										✓						✓		✓
(Liu et al. 2016)											✓																		✓
(Karim et al. 2016)																												✓	✓
(S. Lee et al. 2017)						✓					✓																		✓
(T. Zhang et al. 2017)										✓	✓				✓														
(Xia et al. 2017)										✓	✓																		✓
(Goyal 2017)	✓																						✓						✓
(Florea et al. 2017a)			✓			✓																							✓
(Florea et al. 2017b)			✓							✓			✓		✓														
(Khalil et al. 2017)																												✓	✓
(Sun et al. 2017)										✓	✓				✓		✓								✓		✓		
Total	15	6	15	2	7	2	4	4	7	11	11	4	14	3	9	1	10	4	4	2	3	7	7	2	12	4	3	3	32

Jonsson *et al.* (Jonsson 2013; Jonsson et al. 2012) introduced the problem of "team bug-assignment" in which bugs are redirected to one of several available teams. In this formulation, teams are typically stable over time, which effectively makes the concept of "team" very similar to that of a "developer" in other research. In fact, by assigning bugs to the teams their method makes the BA task easier. In other words, the goal of assigning bugs to teams is to decrease the number of candidate assignees, which may be quite large in a large open-source project. Since the members of each team are likely to work on specific sets of modules over time, their expertise is likely easier to predict, again simplifying the

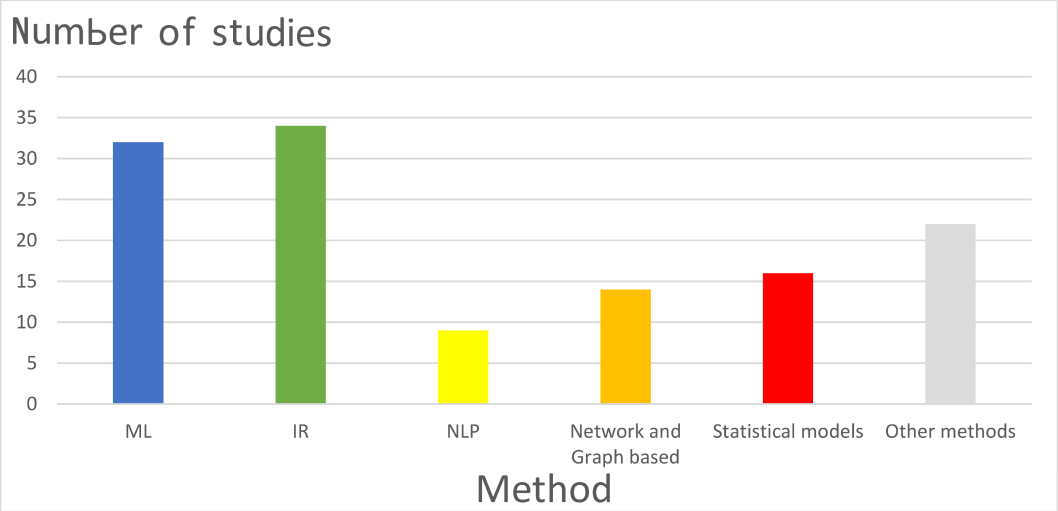

Figure 2.1: Methods used for bug-assignment (overall)

overall BA task.

Instead of maximizing the expertise of the candidate assignee, some methods try to minimize the time-to-fix the bugs (Nguyen et al. 2014; Park et al. 2011; Park et al. 2016). For example, Nguyen *et al.* (Nguyen et al. 2014) propose the construction of a topic model from previously assigned bugs. For an incoming bug, their method predicts the resolution time for each developer (by calculating log normal distribution of combination of three factors; fixer, topic and severity) and ranks the candidate assignees for the new bug report based on this time estimate.

More recently, Liu *et al.* described a method that takes into account both objectives, i.e., expertise maximization and time-to-fix minimization (Liu et al. 2016).

Finally, Karim *et al.* (Karim et al. 2016) proposed a more pragmatic multi-objective problem formulation, assigning multiple bugs to several developers at the same time. Their method assumes competency of developers in different areas (packages) as a function of number of times each developer changed a file in each area in previous attempts for fixing bugs. Using an Estimation By Analogy (EBA) method to obtain an effort estimation for a new bug report, they devised a function of the average time of previous similar bugs as the needed effort for the new bug report. Having estimations of the competency of developers in different areas and the needed effort in each area for the new bug report, they considered two Eclipse sub-projects and developed objective functions to estimate the completion time of "fixing a bug by a developer". Their method aims at maximizing expertise, while also minimizing the total developers' cost and total bug-fixing time, using a genetic algorithm.

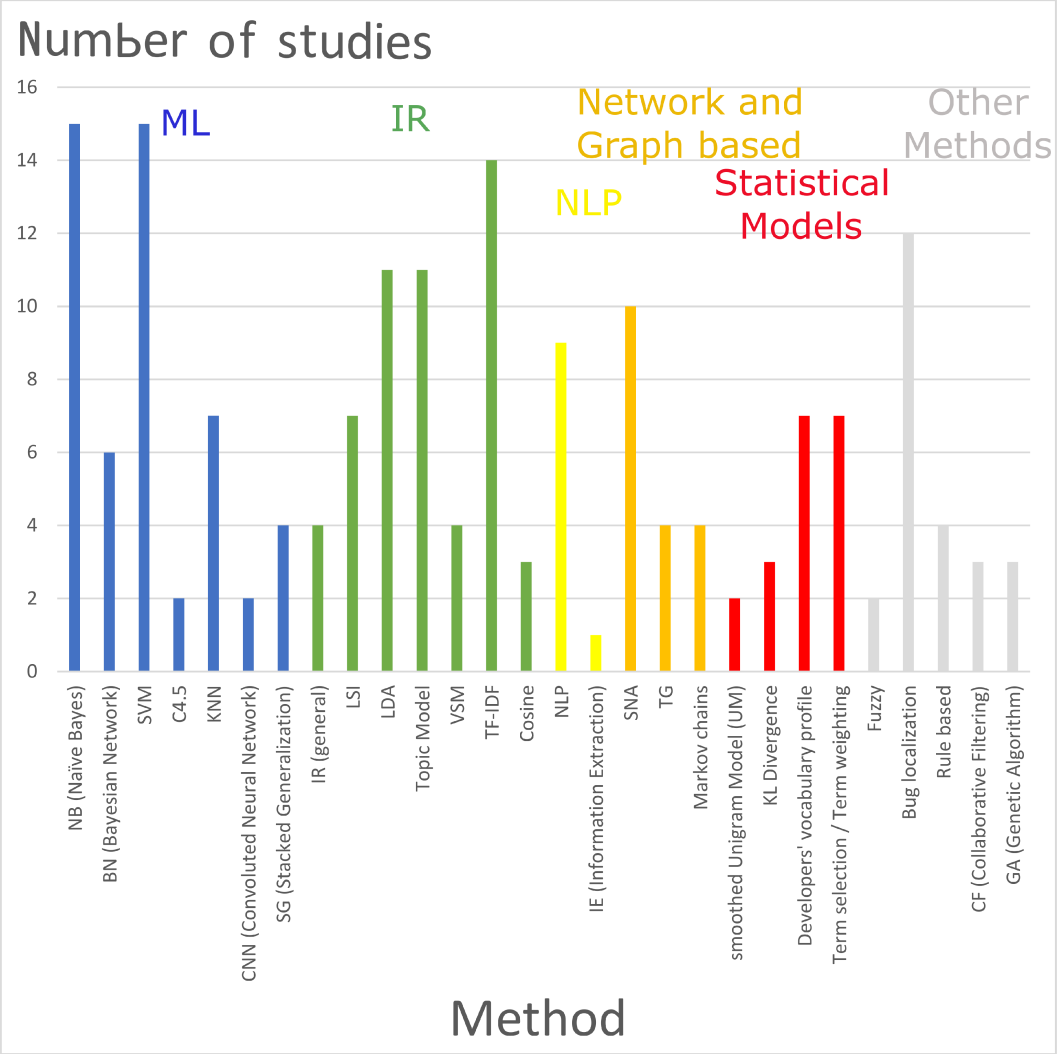

Figure 2.2: Methods used for bug-assignment (details)

Similarly, Rahman *et al.* (Rahmana et al. 2012), Hosseini *et al.* (Hosseini et al. 2012) and Khalil *et al.* (Khalil et al. 2017) proposed multi-objective approaches for BA.

2.2.3 Bug-Assignment Methodologies

As shown in Table 2.1[8], older BA approaches used machine learning techniques that just tried to predict the next assignees by investigating the "relations" between the developers and the previously fixed bug reports and their technical terms. The performance of these approaches depends on the number of previously fixed bugs and they may perform poorly in small projects (Shokripour et al. 2012).

[8]This table shows previous BA research in chronological order.

Then other methods like fuzzy (Tamrawi et al. 2011b) and statistical (Aljarah et al. 2011) approaches appeared. These methods utilized an expertise perspective for the developers regarding keywords. Based on which developer fixed which bug, they gathered an expertise profile for the developers to make a decision upon arrival of new bug reports.

Some researchers used deeper file and meta-data information to relate developers to the new bugs. Examples are location-based techniques like (Hossen et al. 2014; Kagdi and Poshyvanyk 2009; Shokripour et al. 2013). They first predict or find the location of bugs (i.e., methods or classes). Then, based on the available relations between developers and those locations, they predict the best developer who can work on those objects again. Location-based methods usually require bulky Version Control System (VCS) information (e.g., all the changed files in all project branches and commits). That can be the reason why in most methods that use this technique, the authors only run their experiment on a small number of bugs (e.g., a few bug reports up to a few hundreds). Also social network (Park et al. 2011; Xuan et al. 2012) and tossing graph (Bhattacharya and Neamtiu 2010; Jeong et al. 2009) approaches appeared that target more complicated interactions between development objects (e.g., developers, bugs, commits or a combination of them).

The most prevalent methods were Information Retrieval (IR), and then Machine Learning (ML) with usage in 34 and 32 studies respectively. Many of recent studies focused on IR based activity profiling since it usually leads to higher accuracies (Anjali et al. 2016; Shokripour et al. 2013). In the recent years, most of the studies used at least one IR method.

In recent years, some of the studies combined different methods (e.g., combining machine learning and tossing graphs (Bhattacharya and Neamtiu 2010) or combining KNN and IR methods (Zanjani et al. 2015; T. Zhang et al. 2016)). Most recently, the researchers show a tendency toward *social* point of view. For example, (Hu et al. 2014), (W. Zhang et al. 2016b) and (T. Zhang et al. 2016) build a social network of developers to model their relationship with each other or with bugs or even source code components. Also, in our previous research, we proposed a model to recommend developers in Github based on their Stack Overflow contributions and the votes casted by the community (Sajedi-Badashian et al. 2016).

2.3 Select Empirical Bug-assignment Studies

The selected list of 74 papers are all introducing a new methodology for BA. In the next step, we want to focus on a subset of these studies in more detail –based on dimensions of variability and reproducibility criteria. The goal is to enable further researchers compare their results against some reproducible research, as exemplary BA studies. We also elaborate more on the methods, evaluation metrics, knowledge assumptions and design choices of those exemplary studies. To select the new subset of BA studies, we apply the following criteria on those 74 papers:

Inclusion criteria

1. Papers must follow the prevalent formulation of "ranking the developers for the bugs based on appropriateness of a single developer for each bug";
2. Papers must be supported by BA experimental evaluation;
3. Papers must report the experiment results based on major evaluation metrics;
4. Papers should report final results on BA effectiveness (e.g., instead of a comparison of tuning values or data sets);
5. Papers must experiment on full data of developers and bugs (i.e., have no major data filtering); and
6. Experiments must contain relatively high number of developers (˜20) and bugs (˜500) in at least one project, to make the experiment more realistic.

Exclusion criteria

1. Papers which their technique relies on external sources (e.g., Stack Overflow) rather than issue-tracking information are excluded; and
2. Papers that are proposing techniques in other domains with applications in BA are excluded.

Examples of the removed studies are particular BA studies with focus on specific areas like multi-objective BA studies (Karim et al. 2016; Khalil et al. 2017; Rahmana et al. 2012), team BA (Jonsson 2013; Jonsson et al. 2012), market-based bug allocation (Hosseini et al.

2012), component-level BA (Song Wang et al. 2014), time/cost enhancement in BA (Nguyen et al. 2014; Park et al. 2011; Park et al. 2016) and bug report enrichment (T. Zhang et al. 2017) with application in BA. Also we eliminated our previous paper (Sajedi-Badashian et al. 2016) and other studies that work only on a narrow subset of developers (who are shared between version control system and a question answering system like Stack Overflow) (Sahu et al. 2016; X. Zhang et al. 2017).

After considering the above criteria precisely, we obtained **13 papers**[9]. These are great BA examples regarding evaluation, reproducibility and further comparisons. We will focus on these studies in the rest of this chapter as "selected BA studies". Comparing to Table 2.1 that mentions the usage of the techniques by papers in a high level, here, we summarize the techniques used in the selected BA studies in more detail (in chronological order).

Čubranić and Murphy developed a method that uses a **Naïve Bayes** classifier to assign each bug report (a "text document" consisting of the bug summary and description) to a developer (seen as a topic category or the "class") who actually fixed the bug (Čubranić and Murphy 2004). When a new bug report arrives, it uses the textual fields of the bug to predict the related class (e.g., developer).

Canfora and Cerulo proposed an **Information Retrieval (IR)** approach (Canfora and Cerulo 2006). It assumes that the developers who have solved similar bug reports in the past are the best candidates to solve the new one. So, it considers each developer as a document by aggregating the textual descriptions of the previous change requests that the developer has addressed. Given a new bug report, it uses a probabilistic IR model and considers its textual description as a query to retrieve a candidate from the document (developer) repository.

Jeong *et al.* introduced another approach that captures tossing probabilities between developers from tossing history of the bugs (Jeong et al. 2009). Then, it makes a **tossing graph** of developers based on Markov Model. In this graph, the nodes are developers and the weight of the directed edges show the probability of tossing from one developer to the other. Finally, for predicting the assignees of a bug, it first produces a list of developers using a machine learning method. Then, after each developer in this list, adds the neighbor developer with the most probable tossing weight from the graph.

Matter *et al.* employed the **Vector Space Model (VSM)** to create an assignee rec-

[9]Three studies ((Bhattacharya and Neamtiu 2010), (Tamrawi et al. 2011b) and (Cavalcanti et al. 2014b)) are conference version of other studies by the same authors, so we merged them into one row in the table.

ommender (Matter et al. 2009). It considers the source code contributions and the previous bug fixing as evidence of expertise and builds a vocabulary of "technical terms". It builds this vocabulary from the technical terms the developer used in the source code (captured by diff of the submitted revision) or commit messages. Also adds to this vocabulary the technical terms mentioned in the previous bug reports assigned to the developer. As a result, the developer's expertise is modeled and captured as a term vector. Given a new bug report, it calculates the **cosine distance** between the new bug report's term vector and the developers', and sorts them based on this score and reports the top ones.

Tamrawi *et al.* devised a **fuzzy** method toward bug assignment (Tamrawi et al. 2011a; Tamrawi et al. 2011b). It computes a score for each "developer - technical term" based on the technical terms available in previous bug reports and their fixing history by the developers. Considering a new bug report, it calculates a score for each developer as a candidate assignee by combining his/her scores for all the technical terms associated with the bug report in question. Then sorts the developers based on this score. The newer version, (Tamrawi et al. 2011a), also applies a term selection method to reduce noise data and speed up the algorithm. It extracts the top k terms that are most related with each developer. Then, when calculating the fuzzy score for each developer, just considers those selected terms and ignores other terms.

Bhattacharya and Neamtiu created a system similar to (Jeong et al. 2009), that improves the graph by adding labels for each edge (Bhattacharya and Neamtiu 2010; Bhattacharya et al. 2012). The label of each edge indicates product, component and latest activity date. Then, it recommends three developers based on a machine learning method. Also, the tosee ranking uses the graph labels (tossing probability, product, component and last activity date of the developer) for each of the top two developers in this list to recommend a substitute (called tosee) and enhance this list to a top-5 recommendation.

Shokripour *et al.* used **bug report localization**, **Information Extraction (IE)** and **Natural Language Processing (NLP)** in their BA technique (Shokripour et al. 2012). It first applies IE and NLP techniques on file-related components (e.g.,, commit messages and their comments, plus their related source code elements including phrases in methods and classes) and also the bug reports and elicits their important phrases. When a new bug arrives, it first estimates the location of the new bug (i.e.,, the files that should be changed to fix the bug) by comparing the important phrases in the bug report and other file-related components as mentioned above. Then, recommends the developers with the most activity

regarding those files –according the historical data.

Zhang *et al.* proposed a new approach by combining SNA and machine learning methods. It builds a **heterogeneous social network** of developers, bugs and their comments, components and products (W. Zhang et al. 2016b). Then, it selects the top k similar bug reports to the given new bug report by using **KNN** classification, **Cosine similarity** and **tf-idf**. After that, extracts the list of commenters of those k bug reports as the narrowed list of "candidate developers" and obtains the score of each developer by calculating overall **heterogeneous proximity** of each developer with all other "candidate developers" on component and product of the new bug report, using the previously built network.

Cavalcanti *et al.* introduced a semi-automatic approach that combines rule-based expert system (RBES) with information retrieval (IR) methods (Cavalcanti et al. 2014b; Cavalcanti et al. 2016). First, it summarizes the previous bug reports and extracts some simple rules based on meta-data (e.g., component of a bug, or being critical) or keywords in the bug report, and the ground-truth assignee. These rules can decide on some circumstances which developer should be assigned to a new bug. In the second phase, if the simple rules cannot recommend any developer for the new bug report, uses the machine learning SVM classifier to consider previous assignments to developers and assign a label (developer) to it (e.g., the developers who fixed similar bug reports are most likely to fix the new one).

Finally, Sun *et al.* devised another IR method that analyzes and extracts the commits related to the issue request (Sun et al. 2017). This is done by obtaining cosine similarity between new bug report and historical commits. Those commits are considered relevant commits and the changed source code is considered relevant source code. Authors of those commits are considered as candidate developers to fix the new issue. Finally, it uses Collaborative Topic Modeling (CTM) to give a score to each of those developers (based on shared keywords in their relevant source code and the new issue) and sort and recommend them to fix the new issue.

2.3.1 Knowledge Assumptions of Bug-assignment Methods

There are a wide range of information that have been used in various BA research. This information is used in some way to relate a developer to a bug. Table 2.2 shows a summary of the information used by the selected BA research.

According to this table, older approaches mostly rely on textual information (e.g., title and description) of the bug reports as clues for indication of expertise of developers

and matching with new bug reports (Canfora and Cerulo 2006; Čubranić and Murphy 2004; Tamrawi et al. 2011a). Many of the newer research used variety of meta-data fields (e.g., component, product, severity and operating system) to address some of the limitations (Bhattacharya et al. 2012; Cavalcanti et al. 2016; W. Zhang et al. 2016b).

As mentioned earlier, in recent years, many researchers tried to combine different methods to achieve more accurate results. Hence their combined methods need different types of information –as their various methods demand. For example, (Sun et al. 2017) considers commit messages and some meta data. In addition, it extracts and deals with source code and file changes. This sometimes makes a huge demand on the needed data and processing.

Extra information of the meta-data can help obtaining higher accuracies in some cases, but still text-based methods are the most effective techniques used (Shokripour et al. 2015; Sun et al. 2014) and those meta-data based approaches are sometimes difficult to setup; for example, some need to know who maintained different pieces of code in the IDE (Hossen et al. 2014), or who interacted online with whom in different setups (W. Zhang et al. 2016b). As a result, the textual elements remain the most prevalent and effective information used for BA.

2.3.2 Metrics for Evaluation of Bug-assignment Research

We reviewed the 13 selected papers to identify the metrics they used for evaluation, and, cross-validation of their approaches against real bug data. The metrics they used are *top-k*

Table 2.2: A review of the used information for bug assignment in selected BA studies

Method	Bugs' info		Developers' expertise info				
	Title + description	Meta	Bug fixing; title / description	Being a committer	Tossing history	Meta	Changed code
(Čubranić and Murphy 2004)	✓		✓				
(Canfora and Cerulo 2006)	✓		✓				
(Jeong et al. 2009)	✓		✓		✓		
(Matter et al. 2009)	✓		✓	✓			✓
(Tamrawi et al. 2011a; Tamrawi et al. 2011b)	✓		✓				
(Bhattacharya and Neamtiu 2010; Bhattacharya et al. 2012)	✓	✓	✓		✓	✓	
(Shokripour et al. 2012)	✓		✓	✓			
(W. Zhang et al. 2016b)	✓	✓	✓	✓		✓	
(Cavalcanti et al. 2014b; Cavalcanti et al. 2016)	✓	✓	✓			✓	
(Sun et al. 2017)	✓	✓		✓		✓	✓

accuracy, *precision @k*, *recall @k* (all with k=1, 5 and 10), *MRR* and *MAP*. These metrics (except MAP) were the most frequently used metrics for reporting the results in the whole 74 papers as well. MAP, however, was used only in (W. Zhang et al. 2016b). The reported results of the selected papers are shown in Table 2.3. We also reported two important design choices of these studies in the Table; number of developers and number of bugs they experimented on. Those choices can affect some of the evaluation metrics and are needed to mention in BA experiments.

Table 2.3: The evaluation metrics and other design choices of the selected research

Method / Project	#devs	#bugs	**Top1 (%)**	**Top5 (%)**	**Top10 (%)**	**P@1 / r@1**	**p@5 / r@5**	**p@10 / r@10**	**MRR**	**MAP**
(Ćubranić and Murphy 2004)										
Eclipse	162	15,859	30.00							
(Canfora and Cerulo 2006)										
Mozilla	637	12,447				- / 0.12	- / 0.21	- / 0.24		
KDE	373	14,396				- / 0.50	- / 0.10	- / 0.12		
(Jeong et al. 2009)										
Eclipse	?	46,426		77.14						
Mozilla	?	84,559		70.82						
(Matter et al. 2009)										
Eclipse	210	130,769				0.34 / ∼0.27	∼0.16 / ∼0.59	∼0.10 / 0.71		
(Tamrawi et al. 2011a; Tamrawi et al. 2011b)										
Firefox	3,014	188,139	32.1	73.9						
Eclipse	2,144	177,637	42.6	80.1						
Apache	1,695	43,162	39.8	75.0						
Net Beans	380	23,522	31.8	60.4						
FreeDesktop	374	17,084	51.2	81.1						
GCC	293	19,430	48.6	79.2						
Jazz	156	34,220	31.3	75.3						
(Bhattacharya and Neamtiu 2010; Bhattacharya et al. 2012)										
Mozilla	?	549,962	27.67	77.87						
Eclipse	?	306,297	32.36	77.43						
(Shokripour et al. 2012)										
Eclipse	?	?				- / ∼0.32	- / ∼0.71			
Mozilla	?	?				- / ∼0.27	- / ∼0.48			
Gnome	?	?				- / ∼0.10	- / ∼0.45			
(W. Zhang et al. 2016b)										
Mozilla	∼874	74,100							0.28	0.44
Eclipse	∼544	42,560							0.28	0.56
Ant	∼203	763							0.35	0.36
TomCat6	∼79	489							0.35	0.37
(Cavalcanti et al. 2014b; Cavalcanti et al. 2016)										
New SIAFI - A	70	781	31.40							
New SIAFI - B	70	1031	22.00							
(Sun et al. 2017)										
JEdit	123	?	28.0	60.1	79.8					
Hadoop	82	?	8.5	30.1	50.3					
JDT-Debug	47	?	14.4	46.6	66.4					
Elastic	661	?	13.6	43.6	75.2					
Libgdx	345	?	22.0	51.3	69.6					

This Table is useful for further comparisons of efficiency of new BA methods against the previous research for two reasons. First, it gives the researchers flexibility (in terms of choosing metrics) in comparing their results against some other research. Then, since

these studies have met all the inclusion criteria we set for the reproducible research (e.g., the projects and the number of bug reports they tested on are big enough and they did not do major filtering on developers or bug reports), the comparison of the results of a new research against these research would give sufficient feedback.

In the next section, we discuss the suitability of these metrics for reporting efficiency of BA research experiments.

2.4 A Discussion of Bug-assignment Evaluation Metrics

The evaluation metric can affect the outcome of the method; In BA research, to evaluate their methods, researchers evaluate the results of their approaches with some real BA data. According to our survey on the 74 papers, the mostly used evaluation metrics are ***top-k* accuracy**, ***precision @k*** and ***recall @k***. But those metrics are not representative enough for evaluating the *efficiency* of the BA approaches. Elaborating on this statement, in this section, we investigate about the best "evaluation metric" for bug-assignment research. The metrics used in the literature are as follows:

***Top-k* accuracy** (k usually is considered 1, 5 or 10) is a metric that considers the number of ground-truth assignees available in a few top recommended ranks. This metric does not interpret the behavior of the algorithm for other ground-truth assignees (that are ranked in locations k+1 and further). In addition, it does not differentiate between the first and subsequent ranks. Consider the following example; assume that there is only one assignee per bug and there are 100 bugs to test. Also assume that method A recommends the ground-truth assignee in the 2nd, 6th and 11th rank in respectively 10, 20 and 70 test cases (i.e., bug reports). So, the *top-1*, *top-5* and *top-10* accuracies will be respectively 0%, 10%, and 30%. Similarly, assume that method B recommends the ground-truth assignee in the 5th, 10th and 15th rank in respectively 10, 20 and 70 cases. So again the *top-1*, *top-5* and *top-10* will be respectively 0%, 10%, and 30%. While method A performs much better than method B, the *top-k* metrics show the same accuracy for the two methods. Also, they both miss 70 test cases that the ground-truth assignee is ranked after the 10th location. Finally, *top-k* accuracy is highly affected by average number of ground-truth assignees per bug. The more ground-truth assignees are assumed for a bug, the chance of finding at least one of them in the top k recommended list will be higher.

Some metrics are coupled with other metrics and may not be interpreted separately. For example, ***precision @k*** needs to be interpreted along with ***recall @k*** (again, according to the summarized studies, k usually is considered 1, 5 or 10). A fair evaluation demands to compare *both* factors at a time against the results of other studies, which is not straightforward. As an example, Anvik *et al.* 's 0.64 precision @1 (Anvik et al. 2006) in Firefox project is the highest precision @1 in all the previous studies. At the same time, their recall @1 in all their projects is too low (not higher than 0.05). In fact, precision needs to be considered with recall at the same time, when comparing against other methods (note that this makes the comparison more difficult). In addition, like *top-k* accuracy, these two metrics are highly dependent on the average number of ground-truth assignees per bug. As an example, in a previous study by Anvik, *et al.*, the average number of assignees per bug is too high (between 10 and 30 for three projects) (Anvik et al. 2006). So, precision is affected by the average number of ground-truth assignees positively. Finally, like *top-k* accuracy, precision @k and recall @k do not consider the ground-truth assignees ranked after location k in the list.

F-measure is equal to harmonic mean of precision and recall. It combines precision and recall into one metric (Manning et al. 2008; Xu et al. 2010) and is better for comparison purposes. However, since F-measure –like precision and recall– is a set-based measure (that is computed by considering unordered set of *top k* recommendations), it is not suitable for evaluating ranked retrieval results (Shani and Gunawardana 2011). Since ordering of the developers is important, we need a metric that reflects ranking of reported ground-truth assignees. In other words, the BA algorithm recommends a *ranked list* of developers (to the project manager), in which higher ranks are more important than the lower ones. F-Measure does not differentiate between those ranks. Moreover, F-Measure is designed for IR problems in which there are many relevant documents. Unlike those problems, in BA there are only a few documents (e.g., the ground-truth assignees) per bug to retrieve. In the big data set we provided in this study, the average number of assignees per bug is 1.4 (minimum and maximum are 1 and 11 assignees per bug respectively).

Mean Reciprocal Rank (MRR) of the ground-truth assignee tries to capture the importance of the higher ranks, which is a positive point. It is equal to the mean of "reciprocal rank of the highest-ranked assignee of the bug" over all the bugs (or, harmonic mean of the ranks of the highest-ranked assignee of each bug over all the bugs). It indicates "how far down the list of recommended developers for a bug one should proceed to find a ground-truth assignee"? However, it ignores the rank of other assignees for each bug. Also, it is highly

affected by the average number of ground-truth assignees per bug; the chance of having a ground-truth assignee in any recommendation, even a random one, increases when the average number of ground-truth assignees increases.

Mean Average Precision (MAP) is a standard metric which is widely used in text retrieval (Manning et al. 2008) but not much in BA. It provides a single-figure quality measure, representing both precision and recall, with good discrimination and stability properties (Shi et al. 2012). MAP is calculated across all different recall levels and equals to the average area under the precision-recall curve (Manning et al. 2008; Shi et al. 2012). It is a good performance metric when a short list of items is provided to the developer (i.e., the triager) (Shani and Gunawardana 2011; Shi et al. 2012). MAP is a single *effectiveness* metric that measures how *all* the relevant documents are ranked highly (close to top of the list). It considers all the ranks of all the relevant items for all the queries. MAP will be equal to MRR if there is only one correct answer (ground-truth assignee) for each query (new bug report) (Alipour 2013). Generally, to calculate MAP over all queries, first, we need to calculate Average Precision (AP) for each query: we calculate AP at several points at which there is a correct recommendation, and then we calculate the mean of AP over all queries. In bug assignment, each (new) bug is considered a query and each developer in the project is equivalent to a document. The *ground-truth assignees* are actually the *relevant documents*.

The question then becomes "what is the best metric to evaluate BA effectiveness"? We propose four criteria based on which we should select the most meaningful metric:

1. **The interpretation of the evaluation metric should be independent of other evaluation metrics**:

   This supports easier comparison of a study against other approaches. MAP, MRR, *f-measure* and *top-k accuracy* are single-figure metrics and can be interpreted separately. However, precision and recall should be reported and interpreted together (usually when precision is increased, recall is decreased).

2. **The rank of all the ground-truth assignees should be taken into account**:

   The number of ground-truth assignees can be different from bug to bug but can definitely be more than one. *Top-k accuracy*, *precision @k*, *recall @k* and *f-measure* are all bound to a chosen threshold, k. In other words, they only count the ground-truth assignees recommended in the top k ranks and ignore the others. Also, MRR only considers the rank of the first assignee, but MAP considers all of them. In fact, it is the

only metric that considers rank of "all" the ground-truth assignees for each bug. Note that using MAP, there is no need to consider fixed thresholds (e.g., @k). It captures and considers the rank of *all* the ground-truth assignees, no matter in which rank they are recommended.

3. **Errors in the higher ranks are worse than errors in the lower ranks**:

   This is because the triager usually checks the higher ranks in the list and may not proceed to the last one. Again, in *Top-k accuracy*, *precision @k*, *recall @k* and *f-measure* metrics, it does not matter where in the first k ranks the ground-truth assignee is. MAP and MRR, however, are affected by a hit in the first ranks much more than the next ranks. In a sense, they penalize the mistakes in the first ranks more than the next ranks. Note that while MRR stops at the first ground-truth assignee, this penalization continues for MAP, for every incorrect guess until the last ground-truth assignee in the list.

4. **The evaluation measure should be robust to the number of ground-truth assignees**:

   Assume that there are 10 ground-truth assignees (correct answers) per bug. The chance of having at least one of them in the *top-10 accuracy*, would be very high, as compared to the case when there is only one assignee per bug. MAP is affected the least from the average number of ground-truth assignees in the project –unless the average number of ground-truth assignees is too big, i.e., close to the number of developers in the project (developer community).

With the mentioned qualities, **MAP** is the best metric to report for BA results. It satisfies all the four above-mentioned criteria. The other metrics cannot reflect the efficiency of BA regarding at least one of the above aspects. Note that when experimenting on several projects, in addition to reporting MAP per project, reporting a final MAP over all the bug reports of all the projects makes the comparison against other methods more straightforward.

MAP is widely used in evaluating methods in IR problems (e.g., document retrieval (Manning et al. 2008)) and other software engineering problems (e.g., bug-report de-duplication (Aggarwal et al. 2017; Alipour 2013)). Interestingly, it is rarely used for BA evaluation; the only study out of all the 74 papers of our survey that reported MAP was (W. Zhang et al. 2016b). As a result, although reporting MAP would be convincing enough, for comparison

with previous approaches (e.g., meta analysis), it is still recommended to report the mostly used metrics as well as MAP (*top-k accuracy*, *Precision @k* and *recall @k* for k =1, 5 and 10 as well as *MRR*). For this reason, we extracted all of these metrics reported in the 13 selected papers, which can be used for further research (see Table 2.3).

Deriving from previous IR notations (Cormack and Lynam 2006; Manning et al. 2008; Shani and Gunawardana 2011; Shi et al. 2012), having n bug reports, we reformulate the following definition of MAP considering BA notations:

$$
\begin{aligned}
MAP &= \frac{\sum_{i=1}^{n} AP_i}{n} \\
AP_i &= \frac{\sum_{k \in R_i} p@k}{\#\ of\ real\ assignees\ for\ bug\ i} \qquad (2.1) \\
R_i &: \{ranks\ of\ real\ assignees\ in\ the\ recommended\ list\ of\ developers\ for\ bug\ i\} \\
p@k &= \frac{\#\ of\ real\ assignees\ in\ the\ top\ k\ recommended\ developers}{k}
\end{aligned}
$$

AP_i is the Average Precision for bug i and R_i is the set of all the ranks of the real assignees in a recommended ranking for bug i. Figure 2.3 shows an example of MAP for three assignments.

2.5 Dimensions of Variability in Bug-assignment Empirical Studies

In addition to the metrics, there are two dimensions that can affect the measurement and evaluation of the results. We found that the choices of "ground-truth assignee" and "developer community" are very important in this regard and they vary a lot in previous work. To better understand these dimensions, we show an example bug report in Figure 2.4. The Figure depicts a sample bug report[10] in **Travis-ci**, one of the most popular Github projects we studied in our previous study (Sajedi-Badashian et al. 2016). On the page of each bug, Github shows the work around that bug. For example, the bug report in this figure has a

[10]https://github.com/travis-ci/travis-ci/issues/257

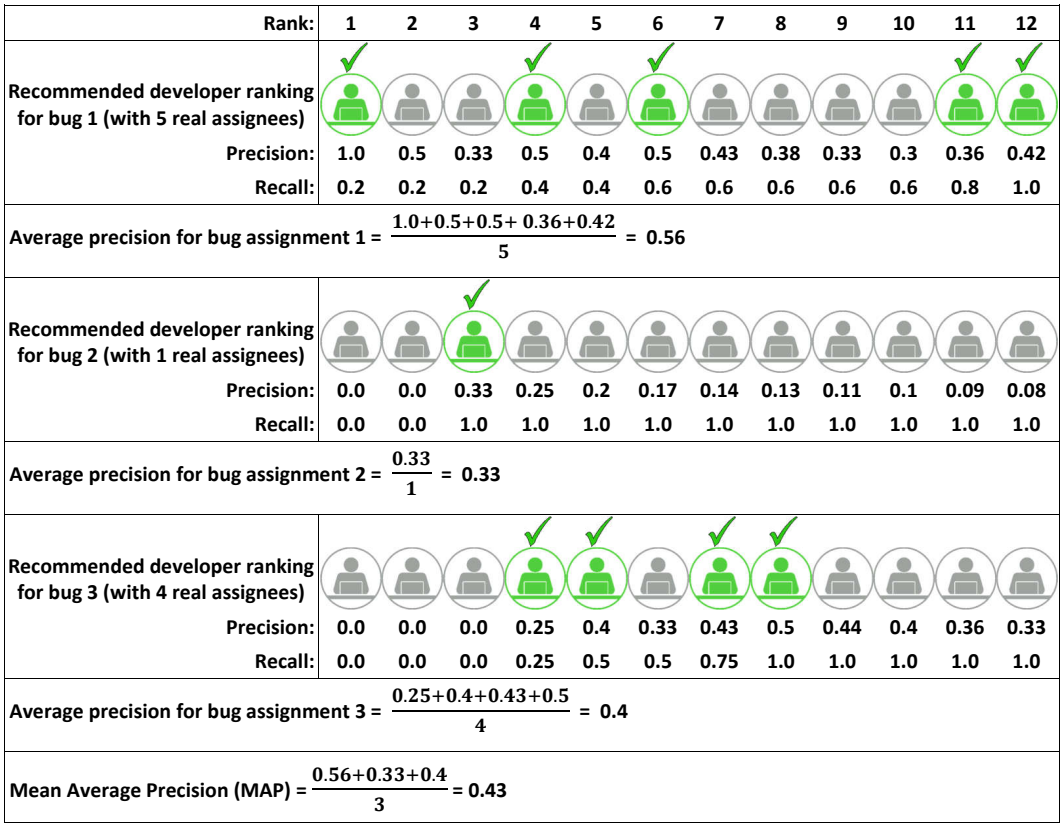

Figure 2.3: An example of use of MAP for bug-assignment over three bugs

title, description, meta-data elements (reporting date, reporter, labels, etc.) and the interactions around the bug –e.g., comments, being assigned or closed. In case the bug is referenced from a commit or other actions, then an entry will be shown in the list of interactions, with links to those actions. These types of information are used for extracting developers' expertise and the ground-truth assignees as we will discuss later.

2.5.1 Ground-truth assignee

In almost all the previous research which proposed a new BA approach, before recommending the assignees, the authors used some rule of thumb to identify at least one developer for each bug, as the real/actual assignee (Canfora and Cerulo 2006; Čubranić and Murphy 2004; Jeong et al. 2009; Matter et al. 2009; Sun et al. 2017; Tamrawi et al. 2011b). Then they compared their recommendations against these ground-truth assignees to measure the effectiveness of their approach and report it using some metrics. The definition of "ground-truth assignee" (ground truth) is critical in validating and understanding the goodness of a method, and, comparatively analyzing the merits and shortcomings of alternative methods.

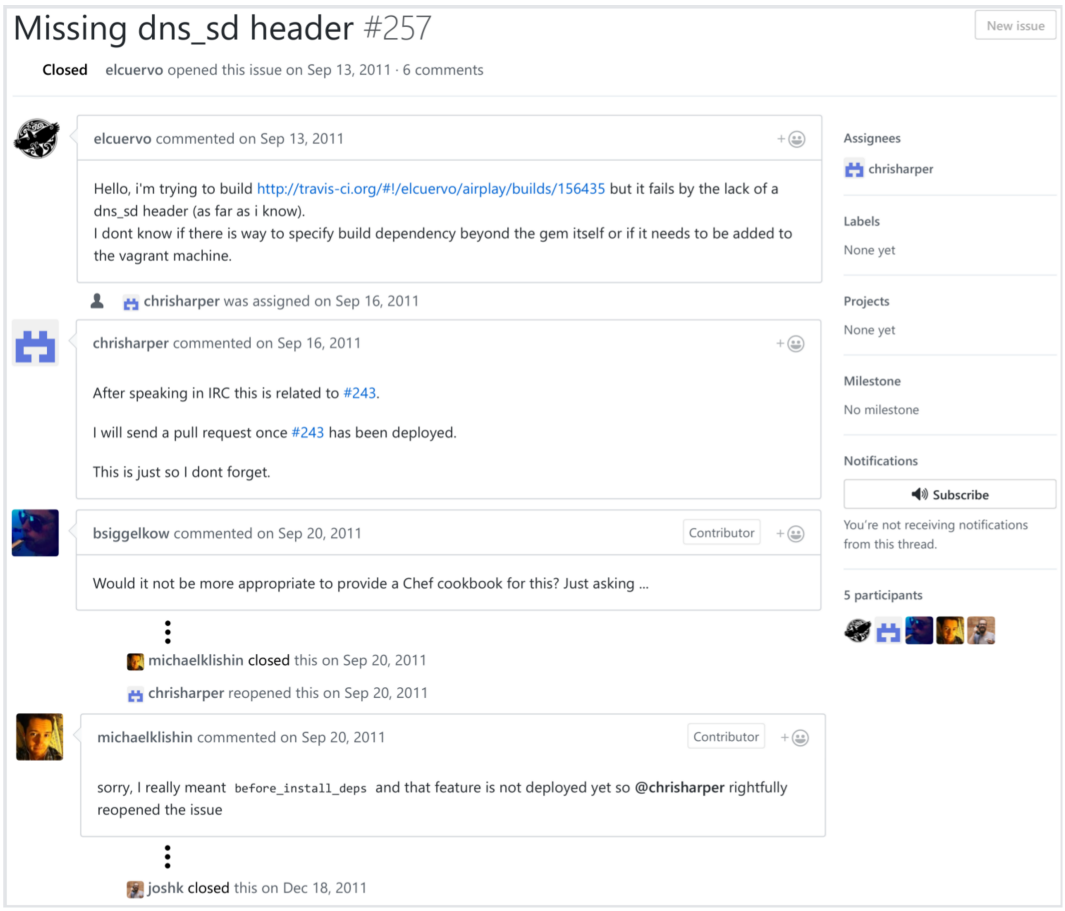

Figure 2.4: A sample bug report (#257) in *Travis-ci*

Consider the following example as the effect of this ground truth on the evaluation results. Anvik *et al.* (Anvik et al. 2006) adopted some heuristics to assume developers as assignees of a bug. Based on those heuristics, they obtained (on average) 30, 10 and 12 assignees per bug report in their three projects, Firefox, Eclipse and gcc respectively. This is the highest among all the studies we observed in our survey. In Firefox, they obtained 0.64 precision @1, which is again the highest among all the previous studies of all the time (even till now, after more than 10 years) regarding precision @1 metric. In Eclipse and gcc, their precision @1 decreased to 0.40 and 0.06 respectively. One reason to the high precision in Firefox was the high number of ground-truth assignees per bug (i.e., 30 in average), which is almost three times their other two projects. Based on our observations in almost all the previous studies, each bug on average has only one, or up to a few ground-truth assignees[11]. One reason that this number varies from one research to another, is that they adopted different "definitions

[11]This does not include the bugs without any assignee. To the best of our knowledge, in all previous studies, those bug reports are filtered since they lack information (ground-truth assignee) for evaluation.

of *ground-truth assignee*". So, the definition of *ground-truth assignee* adopted by a research experiment can highly affect its results. Without a clear definition of *ground-truth assignee*, fair judgment or reproduction of a research would be problematic.

In our survey on BA research, we found that there are various definitions of ground-truth assignee in different studies, and that there is no unanimity in using/addressing this definition. In other words, the definition(s) of "ground-truth assignee" adopted by different studies is not consistent over those studies. This can make the process of evaluating the effectiveness of a method more subjective and obstruct reproducibility of the research. We will show later, that the adopted definition of ground-truth assignee can affect or even bias the reported results.

We surveyed BA studies, and, based on interactions between the developers and development objects (e.g., bugs or commits) extracted the following definitions of *ground-truth assignee*:

- *Type 1 (T1); AUTHOR: The author of a commit referencing a bug number as resolved.* In this case, the author, who is the original developer of the code includes in the commit message a mention that the newly contributed code fixes a specific bug in the project. In the Github page for the bug report, this reference is shown as an event of type *commit* in the bug's life cycle.

- *Type 2 (T2); COAUTHOR: A developer (other than the original author of the committed code), who actually commits the code and references a bug as resolved.* In this case, the *coauthor* is different than the *author*. Typically, the *coauthor* has some higher-level permissions. In some cases, it indicates an additional code reviewer role examining and confirming that a piece of code fixes a bug. For example, the project maintainer who merges the patch or last applies it (approves it), the one who accepts a pull request, or the one who does the rebase is called *coauthor*. In all these examples, this person is different from the one who actually writes the code (i.e., the *author*).

  Note that this type of assignee has not been explicitly studied before. Previous research either ignored T2 assignees or considered them under T1. The fact is, however, that this is a different indication of bug-fixing contribution. Therefore, we believe that it is worth examining it separately.

  Again, like T1, in the Github page for the bug report, this type of reference / fix the bug is shown as an event.

- *Type 3 (T3); ADMIN_CLOSER: The developer who closes the bug.* If a developer decides to close a bug, one can assume that they know enough about that bug and may be competent to fix it. In some projects, any developer can close a bug –or re-open it later; however, in many big projects, this privilege is reserved for higher-level or administrative roles that usually the *core* developers have. In some cases, these developers review the code and the proposed bug as soon as it is reported, and then bring the bug to the attention of appropriate programmers in the team. In the sample bug report in Figure 2.4, developer "michaelklishin" closes the bug and is a T3 assignee. Note that a bug may be re-opened, worked on by a few developers, and closed again several times. This is happened in the example of Figure 2.4; somebody opened the bug again, and developer "joshk" closed it again. In these cases, any developer who closes the bug is considered as a *T3 assignee*.

- *Type 4 (T4); DRAFTED_ASSIGNEE: The developer tagged as "assignee" when the bug is closed.* In Figure 2.4, developer "chrisharper" is assigned to the bug and is considered a T4 assignee. This developer is shown in the "Assignees" section, at the top of the right panel for each bug. At each point in time, several people can be assigned/unassigned/reassigned to a bug, either by their own initiative or by other project members. The developer who is tagged *assignee* at the time when the bug is closed is assumed to be the T4 assignee of the bug. Note that just being tagged as an "assignee" should not be considered as evidence of relevant expertise. For example, if a developer is assigned to a bug, but finds that cannot fix it, then they may opt out or may be un-assigned. As another example, a developer may be the tagged assignee while the bug remains open forever; we do not consider these cases *useful* and simply ignore them.

- *Type 5 (T5); ALL_TYPES: The union of all the above four types including all sorts of work toward fixing the bug.* This definition is useful in that it leads to a broad and realistic formulation of the BA problem. It includes code authorship and co-authorship, administrative bug manipulation and being drafted as assignee.

Most previous research used the definitions of T1, T2[12], T3 and T4 or a limited combination of them for the ground-truth assignee. No previous research, however, used the

[12]In some of the previous research it is not clear if they are using just T1 or a combination of T1 and T2. Due to lack of a clear separation in these cases, we assumed they considered both T1 and T2 in Table 2.4

Table 2.4: The types of assignment used in previous studies, varied from T1 to T4

Method	Assignee types	Developer community
(Čubranić and Murphy 2004)	T3, T4	The ground-truth assignees of the selected bugs for the experiment (162 developers) are considered members of "developer community".
(Canfora and Cerulo 2006)	T1, T2	The ground-truth assignees of the selected bugs for the experiment (373 and 637 developers in two projects)
(Jeong et al. 2009)	T4	The ground-truth assignees of the selected bugs for the experiment (number of developers is not mentioned)
(Matter et al. 2009)	T1, T2, T4	The ground-truth assignees of all the bug reports (210 developers)
(Tamrawi et al. 2011a; Tamrawi et al. 2011b)	T4	The ground-truth assignees of the selected bugs for the experiment (between 156 and 3,014 developers in 7 projects)
(Bhattacharya and Neamtiu 2010; Bhattacharya et al. 2012)	T3	The ground-truth assignees of all the bug reports (number of developers is not mentioned)
(Shokripour et al. 2012)	T1, T2	The ground-truth assignees of the selected bugs for the experiment (number of developers is not mentioned)
(W. Zhang et al. 2016b)	T3	The ground-truth assignees of all the bug reports except the developers who were assigned to only one bug (In total, between 70 and 874 developers in four projects)
(Cavalcanti et al. 2014b; Cavalcanti et al. 2016)	T1, T2	The ground-truth assignees of the selected bugs for the experiment (70 developers)
(Sun et al. 2017)	T3, T4[13]	The ground-truth assignees of all the bug reports (between 47 and 667 developers in 5 projects)

comprehensive combination of them, i.e., T5 (see Table 2.4). Note that these categories are inclusive of similar definitions across the existing literature which use other bug-tracking and version-control systems rather than Github. But no previous research has used the union of those types (e.g., T5 as we defined above).

Note that ideally, there can be another type of ground-truth assignee. The project manager can determine a list of candidates who would be proper developers to fix each bug. These developers might never have worked toward fixing that bug (due to high workload, unavailability or other reasons), but still would be included in the ground truth. Although this definition would be the ideal ground truth, it is very expensive to produce such a list for thousands of bugs in a big project. So, we just ignore it and only consider the five main types of assignee (e.g., T1 to T5) as we discussed above. These five types of assignee can be extracted from the issue-tracking system directly and easily.

[13]The type of assignee in this paper was not directly mentioned in the text, but it is conceived indirectly from the text.

2.5.2 Developer community

We define the *developer community* as the set of developers who are considered potential assignees in a project at any time. BA researchers try to sort and recommend the top developers in this set, based on their competence to fix a given bug. Just like how adopting a narrowed "ground truth" can impact the reported results, the size of this community affects the accuracy of the proposed BA methods. The more limited (and smaller) the developer community becomes, the easier the prediction of actual ground-truth assignee will be. For example, predicting the ground-truth assignee from a set of 10 developers is easier than a set of 500 candidates.

It is hard to obtain a unanimous definition for developer community. Github, through its comprehensive APIs, gives a list of collaborators to the project (Github 2017b). Some of them are outside collaborators, who make contributions through pull requests (that should be reviewed before being approved as project contribution). These people do not have access to the organization but can have controlled contribution towards the project. Direct collaborators are appointed by the project managers and can have limited or full write access to the repository or organization. Also, there are organization members who have access to all the projects of the organization through team membership or other default organization permissions (note that in Github each organization is a virtual organization and can include several projects in it). Finally, there are the project managers and organization owners.

The project and organization managers can give different access levels to people regarding type of their collaboration. Even if a developer who is appointed by the managers as a project member (collaborator), does not contribute in the project (i.e., does not do any commits), he still is a project member. The fact is that the project managers found this developer a good candidate for contributing in the project. So, he should be considered as a possible assignee.

Some previous research considered all the project members as their developer community (Khatun and Sakib 2016; Sajedi-Badashian et al. 2015; Sajedi-Badashian et al. 2016). This is the most comprehensive definition for developer community. Some others took a subset of the developers in the project (e.g., the committers) as the developer community (Hossen et al. 2014; Kagdi et al. 2008). Many others considered the set of previous assignees as developer community (Anvik et al. 2006; Bhattacharya et al. 2012; Čubranić and Murphy 2004; Tamrawi et al. 2011a). Also, there are many others that did not have a clear definition for it (M. M. Rahman et al. 2009).

Regardless of the general definition for developer community, many of previous research have been filtering the community, to remove less-active developers and subtle bugs assigned to them. This is another restriction on the developer community in the data set and we discuss it later, along with the effect of developer community on the evaluation of the results.

2.6 Experiment setup and Data Set

To investigate the effect of adopted definition of "ground-truth assignee" and "Developer community" on the evaluation of BA research, we perform a set of experiments. We use the findings of those experiments to answer the research questions of our study.

In our experiments, we use the IR notation where the description of the new bug is the query and the developers' profiles (concatenated text of all the previously fixed bug reports by each developer) are the documents. We use the *tf-idf* composite weighting for giving a score to each document (developer) for the given, new query (bug report). This method or its variations were used previously to emphasize on specific keywords and de-emphasize the common terms, and shown to work well on textual data (Khatun and Sakib 2016; Shokripour et al. 2015; T. Zhang et al. 2016; W. Zhang et al. 2016b).

We applied the baseline *tf-idf* method to sort the developers for the given bug report. At each point in time, we considered each developer a document including the textual elements of all the bug reports assigned to that developer up to that time. We considered the bug report's title and description, plus the main languages of the project (after removing stop-words) as the contents of these documents. We assumed main language of a project as any programming language that contains at least 15% of the lines of code of that project.

Tf-idf (Cavalcanti et al. 2014a; Manning et al. 2008; Shokripour et al. 2015) is a weighting technique which produces a composite weight for a term in each document. It is used for measuring the similarity between a query and a document (having several documents, the document with highest similarity score with the given query is considered the document most similar to the given query). Having a **query (bug) *"q"***, we use the following equation to calculate the score for **document (developer) *"d"*** in the **corpus *"D"*** (previous bugs):

$$\textbf{score(q, d)} \;=\; \sum_{t \in q} tf(t,\ d) \cdot idf(t, D) \tag{2.2}$$

In the above formula, *tf* (term frequency) is the number of occurrences of term t in

document d. On the other hand, *idf* (inverse document frequency) measures the importance of a term with regard to all documents in the corpus D (Manning et al. 2008).

We use Equation 2.2 to evaluate the similarity of a bug and a developer. It assigns a score for each developer, regarding a given bug. Then we sort the developers in the potential-assignees community based on this relevance score from high to low. We implemented the above metric and the experiment using Java. Then, we ran our code and evaluate our recommendations by comparing them against the ground-truth assignees in our data set. Based on the position of all the ground-truth assignees in our recommended ranked list, we calculate a MAP for each project and an overall MAP. We made the source code of our experiments available online[1].

2.6.1 The Data Set

In our previous research, we studied "several Github projects" (Sajedi-Badashian et al. 2016), chosen due to their popularity in Github. In this study, we used the new data for the same projects. There were 20 projects in our previous study, out of which we could extract the information of bug reports of 13 projects[14]. The other 7 projects stopped publishing their issues publicly (e.g., switched to private issue tracking systems).

We extracted data of those 13 projects from their beginning (2009-04-28 or later, based on project start dates) to 2016-10-31. This data set is inclusive of the old data set and includes several times more bug reports. Another limitation that we resolved is that in our previous data set (Sajedi-Badashian et al. 2016), we limited the developers to only the shared ones between Stack Overflow and Github (which was around 10% of the total developers). Here, we do not have such a dependency. So, we do not filter the data set. We extracted and stored information of all the project members in Github as developer community. To obtain the data, we wrote a set of JavaScript programs to extract the data of those projects online using Github APIs. The source code of this program is accessible online[15].

Table 2.5 shows statistics of our data set, including different assignment types and the number of bug reports based on each definition. The important aspect of this data set is

[14]The link to the 13 projects we studied are: http://github.com/lift/framework, http://github.com/html5rocks/www.html5rocks.com, http://github.com/yui/yui3, http://github.com/khan/khan-exercises, http://github.com/tryghost/ghost, http://github.com/fog/fog, http://github.com/julialang/julia, http://github.com/adobe/brackets, http://github.com/travis-ci/travis-ci, http://github.com/elastic/elasticsearch, http://github.com/saltstack/salt, http://github.com/angular/angular.js and http://github.com/rails/rails

[15]https://github.com/TaskAssignment/software-expertise

that we also reported the number of members in the developer community who have ever fixed any bugs during the captured lifetime of project. The minimum, average and median of this number are 38, 583 and 541 respectively, which shows the assignee prediction on this data set is not trivial. We made the data set available online[1] for further BA research.

Table 2.5: The data set including T1 to T5 bug-assignments in each project

Project	#of members in Developer Community (DC)	#of members in DC who have ever fixed any bugs	# of bugs	# of bug-assignments				
				T1: AUTHOR	T2: CO-AUTHOR	T3: ADMIN_CLOSER	T4: DRAFTED ASSIGNEE	T5: ALL TYPES
Framework	75	38	325	129	97	225	115	566
Html5rocks	159	47	627	90	1	638	269	998
Yui3	175	77	526	122	4	541	235	902
Khan-exercises	206	82	624	19	5	654	179	857
Ghost	473	357	3,578	1,371	113	3,713	945	6,142
Fog	770	208	1,124	91	27	1,146	63	1,327
Julia	831	541	9,086	1,759	54	9,590	1,345	12,748
Brackets	864	646	6,255	171	6	6,554	3,731	10,462
Travis-ci	1,159	1,096	5,473	13	2	5,716	603	6,334
Elasticsearch	1,262	758	10,423	2,192	498	10,362	3,132	16,184
Salt	2,283	1,227	10,237	2,344	233	10,682	2,274	15,533
Angular.js	2,386	1,069	7,402	503	196	7,671	1,288	9,658
Rails	4,079	1,431	8,794	857	138	9,366	944	11,305
Total	Average: 1,132 Median: 831	Average: 583 Median: 541	64,474	9,661	1,374	66,858	15,123	93,016

In order to capture different types of assignees, we used the definitions of T1 to T5 (Section 2.5.1) precisely. For the two assignee types related to the commits (i.e., T1 and T2), we needed to check the commit messages. In Github projects, the fixers reference the bugs from commit messages with one of the following specific keywords (or the capital cases of any of the keywords), followed by an optional space, followed by a number sign (#) and one or more bug number(s)[16] (Github 2017a):

- "fix", "fixes", "fixing", "fixed"
- "close", "closes", "closing", "closed"
- "resolve", "resolves", "resolving", "resolved"

We cross-referenced this with issue events[17] (which were also extracted by our code using Github APIs) to avoid capturing of communications between developers or typos as bug resolving. For T2, we captured the referencing developer as a second assignee of this type

[16]Bug numbers are iterative numbers, usually starting from 1. Pull requests also share this numbering system with bug reports in a manner that a number is considered only for a pull request or a bug report (called issue in Github).

[17]Issue events include every interaction about bugs including opening, closing, referencing, subscribing, assigning and reopening.

only if the *committer* was different than the *author*. Otherwise we just considered one assignee as T1.

For T3 and T4 assignment types, we did not need commit messages, but we examined the issue events precisely (especially the events *closed*, *assigned* and *unassigned*) to extract those two types of assignment correctly. We also paid enough attention to the possible complications in which there are more than one assignee –e.g., a bug report is closed and opened again several times, each time assigned to a developer who does some work.

Our data set is one of the most complete data sets currently available for BA regarding size, number of bug reports and community members, duration of projects and selection of projects (i.e., based on popularity in Github). In terms of number of bug reports and especially size of *developer community*, it is one of the most extensive data sets, comparing with the respective values in other research in the field (even comparing the 13 selected BA studies).

We use the above experiment setup and data set for inspecting the research questions in next section. Our source code, data set, input and output files (as well as documentation for running the code in simple steps) are available online[1].

2.7 Findings

In this section, we investigate the two dimensions of variability (mentioned in Section 2.5), and the effect they can have on the evaluation. Then we discuss the best choices regarding them.

2.7.1 Comparing Different Types of "Ground-truth Assignee"

In the previous sections, we discussed T1 through T4, the main types of "ground-truth assignees" used in the BA literature. We also proposed T5 as the union of the four. Here, we investigate which definition is better to be adopted as the ground truth in BA experiments. In other words, we address the following research question:

RQ1: What is the best definition of "ground-truth assignee"?

In order to study different types of ground-truth assignee and analyze their qualities, we implemented the baseline *tf-idf* method. We run this method using 5 different versions of the same data set related to T1 to T5 (see Table 2.5). Then we compare the results of this method on them and perform some statistical analyses.

Table 2.6 shows the results of the baseline method over 13 projects considering five different assignee types. The average overall MAP is shown in the last row. It starts from 34% for T1 and goes to 46% for T2 (around 35 percent difference, which is a big difference). The overall MAP regarding T5 is in a moderate level (comparing against the other four types). This makes more sense since T5 is inclusive of all the extreme cases of other four definitions together. So, when using T5, those extreme cases would not easily bias the results, but will be considered along with other cases in a more reasonable way.

Table 2.6: MAP (%) for 13 projects using different assignee types

Project \ Assignee Type	T1	T2	T3	T4	T5
Framework	56.41	83.02	51.24	44.19	49.13
Html5rocks	69.73	1.82	70.29	37.99	59.85
Yui3	48.69	13.75	40.24	47.40	53.13
Khan-exercises	28.06	1.70	39.32	48.84	41.80
Ghost	32.99	76.64	77.02	42.44	58.33
Fog	44.74	55.66	63.54	42.71	60.58
Julia	37.41	37.85	44.12	62.34	45.32
Brackets	38.36	42.08	32.71	37.17	34.70
Travis-ci	23.21	50.71	50.93	61.08	52.63
Elasticsearch	37.80	31.59	44.63	29.84	37.95
Salt	31.29	68.64	36.00	39.24	35.22
Angular.js	35.67	42.93	36.49	40.72	37.21
Rails	17.40	19.83	34.56	38.76	32.21
Total (13 proj)	34.27	46.25	42.51	40.28	40.52

We also studied how the results based on each assignee type are distributed over the projects. Figure 2.5 reports the distribution of project-based MAP values, using different assignee types over 13 projects; T2 exhibits the highest variance over different projects. Also, T1, T3 and T4 have outliers, which shows the unexpected difference for different projects. On the other hand, T5 have low variance and no outliers. Overall, considering these differences with the fact that T5 produces a moderate average MAP (comparing against the four other types), we can say that T5 gives the results in a more acceptable and robust range (without much tolerance over different projects). Note that in Table 2.5, some projects have only a small number of bug-assignments (e.g., less than 50) regarding T1 or T2. To verify that this high variance is not specific to those projects, we excluded those projects and re-generated the box plot (not shown here). Interestingly, again, T2 has the highest variance and T5 shows the most robust behavior over different projects.

A quick look over the results of each project regarding T1 to T4 in Table 2.6 shows their high disparity. As the reported results based on those four ground-truths would vary a lot

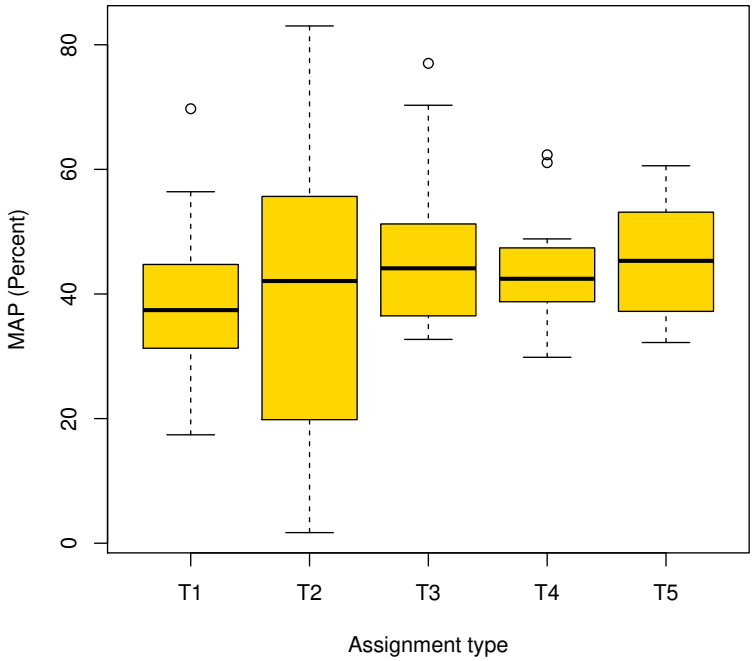

Figure 2.5: Distribution of MAP over 13 projects using five different assignment types. T5 has the lowest variance

for a project, we see that those narrow definitions can lead to biased or extreme results in different projects. Hence, picking any of them for evaluation of a research can make subjective results.

In addition, we believe that each definition, from T1 to T4, is targeting only a specific aspect of bug fix. So, considering the results based on any of them as ground truth measures how good the proposed approach is regarding that specific aspect. The nature of T1, for example, is targeting whoever commits the code as the regular programmer. T3, however, indicates a developer with some higher-level status in the project (e.g., a core developer), who reviews the code and closes the bug. In some cases, this developer brings the bug to the attention of appropriate programmers in the team. The "regular developer" nature of T1 is not comparable with the "authoritative developer" nature of T3. In fact, each of the assignment types are targeting different ground truths, without having major coverage with other ones.

Since narrowed types of assignee just capture a limited type of work towards fixing a bug, evaluation or making inferences, based on these narrowed definitions may lead to subjective judgments and would bias the evaluation process. Software development is a collaborative work; many bugs are fixed by a group of developers, who can have different roles in the project. So, all the developers who perform any type of development work towards fixing a

bug should count as proper assignees of that bug.

Considering from another point of view, the adopted definition for ground-truth assignee should not be impractically *broad* or *narrow*; Too broad definitions of ground-truth assignee can cause superficially high precisions because many developers would be assumed bug fixers of the bug (as mentioned in the above example). On the other hand, too narrow definition of *ground-truth assignee* can limit the total number and diversity of assignees. This fact can then be utilized by any insignificant approach to obtain high accuracies only by prioritizing the previous assignees at any point in time. Even we see later that it can restrict the *developer community* (if the developer community is not defined correctly) and bias in precision, recall, or accuracies.

When dealing with industrial projects, considering T5 –as the union of T1 to T4– reduces the bias in the single definitions and makes a richer data set including various types of work (towards bug-fix). T5 covers different indications of bug-fix on which we can fairly validate a BA approach. Since the assignees of all those types have done some useful work toward fixing a given bug, suggestion of any of those developers to the project manager would be some help towards fixing the bug.

Based on all the above analyses, we can conclude that:

> The most comprehensive definition of "ground-truth assignee" is T5 (ALL_TYPES). While it is essential to consider different types of work towards fixing a bug as indication of assignee, T5 enables us to capture this variety of assignees. This set of assignees is a proper ground truth for evaluating goodness of an assignee recommendation approach.

In the next sections, we perform the rest of our analyses based on this ground truth (T5).

2.7.2 The effect of "Developer Community"

The size of developer community is another important factor in evaluation of BA research. The bigger the developer community becomes, the more difficult the prediction of actual ground-truth assignee will be. In the previous sections, we discussed that "all the project members", "committers" and "set of all the developers who have assigned any bugs during lifetime of project" are the mostly used options in previous research regarding developer community.

Regardless of the definition of developer community, many of the previous research tried to filter the members of this community in their data set (i.e., remove developers who have

fixed less than a threshold number of bugs), which results in removing a number of bugs as well.

We would like to investigate which definition is more appropriate for evaluation of the results. Also, we want to understand if there is any side effect (e.g., bias in the results) in filtering this community or not. We address the following research question:

RQ2: What is the best definition for "developer community" from which the bug-assignment methods recommend appropriate developers? And what is the effect of filtering this community?

The assumptions of BA experiments should be realistic, to be useful in industrial applications. The definition of "developer community" determines a set of developers from which a BA approach recommends somebody to fix a bug. So, restricting this set to limited subsets of project members (e.g., the previous assignees, or the committers) eliminates the usage of the proposed BA method. Most open-source software companies cannot differentiate between their developers or restrict their bug-fixers to a subset of their developers. Note that in reality, even we do not know which developers would commit or fix a bug in advance, to limit our list to those developers [18].

Hence, we choose "*all the project members*" as the practical definition of *developer community*. It is comprehensive enough to contain any developer who works in the project and would step in to fix a new bug report. In open-source projects in Github, the list of project members is easily accessible[19].

Also note that when considering previous assignees (instead of all developers in the project) as the *developer community*, this community would be sensitive to the adopted definition of ground-truth assignee (e.g., T1 to T5; narrower definition of ground-truth assignee makes the community smaller). This might itself lead to more subjective judgments. Especially some evaluation metrics (e.g., top-10 accuracy and precision @10) are highly affected by a narrow definition of developer community. Suppose that a developer community is narrowly defined to contain 15 developers and each bug report has a single ground-truth assignee. Then, even a random selection of developers would superficially give high accuracy (i.e., 66.67% top-10).

Filtering the data set is another adverse restriction. In our survey, we saw many examples

[18]Another factor is "time"; this includes the time each developer starts and stops working in the project and can have realistic effects in industrial projects. Since considering "time" poses some real-time filtering in the developer community, we ignored this factor and just considered all the developers at all the time.

[19]As an example of list of project members in open source projects, Github has APIs for returning members of a project; https://developer.github.com/v3/repos/collaborators/

of BA research that suffer from filtering data sets (Ahsan et al. 2009; Aljarah et al. 2011; Anvik et al. 2006); this might bias the results and make the reproduction of the research difficult (note that different data sets react differently against filtering). We would like to investigate the effect of this filtering on the evaluation results in a high level. The studies that perform a data filtering, usually remove less active developers. Some mention that they removed bug reports assigned to developers who fixed less than a threshold number of bugs. These two cases have the same result; they eliminate both developers and bug reports and get rid of the challenging bug reports. In fact, those bug reports were **specific bug reports** that need more in-depth investigation to be connected (and assigned) to some **specific developers**, both of which are removed from the data set for simplicity.

Filtering less active developers can cause flimsily high precision, recall, or accuracies by artificially shrinking the *community of developers* considered. This was verified before by Lee *et al.* (S. Lee et al. 2017). They compared their method on two versions of the data set, for three open-source projects; a version including all the developers, and another one including only the developers who fixed at least 10 bug reports. Interestingly, they obtained much better results in the second case. Similarly, Tamrawi, *et al.* (Tamrawi et al. 2011a) reported increases in their accuracy, and, Anvik and Murphy (Anvik and Murphy 2011) reported increases in their recall by filtering the less active developers. In another study, Canfora and Cerulo (Canfora and Cerulo 2006) showed that filtering the developers to 100 (out of around 400 to 600 developers in Mozilla and KDE projects), multiplies the recall by around 3 and 4 respectively.

To have a deeper understanding of the effect of filtering developers on the accuracy of bug assignment, we ran an experiment using the baseline *tf-idf* approach as we discussed before (again, we made the source code of this experiment available online[1]). We considered T5 as the definition of assignee and used MAP for evaluation. We ran the program several times and each time, applied a different filtering on the developer community and bug reports fixed by them. Then we captured the overall MAP. The results are shown in Figure 2.6. Moving from top to bottom, the number of bug reports decreases (~5% in each row). The first row is the complete data set. In the next rows, the less-active developers and the bugs they fixed are filtered. The cut-offs are selected so that in each row, the number of bugs is decreased around 5% of the total bugs, with respect to the higher row. In each row, the remaining bug reports are those that were handled by the most active developers. In the last row, only 20% of the bugs are considered. It is remarkable that the MAP increases up

Least number of bugs fixed by each developer	Considered developers	Considered bug reports		Overall MAP
		Number	Percentage	
1 (no filter)	7577 (all)	93016 (all)	100	40.52
2	1956	87,395	95	42.68
6	568	83,794	90	44.30
40	236	79,125	85	46.46
88	186	74,506	80	48.52
145	156	69,832	75	50.55
202	113	65,208	70	53.32
321	67	60,274	65	55.62
415	55	55,978	60	59.00
532	44	50,981	55	62.02
613	36	46,431	50	65.50
656	29	42,021	45	68.22
872	22	36,979	40	74.13
1090	17	32,205	35	79.57
1269	13	27,673	30	86.26
1435	10	23,735	25	92.31
1919	7	19,185	20	95.66

Correlation: -0.98 ⇨ negative (almost) linear relationship

Figure 2.6: The effect of filtering less-active developers, and the bug reports they fixed, on accuracy of results

to 95%. In other words, a simple filtering on the developer community and their fixed bugs can increase the overall MAP by a factor of 150% (make it 2.5 times the original value) and make the overall MAP close to 100%. We also obtained the correlation between "percentage of remaining bugs that we run the algorithm on", and "overall MAP". They have a negative linear correlation (-0.98) which shows how the filtering can manipulate the results. Note that this is only an example with *tf-idf* as a baseline method. One could envision manipulating the utilized method to gain more benefit from the narrowed list (i.e., considering the time of activity of developers as well) and enhance results after filtering. Also note that this effect can be different (higher or lower) in another data set, based on the dynamics of that data set.

So, developer filtering can cause to obtain higher (biased) results than would be obtained in real cases. In industrial projects, the project managers need realistic solutions that consider all the developers; even the less active developers may fix some bugs and the project managers need to take them into account. According to several previous research (Sean et al. n.d.; Song Wang et al. 2014; M. Zhang and Hurley 2008), filtering less active developers reduces the diversity of developer recommendation, which results in diminishing practicality

of the proposed BA approach.

We conclude that the size of developer community should be reported clearly. In addition, the number of members of developer community who have ever fixed any bugs during lifetime of the project is also important. It is another useful indicator of the broadness of the project and should be reported. It gives us an indication of difficulty of predictions. Failing to do so makes it harder to compare against other research. In general, regarding the proposed research question, we conclude that:

> The best definition to adopt as developer community is "all the project members". It is the most comprehensive option to contain any developer who works in the project and may step in to fix a given bug. Filtering this set might produce biased results and mislead the evaluation.

2.8 Discussion

In our comprehensive survey on BA, we found that the field suffers from lack of a systematic framework for evaluation. Many of previous research are using small projects (or filtering the data of big projects and obtain limited circumstances) in which the assignee prediction is straightforward. Many of them implement another previously published method, to validate their approach through comparison. But again, the reporting may be skewed. Note that we are not criticizing the valuable amount of work toward research in the field. We identified lack of a systematic framework to facilitate (and standardize) the evaluation and reporting of scientific claims in BA domain, which needs more attention from software engineering community. In order to validate a new BA method and show that it works better than the previous approaches, we need to compare it against one or more previously published methods. Generally, there are three ways to do this: (1) to run the code of the previous approach and test it on our data, (2) to test the code of our approach on the data of the previous research, and (3) to compare the two approaches considering the reported evaluation metrics (e.g., meta analysis).

Initially, the first option above seems the best solution. Many research publications utilized their comparisons this way. However, in our survey, we found that most previous research did not publish their code. On the other hand, it is almost impossible to take into account all the details of a previous approach and re-implement it (note that usually the implementation details are not mentioned in the papers). Re-implementation of a

generic version of their approach is possible but less efficient. Moreover, many of them have no comprehensive source code and are, in fact, just a series of commands or transactions (e.g., interactions with Weka) which are almost impossible to replicate. Even if the previous researchers published their code or exact instructions of the needed transactions, our data might be inefficient for their approach. Finally, a research might have a data cleaning, that is a critical part and have important effects on its results.

The second option is a better solution, since we do not judge about the previous approach unless from their reported results. However, as we encountered in our survey, most previous research in the field did not publish their data, or if they did, the link to their data is broken. Even having their data, it might not be compatible with our approach (e.g., assume that our method needs component information which is not available in the data set of a previous research).

To the best of our knowledge, meta-analysis (the third above-mentioned option) is not used before in the field. However, it is a proper choice and can provide appropriate comparison against the previous research. It is not dependent on the data or code of previous research. Then, it is feasible to compare against many previous research (e.g., the ones mentioned in Table 2.3). Doing so, excelling a bunch of approaches is more persuasive than outperforming only one or two implementations. However, the project conditions and characteristics might affect the comparisons in this option and make the comparison unfair.

In either case, validation of a new approach needs *fair* comparison against state-of-the-art methods. In this way, observing several points as mentioned in the next sub-section are essential.

2.8.1 The proposed evaluation framework

We provide our evaluation framework based on the discussed research questions of this study, and regardless of how we compare a new approach against state-of-the-art methods (above three options). It contains guidelines for maintaining a reproducible BA research, with standards for judgment of scientific claims (Peng 2011) and replicating the study (Fomel and Claerbout 2009). They cover important aspects about evaluation, reporting and comparison against state-of-the-art methods:

1. It is recommended to evaluate and report based on MAP as the most stable and inclusive evaluation metric. MAP is a single measure, representative of both precision

and recall. It bundles the rank of all the assignees for each bug report, with emphasis on higher ranks. When there are several projects, an overall MAP for all the bug reports over all the projects is extremely useful. This helps to compare fairly when project-based comparison of two methods is difficult. In addition to MAP, we recommend to calculate and report the other widely used metrics (see Table 2.3), to enable meta-analysis and comparison against the previous approaches that reported other metrics.

2. It is recommended to use the most comprehensive definition of ground-truth assignee, T5, for evaluation. It includes every development work towards fixing the bugs. Using this as the ground truth, the results would be less sensitive to specific types of work (e.g., commit history) or other parameters in the project (e.g., number of bug reports in the project).

3. "All the project members" is the best choice for the developer community in evaluation of BA approaches. This generalization makes the proposed BA approach more comprehensive and useful for industrial projects, in which all the developers may step in to fix a bug. This list, like ground-truth assignees, can be extracted from open-source projects easily. The important point is not to use any filtering (on bugs or developers). Filtering would eliminate the developer community, reduce the challenging bug reports and make an isolated evaluation which might not be representative of real situations.

4. It is recommended to perform the evaluation on relatively large number of bug reports, number of bug-assignments (since a bug can be assigned several times), size of developer community and number of developer community members who have ever fixed any bugs. If this last number is too small (e.g., roughly around 20 or less) in a project, then that project is not a good case for evaluation, since it is not challenging enough to reflect the goodness of different methods. Also, to avoid judgments based on limited data, it is recommended to test on fairly large number of bug-assignments (e.g., roughly around 500 or more). After all, it is needed to report all these details per project.

To publish reproducible BA research, we recommend the above baselines to be established. This makes the reporting, review, judgment, comparison and replication of the study easier.

2.8.2 Threats to validity

Construct validity:

Most reasonings of this study were based on intuitions obtained from our survey (e.g., the metrics used, and the dynamics and settings used in various BA studies, as well as their data sets). In the second step of our survey, we removed studies that have major data-filtering. While this reduces bias in general, it might eliminate some good prior work with assumptions useful for industrial projects. We argue that we considered the general case to simplify the problem and end up with a list of publications that have the highest level of reproducibility. The more general case needs more investigation and can be considered as a future work.

Internal validity:

To capture the ground-truth assignees (i.e., the ground truth to validate our approach), we used projects that use Github's issue tracker. We looked for bugs in Github projects and their certain links to the commits and issue (bug) events, to find the ground-truth assignees. Although it is a common practice to mention and preserve those links in Github's open-source projects (Github 2017a), there might be some missing links (Bachmann et al. 2010) that we did not consider. This can potentially affect our validations. However, note that validation of these cases and inclusion of the full links between commits and bug reports is a tedious task, that must be done manually (Bachmann et al. 2010). To the best of our knowledge, no previous research has done this manual process. All the previous BA research in the field evaluated their approach against heuristic-based ground-truth extracted automatically from available data of software projects.

External validity:

A threat to validity is that we supported our arguments based on some experiments on our data set. These arguments might do differently in other data sets and settings. While we admit this as a threat, we argue that first, the experiments, data set and settings we considered is one of the most comprehensive cases in the whole literature. It includes 13 big projects, thousands of active developers and near 100k of bug-assignments which is non-trivial. Moreover, the experiments we established are not the only basis for our statements. This study is also supported by the findings from the comprehensive survey we proposed, the arguments we provided, and some evidence from the literature.

2.9 Conclusions

In this study, we accomplished three main contributions;

1. We comprehensively surveyed the previous research in BA, and reviewed the different BA objectives, methods, used data and dimensions of variability. This survey can be highly useful for further researchers planning to explore and research in the field.

2. We proposed a framework for evaluation of BA research; We investigated the mostly used evaluation metrics in BA studies. We argued that MAP is the best evaluation metric. It can be independently (from other metrics) interpreted and considers rank of all the ground-truth assignees. It emphasizes the higher ranks and is not highly affected by number of ground-truth assignees. Despite all these benefits, and its wide usage in evaluation of ranked retrieval results (Manning et al. 2008), it is rarely used in BA. We hope that this study highlights its benefits for evaluation of further research.

   Then, we demonstrated the impact of two important dimensions of variability and provided arguments for establishing a realistic evaluation; first, *definition of ground-truth assignee*, which is used as the ground truth in BA research, should comprehensively include every bug-fix effort by developers. Second, the *developer community*, who are the potential assignees used to validate a BA approach, should be inclusive of all project members.

   All in all, validating a new BA approach needs some spirit of equity and fairness. In our proposed framework, we mentioned the important aspects of evaluation and reporting BA research. Addressing those aspects enables replication of the study and promotes its usage in other research or industrial applications.

3. Finally, the data set we extracted from popular Github projects contains the full set of assignees based on the comprehensive definitions of ground-truth assignee and developer community. We made it available online[1] for further research. This data set is one of the most comprehensive and recent data sets available for further BA research.

Acknowledgements

The work is supported by Graduate Student Scholarship[20] funded by Alberta Innovates - Technology Futures (AITF)[21] and Queen Elizabeth II Graduate Scholarship [22] funded by Faculty of Graduate Studies and Research (FGSR)[23] at University of Alberta.

[20]https://fund.albertainnovates.ca/Fund/BasicResearch/GraduateStudentScholarships.aspx
[21]https://innotechalberta.ca
[22]https://www.ualberta.ca/graduate-studies/awards-and-funding/scholarships/queen-elizabeth-ii
[23]https://www.ualberta.ca/graduate-studies

Chapter 3

TTBA: Thesaurus and Time Based Bug Assignment

Preface

In this chapter, we describe our new method called *TTBA*, which only relies on bug descriptions as the most prevalent source of expertise of developers in previous approaches. *TTBA* benefits from two key intuitions. First, considering the time of evidence of expertise of developers, *TTBA* uses the recency of sources of expertise to emphasize on newer evidence and neglect the older ones. It benefits from a high granularity of time of usage of the keywords in previous bug-assignment instances. Second, using Stack Overflow as a thesaurus of technical programming keywords, *TTBA* highlights technical terms and also considers their specificity. We show that using these two key points, *TTBA* outperforms state-of-the-art methods.

This section has been submitted to the Journal of Systems and Software (JSS).

Abstract

Bug-assignment, the task of ranking developers in terms of the relevance of their expertise to fix a new bug report is time consuming, which is why substantial attention has been paid to developing methods for automating it.

In this chapter, we describe a new bug-assignment approach that relies on two key intuitions. Similar to traditional bug-assignment methods, our method constructs the expertise profile of project developers, based on the textual elements of the bugs they have fixed in the past; unlike traditional methods, however, our method considers only the programming keywords in these bug descriptions, relying on Stack Overflow as the thesaurus for these keywords. The second key intuition of our method is that recent expertise is more relevant than past expertise, which is why our method weighs the relevance of a developer's expertise based on how recently they have fixed a bug with keywords similar to the bug at hand.

We evaluated our BA method using a data set of 93k bug-report assignments from thirteen popular Github projects.[1]. In spite of its simplicity, our method predicts the assignee with high accuracy, outperforming state-of-the-art methods.

[1]The data set as well as source code, documentations and detailed output results are available at: https://github.com/TaskAssignment/TTBA-Outline

3.1 Introduction

As a key task of software quality-assurance process, bug-assignment (BA) aims at identifying the most appropriate developer(s) to fix a given bug. Typically, BA methods consider the developers' previous development activities as indicators of their expertise and rank the developers' relevance to the bug in question using a variety of heuristics. BA is an important problem for the software-engineering industry, and it involves a number of challenging questions, including what project data to consider as evidence for a developer's expertise, and how to utilize developers' expertise to recommend the best developers to fix new bugs.

Because of its importance, the problem has already received substantial attention over the past decade (Aljarah et al. 2011; Bhattacharya and Neamtiu 2010; Jeong et al. 2009; Linares-Vásquez et al. 2012; Liu et al. 2016; Nguyen et al. 2014; Shokripour et al. 2012). Nevertheless, BA is still a time-consuming task in software development (Akbarinasaji et al. 2017; Saha et al. 2015). Large projects receive hundreds of bug reports daily (Tian et al. 2016), which get recorded in their issue-tracking tools, which typically support flexible searching and, in some cases, duplicate detection. No issue-tracking tool, however, automates the BA task, which is still manual or, at best, semi-automated. Especially for big projects, a valid and accurate BA tool would be extremely desirable (T. Zhang et al. 2016; W. Zhang et al. 2016b) since it can automate the BA process and save project managers' time and effort which is currently spent on assigning bugs to developers.

The first key question in developing an automated method for assigning a bug to the developer best qualified to fix it is to decide on "how to match the information available on the bug with the information available on the developer's prior experience and contributions". The methodological assumption of our work is that *technical terms* and their *time of usage* are critical in this "matching". Even though much of the information about bugs and developers is textual, recognizing expertise relevant to a bug is very different from textual-similarity assessment, which has been the prevalent paradigm for this task to date. This is why some previous research used the traditional *TF-IDF* (Manning et al. 2008) to differentiate between specific and common keywords (Banerjee et al. 2016; Shokripour et al. 2015; T. Zhang et al. 2016; W. Zhang et al. 2016b). Shokripour *et al.* used a time-based variant of the original *TF-IDF* method as the main similarity metric between bug reports and developers (Shokripour et al. 2015). It used the document-length and frequency of the keywords (as the requirements of the *TF-IDF* method), plus the "last" time of usage of the keywords. However, it ignores other usages of the keywords and their times.

In this study, we propose a new similarity metric originally based on *TF-IDF*, but with two important enhancements. 1) The metric deals with the relevance of a developer's expertise to a given bug by considering the **technical-keyword space**. We ignore all words that do not belong to the technical vocabulary of Stack Overflow tags, which are curated by the software-engineering community. Our method weighs the importance of the keywords based on their distinctiveness in Stack Overflow. 2) Furthermore, the developers' expertise shifts as their tasks evolve over time. So, our similarity metric takes into account the recency of the technical-keyword appearance in the developer's record. It benefits **high granularity of the time** [2] of usage of the terms in previous BAs. In effect, our similarity metric is a **t**hesaurus and **t**ime-aware **b**ug-**a**ssignment, henceforth TTBA[3]. We show that our model notably enhances the assignee recommendation accuracy.

To evaluate this metric and to examine the relative importance of its two constituent intuitions, we have curated an extensive data set, including bug reports from thirteen open-source projects, their meta-data and textual information and their assignee(s). This data set contains 93k bug-assignments and we will publish it, as well as all the source code and instructions for our experiment, for further research and investigations. Using this data set, we demonstrate that TTBA outperforms current state-of-the-art methods.

The remainder of this chapter is organized as follows. Section 3.2 places our work in the context of the recent relevant literature. Section 3.3 describes the new metric we have developed for estimating the relevance of a developer's expertise for a given bug. Sections 3.4 and 3.5 describe our data set and details of the experiment we performed. Sections 3.6 and 3.7 report our results and discuss their implications. Finally, Section 3.8 concludes with a review of the lessons we learned from this work and outlines some avenues for future research.

3.2 Background and Related Research

The most prevalent formulation of the BA task is as follows: "Given a new bug report, identify a ranked list of developers, whose expertise (based on their record of contributions to the project) qualifies them to fix the bug" (Bhattacharya and Neamtiu 2010; Hu et al.

[2]The high granularity of the time is provided by the fine-grained usage of sub-documents (i.e., previously assigned bugs) and their assignment time.

[3]In this thesis, we do not use the full synonym-related potentials of the thesauruses, but it is possible to extend our work to use this feature for word inferences as well (e.g., for expanding the queries).

2014; Matter et al. 2009; Shokripour et al. 2015); this is the formulation we try to solve by our BA method –ignoring other formulations like "team BA" (Jonsson 2013; Jonsson et al. 2016) and "multi-objective BA" (Karim et al. 2016; Khalil et al. 2017; Liu et al. 2016) [4].

As we saw before in Section 2.3 in our survey, recent studies mostly focused on IR based activity profiling since it usually leads to higher accuracies (Shokripour et al. 2013). While some of those approaches mostly rely on textual information (e.g., title and description) of the –new and old– bug reports as clues for indication of expertise of developers and matching with new bug reports (Čubranić and Murphy 2004; Tamrawi et al. 2011b), others used variety of meta-data fields (e.g., component, product, severity and operating system) (Matter et al. 2009) to address some of the limitations. But still text-based methods are the most effective techniques used (Shokripour et al. 2015; Sun et al. 2014).

In recent years, some of the studies combined two or more different methods (e.g., ML, tossing graphs, Information Extraction, NLP and IR) (Bhattacharya and Neamtiu 2010; Bhattacharya et al. 2012; Cavalcanti et al. 2014b; Cavalcanti et al. 2016; Jeong et al. 2009; Shokripour et al. 2012; Sun et al. 2017; W. Zhang et al. 2016b). Some of these studies (W. Zhang et al. 2016b) showed a tendency towards *social* point of view, combined with the other methods. Some other examples (not shown in the above table) are building a social network of developers to model their relationship with each others or with bugs or even source code components (Hu et al. 2014; T. Zhang et al. 2016) or combining KNN and IR methods (Zanjani et al. 2015).

Conceptually closer to our work are BA methods that consider the timeline of developers' activities and the importance of keywords:

First, previous research considered the *time* of previous evidence of expertise of developers as indication of relevance to the new bug report. The old evidence are outdated and have less effect on appropriateness of the developer for the new tasks. Having a new bug report at hand, they considered the time of "last" usage of its keywords by a developer, to obtain the recency factor of that keyword for the mentioned developer. Then, they apply this factor in the developer's score (Shokripour et al. 2015; Tian et al. 2016); the former considered *last usage time* in a custom expertise formula and the later embedded it in the *TF-IDF* formula.

However, we believe that a fine-grained view over the usage of "all" the keywords by developers is needed; in fact, higher granularity over *time*, helps providing a more precise

[4] "Multi-objective BA" is when there are more than one factor (e.g., assignment accuracy, total cost and time-to-fix) as objectives of the assignment problem.

weighting for all the times a keyword have been used by a developer and gain more practical scoring scheme. Our similarity metric considers all the times of the usage of a term by a developer.

Second, the *importance* of the keywords is captured by some previous research. A number of previous BA studies used *TF-IDF* as a term-weighting technique (Manning et al. 2008) to emphasize on some keywords (T. Zhang et al. 2016; W. Zhang et al. 2016b). The motivating assumptions behind such term weighting are that some keywords are more important than the others because they are more specific and/or distinctive, or because they better capture specific interests of developers. There are other studies that tried to deal with different words differently. Aljarah *et al.* (Aljarah et al. 2011) uses Log-Odds-Ratio as a term-selection technique to identify the discriminating terms (i.e., those that are frequently used in the bugs assigned to a developer, but not frequently used by the others) and removed the rest of the terms. This way, it emphasizes on specific keywords that each developer uses or responds explicitly –by fixing the bugs containing those keywords.

The problem with these methods is that they just use the frequencies of the terms in the corpus, to determine importance of the keywords. It is true that less-frequently used keywords might be more important to focus (rather than widely used, general-purpose keywords), but there is no indication of the "technicality" of those keywords. We believe that capturing *less-frequently used* ‘*"technical"* terms can help obtain the similarity of a bug report to a developer more precisely. Our approach uses Stack Overflow for inferencing about the technicality of the terms and obtaining term-weights consequently. Stack Overflow, as the leading question answering platform, is the first choice of most developers in the world, for seeking their programming answers (Meldrum et al. 2017; Ponzanelli et al. 2015). With more than 15 millions of questions as of now, it covers every important programming topic (Li et al. 2015; Meldrum et al. 2017). In our previous research, we proposed a model to recommend developers in Github solely based on their posted questions and answers in Stack Overflow (Sajedi-Badashian et al. 2016). We used neither term-weights nor Github contributions in that study. Also, unlike our previous research, in the current study, we do not use information (i.e., textual elements) of questions and answers. In fact, the neat structure of the tags and their usage in more than 15 million questions inspired us to use them as a rich set of software-programming keywords. So, in this study, we use the set of Stack Overflow tags as a thesaurus to obtain general weights for the technical terms[5].

[5]Note that this is the only usage of Stack Overflow in the current study. There is no overlap with our

We add these two factors (*time of usage* and *importance* of the keywords) to the well-known *TF-IDF* metric, and obtain a fine-grained method, called TTBA.

3.3 Recognizing Developers with Relevant Expertise: The *TTBA* Metric

In the IR formulation of BA, the profiles of developers (including the text of previously fixed bugs and other contributions) correspond to the *documents* and the description of the new bug report is considered as the *query*. Using this notation, Shokripour *et al*. combined *TF-IDF* (as the main scoring function for developers regarding a new bug report) with *time* (Shokripour et al. 2015)[6]. Their approach added a recency factor based on the **last time of usage** of keywords by the developers. It only considers the time of last usage of the terms, but we believe considering "all" usage times of the keywords enhances the predictions (since the usage of the terms is scattered over time). In fact, more granularity is needed to capture the time of usage of a keyword by a developer precisely. Second, these methods infer the term weights based on raw counts of how many times they appear in the project. They do not consider which of these terms are part of the *technical*-keywords of software-programming vocabulary, which is likely to be more important than regular words in English. Third, weights inferred from small projects (or at the early stages of bigger projects) are bound to be inaccurate, since there are not enough documents available. Finally, due to the dynamic nature of term weights calculated over the history of the project lifecycle and/or the developers' contributions, these methods need to re-calculate the term weights by proceeding with new bug reports. Although this is mitigated by exploiting inverted indices, it still can be time-consuming since some bug reports include considerable number of generic keywords.

To mediate these shortcomings, we propose a *term-weighting* method that (a) has the ability to consider the timestamps of the (sub-)documents that contain these keywords; (b) emphasizes specific, *technical* (programming-related) keywords rather than general ones; (c) has a fixed term-weighting which is independent of the corpus; and (d) reduces the search space by filtering the text of bug reports and making them shorter.

previous work.

[6]Here, the usage of *TF-IDF* as the main scoring function regarding a developer (document) for a new bug (query) is regarded. There are some other studies that used *TF-IDF* for simple term-weighting (Cavalcanti et al. 2016; T. Zhang et al. 2016; W. Zhang et al. 2016b) within another main method and are excluded here.

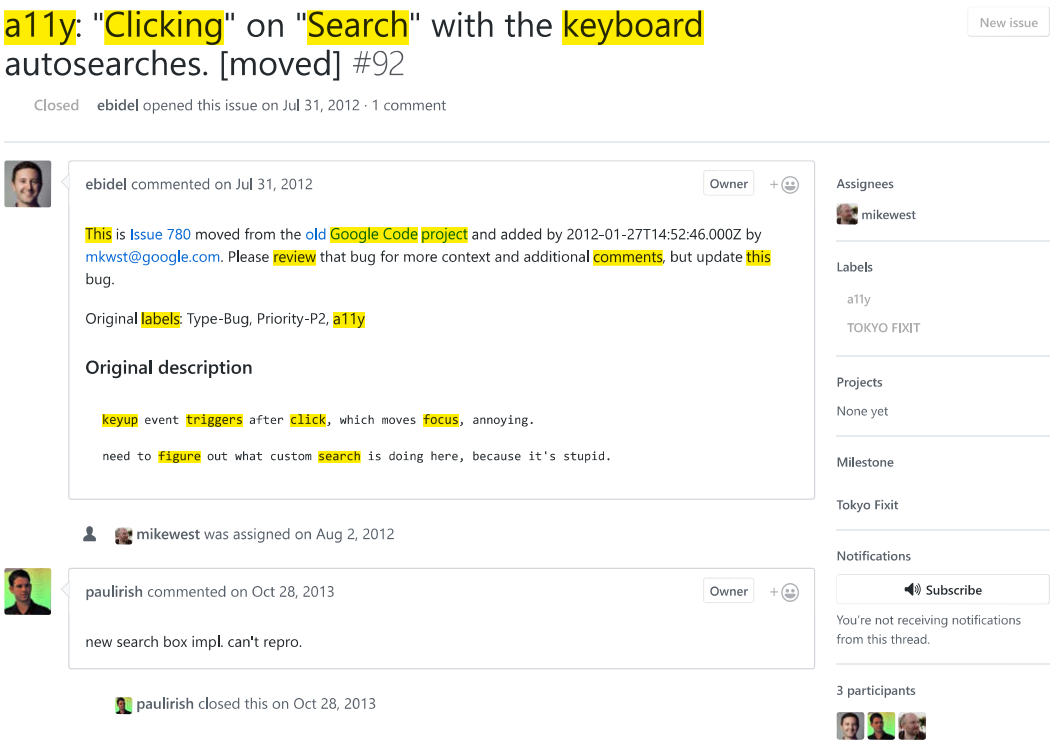

Figure 3.1: A sample bug report (#92) in *www.html5rocks.com*

To set the context for a better understanding of the dynamics of bug and their assignment, we use the example in Figure 3.1. The figure depicts a sample bug report[7] in one of projects[8] we studied in our experiments.

On the page of each bug, Github shows the title, description, meta-data elements (reporting date, reporter, labels, etc.), and the work around that bug. The bug report depicted in this example is reported on Jul 31th, 2012 and closed on Oct 28th, 2013. During this time period, the interactions around the bug –comments, being assigned or closed– are shown in the page of the bug report. If the bug is referenced from a commit or other actions, then an entry will be shown in the interactions section (bottom section of the bug). In this figure, the keywords in the bug's title and description that are also a Stack Overflow tag are highlighted.

A developer who fixed a bug report, is considered knowledgeable regarding the keywords mentioned in that bug report (e.g., title and description). This fact is usually taken into account by BA approaches. However, some of these keywords are general terms to describe

[7]https://github.com/html5rocks/www.html5rocks.com/issues/92

[8]The title of project is "www.html5rocks.com" and its Github page is https://github.com/www.html5rocks.com

the case and do not contain information value for BA. For example, the bug report #92 shown in Figure 3.1 is reported by *ebidel* and is assigned to two different developers –*mikewest* as the indicated assignee, and *paulirish* as the closer, which are both indications of bug-fix (Sajedi-Badashian and Stroulia 2018a). The terms *priority-p2*, *original* and *description*, for example, are used by the reporter to give some high-level instructions rather than technical information about the bug. The same author used the same keywords in several other bug reports. Most of these keywords do not contain technical information about the bug and can be rather misleading when calculating the *TF-IDF* score for developers. *Idf* only addresses the case if the terms appear as a boilerplate for all (or many) bug reports. But if this is repeated in a small number of bugs (e.g., this reporter used these keywords as his signature or partial standard, as it is here), *idf* just slightly reduces the weight of those terms. Stop-word removal cannot solve the problem. It just removes some of those misleading keywords, not all of them. The same problem exists for other non-technical keywords (e.g., words *issue*, *context* and *update*) that are rather generic terms used for communication purposes. Finally, there might be specific terms which do not appear a lot in the bug reports, but still do not have technical value for the triager. In all the above cases, some domain-specific reasoning is needed to remove non-technical terms.

As a specific solution to target the important keywords, we perform a simple filtering; we use Stack Overflow as a thesaurus of programming terms including more than 46,000 tags. The tags are defined to describe the questions. Each question should have between one to five tags. Tags cover all the important details of programming topics in any programming language. They are high level enough to be used as subjects when searching for questions, and low level enough to contain technical keywords. A *tag* can be as high level as the name of a programming language, framework, library, technology or method, or as low level as the name of an exception (in a specific programming language), error, web service, layout, plugin, widget, or any other programming topic. Stack Overflow tags are very reliable references because they are authentic keywords generated by the community of developers and curated by them during time.

We remove any term that is not an Stack Overflow tag. So, there is no need to do the stop-words removal, stemming and so on (since it is somehow included in the process). Considering Figure 3.1 again, the highlighted terms are the Stack Overflow tags. Removing the non Stack Overflow tags not only gets rid of those misleading terms, but also removes the stop-words, etc.

We consider each developer a document consisting of one or more sub-documents. Each sub-document includes the textual elements of one of the bug reports assigned to that developer in the past. Later, we will use the time of each sub-document as a decay factor for evidence of expertise of that developer regarding the keywords in that sub-document. Each sub-document consists of a bug report's title and description, plus the main languages of the project (after removing non-Stack Overflow tags). We assumed main language of a project as any programming language that contains at least 15% of the lines of code of that project.

We propose our similarity metric that is obtained from *TF-IDF*, but is time-aware and relies on fixed term weights, to give a score to each developer.

3.3.1 *TF-IDF*

The traditional *TF-IDF* was described in the previous Section in Equation 2.2. The provided definition was assuming that each term appears only once in q. To generalize this to include repeats of the terms, the measure becomes as follows:

$$score(q,\ d)\ =\ \sum_{t \in q} freq(t,\ q) \cdot tf(t,\ d) \cdot idf(t, D) \tag{3.1}$$

In the above formula, t is a distinct term in q, and, $freq(t,\ q)$ is the frequency of the term t in query q. The promises of *TF-IDF* is to differentiate between general and specific terms by calculating two main components *tf* and *idf*; *tf* (term frequency)[9] measures number of times a term, t, appears in document d, normalized by document length:

$$tf(t,\ d)\ =\ \frac{\#\ of\ times\ t\ is\ mentioned\ in\ d}{total\ \#\ of\ terms\ in\ d} \tag{3.2}$$

On the other hand, *idf* (inverse document frequency) measures the importance of a term with regard to all documents in the corpus D:

$$idf(t, D) = \log_{10}\left(\frac{total\ \#\ of\ documents\ in\ the\ corpus}{\#\ of\ documents\ containing\ t}\right) \tag{3.3}$$

[9]Note that the definition provided here includes normalization and is one of the alternatives of *tf* which we found more effective. The main definition of *tf* according to (Manning et al. 2008) is equal to "number of occurrences of term t in document d".

Regarding occurrence of a term "more than once" in the query or a document, there are alternatives for the relative factors –i.e., $freq(t,\ q)$ and $tf(t,\ d)$. For this purpose, different weighting schemes are used to normalize these two factors (Manning et al. 2008; Manning, Schütze, et al. 1999). For example, occurrence of a term twice may or may not double the value related to that term. We will embed a few of these normalization factors later in our enhanced version for tuning.

With the original *TF-IDF* metric, the documents in a corpus can be ranked based on their similarity with a given query. However, since a document (developer) contains several sub-documents (e.g., the text of previously fixed bug reports) related to different times, we adapt the formula in two steps to be applicable for our problem as mentioned in the following sub-sections.

3.3.2 Focusing on a Thesaurus of Terms

We make two changes to the *TF-IDF* (one for each part; *idf* and *tf*), to make it more suitable for our problem.

In addition to emphasizing on specific keywords, we would like to highlight "technical" terms (i.e., important *software programming* keywords). So, we use a bigger thesaurus (than the increasing set of documents as the *corpus*) to decide about technical value, specificity or generality of a term. Instead of inferring *idf* from the corpus D; we replace $idf(t, D)$ in the Equation 3.1 by $w(t)$, and calculate the score of document (developer) d for query (bug) q, containing several distinct terms ($t \in q$) as follows:

$$score(q,\ d)\ =\ \sum_{t \in q}\ freq(t,\ q) \cdot tf(t,\ d) \cdot w(t) \tag{3.4}$$

In the above formula, $w(t)$ is a generic term weight, independent of any particular project (one time definition) obtained based on frequency of the tag t in all Stack Overflow questions. For the term t, we define $w(t)$ in the similar way $idf(t, D)$ was defined in Equation 3.3:

$$w(t) = log_{10}(\frac{total\ \#\ of\ SO\ questions}{\#\ of\ questions\ tagged\ with\ t}) \tag{3.5}$$

The idea is that the more a tag is repeated in different questions, the less specific the keyword is (because it appears in many situations and becomes a common term). In other words, tags that appear in a small number of questions are specific keywords which can infer

a specific problem and are more useful in keyword matching between query q and document d (i.e., the new bug report and a developer).

Stack Overflow covers the used terminology in open-source projects. All the tags are technical, programming keywords curated by expert programmers[10]. So the above formula identifies the technical terms, and, among them, weights the general keywords (that appear frequently in different documents and are not much informative) lower than the specific ones. The keywords that are not a Stack Overflow tag are filtered and removed from the bug reports. In other words, the set of tags of Stack Overflow is considered as the thesaurus of technical terms and the rest of the words are not considered. In addition, our weighting is constant and there is no need to re-calculate it after processing each bug report, unlike the *idf* values (note that the *tf* value still needs to be calculated for each new bug report)[11]. In addition to other technical benefits, the above points lead to speeding up the computations in the assignment process as we will see later.

The second amendment to the original *TF-IDF* is regarding the *tf* formula. Since each document (i.e., developer) consists of several sub-documents (i.e., text of their previously fixed bugs), we calculate the tf value in each of its sub-documents separately, and then aggregate them over all the sub-documents. This, allows considerations (e.g., normalization) based on sub-document length to preserve the effect of all the sub-documents equally. So, the tf will be as follows:

$$tf(t,\ d)\ =\ \sum_{j=1}^{n_d} tf(t,\ sd_j) \tag{3.6}$$

In the above equation, sd represents a sub-document and n_d is the number of sub-documents of d (i.e., the different assigned bugs to a developer in our problem, or generally, any other work evidence of that developer). This will replace the $tf(t, d)$ factor of the scoring function in Equation 3.4. This granularity in sub-document (previously assigned bug) level rather than document (developer) level also allows considering time of each sub-document in the next step.

[10]Only top Stack Overflow developers with a minimum reputation of 1,500 can create new tags or manage them

[11]Note that for higher performance, the term weights, $w(t)$, can be re-calculated once every while (e.g., every few months), to consider the new Stack Overflow tags that might emerge.

3.3.3 Recency-aware Term Weighting

We consider a developer's expertise as a sequence of evidence (constructed from a variety of tasks that the developer has contributed to the project), called sub-documents (e.g., the textual elements of previous bugs assigned to a developer). The sub-documents belong to different times (from years ago to a previous moment). Over time, developers' interest, their expertise and even their assigned module change. In fact, there is decay in expertise and change in interest areas of developers (Matter et al. 2009; Sajedi-Badashian et al. 2015; Servant and Jones 2012; Shokripour et al. 2015). Evidence of expertise for developer *d1* about subject matter x that occurred last week is more important than other evidence for developer *d2* about x that occurred last year.

In order to capture the time of usage of the keywords in calculating the *tf* in Equation 3.6, which affects the total score of Equation 3.4, we inject a *recency* factor. The idea is to weigh the recent sub-documents of document (developer) d higher than the old ones. Intuitively, expertise of people decays over time exponentially (Carlson 2015). However, according to the literature (ElSalamouny et al. 2009), the decay principle can be implemented in many ways, including time-based and event or activity based. We try two *recency* options for sub-documents based on the previous research. First, we just consider the timestamp of the sub-document, according to (Servant and Jones 2012). We calculate the recency of a sub-document as the portion of time up to the time of occurrence of the sub-document to the total time –which is between zero and one. For $\mathrm{sd_j}$, the $\mathrm{j^{th}}$ sub-document of d, we define the "absolute time-based" recency, $abs\_recency(sd_j)$ as shown below:

$$abs\_recency(sd_j) \;=\; \frac{date(sd_j) - \; beginning\ date\ of\ project}{today \; - \; beginning\ date\ of\ project} \tag{3.7}$$

Alternatively, the recency of a document can be estimated based on the amount of work (e.g., bug fixes) that has been done after a specific evidence of expertise (or interest), as a decay factor of that expertise (Sajedi-Badashian et al. 2015). We calculate the recency as the inverse of the number of sub-documents occurred between this sub-document and the bug. The value of this recency factor is again between zero and one. The "relative activity-based" recency factor for $\mathrm{sd_j}$, the $\mathrm{j^{th}}$ sub-document of d, is calculated as shown below:

$$rel\_recency(sd_j) = \frac{1}{1 + number\ of\ other\ sub\text{-}documents\ occurred\ after\ sd_j} \tag{3.8}$$

In the above two definitions, since the older documents (i.e., evidence) are out-dated, their weight decreases. On the contrary, the weight for the new documents increases. We will test both these factors in the tuning of our algorithm and compare their efficiency. With the injection of *recency* factor in Equation 3.6, the $tf(t, d)$ for term t in document d will be as follows:

$$tf(t,\ d)\ =\ \sum_{j=1}^{n_d} tf(t,\ sd_j) \cdot recency(sd_j) \tag{3.9}$$

Finally, substituting this new definition of $tf(t, d)$ in the score function of Equation 3.4, we obtain $score(q,\ d)$ as our final similarity metric between query (bug) q and document (developer) d:

$$score(q,\ d)\ =\ \sum_{t \in q}\ freq(t,\ q) \cdot w(t) \cdot \left(\sum_{j=1}^{n_d} tf(t,\ sd_j) \cdot recency(sd_j) \right) \tag{3.10}$$

In the above equation, t is any distinct term in q, n_d is the number of sub-documents of d (e.g., the number of previous assignments of any bugs to the developer d) and $w(t)$ is the weight of the term t. We use the above scoring function for assessing the suitability of a developer for a given (new) bug report. We calculate the above score for each member in the developer community. Then sort and rank them based on their obtained score. We call our approach *TTBA* (*Thesaurus and Time based Bug-Assignment*). We evaluate *TTBA* using real BA data from several open source projects regarding different evaluation measures as described in the next sections.

3.4 The Data Set

We use the same data set of 13 projects as described in Section 2.6.1. We choose T5 as the definition of ground-truth assignee. This notion includes anyone who worked towards fixing the bug in any of the four manners; 1) the author of a commit referencing a bug as resolved, 2) the committer (co-author) of a commit referencing a bug as resolved, 3) the developer who closes the bug and 4) the developer who is tagged as "assignee" when the bug is closed. So practically, a bug report can have several real-assignees at different points in time (e.g., a bug report is closed and opened again several times, each time assigned to a developer who does some work), all of which are proper fixers to be considered as ground truth (Anvik et al. 2006; Sajedi-Badashian and Stroulia 2018a).

Regarding community of developers that could be considered as assignees, we extracted and stored information of all the project members in Github[12]. In each project, we run the algorithm for all its members, rank them for each bug report and recommend the top ones.

Regarding the Stack Overflow data, we used the Stack Exchange Data Dump (I. Stack Exchange 2017) provided by Internet Archive. We only used the tags and posts information, in order to obtain the weights as mentioned before in Equation 3.5.

Our source code, data set, input and output files (as well as documentation for running the code in simple steps) are available online[1]. It is one of the most comprehensive data sets currently available for bug-assignment regarding selection of projects (i.e., based on popularity in Github), duration of projects, number of bug reports and community members.

3.5 Evaluation

To investigate the effectiveness of our approach, we evaluate the recommendations of our method by comparing them against the ground-truth assignees in the above data set. For each bug in the projects, we use the metric defined in Equation 3.10 to assign a relevance score to each developer member of the potential-assignees community, and then sort them based on this score from high to low to recommend a ranked list of developers. Once a bug is processed, we update the list of ground-truth assignees associated with this bug in all future assignments, to include all past assignees. Based on the position of the ground-truth assignee(s) in our recommended ranked list, we evaluate and report the effectiveness of our approach.

Note that there is no need to divide the data set to training and test sets. In fact, our approach is on-line; at each point in time, for predicting the assignee of a given bug, we use all the evidence from beginning up to previous bug. The first few assignments would be less accurate, but while it proceeds with any new assignment, it updates the documents (technical terms in bug reports fixed by each developer) as the expertise profiles of developers.

We perform two types of evaluation; first, we compare *TTBA* against other methods we implemented; *TF-IDF* and *Time-TF-IDF* (Section 3.6.1). Then, we compare our results against the reported results of 13 other studies in the literature (Section 3.6.2). Before doing so, in the rest of this section, we introduce the metrics we used for evaluation, and evaluate the impact of parameters in the formulation of *TTBA*.

[12]Github has APIs for returning members of a project; https://developer.github.com/v3/repos/collaborators/

3.5.1 Evaluation Metrics

As we mentioned in previous chapter (in our survey on BA), the best evaluation measure for BA is Mean Average Precision (MAP). After MAP, there are 10 other widely used metrics in the literature that are recommended for reporting the results. These measures are Mean Average Precision (MAP), top-k accuracy (k = 1, 5 and 10), Precision @k (k = 1, 5 and 10), Recall @k (k = 1, 5 and 10) and Mean Reciprocal Rank (MRR).

In order to enable the reproducibility of our work, we report our results based on all these metrics (11 metrics in total) for our experiments including a comprehensive set of projects and bug reports. In addition to implementing two other methods and comparing our results against them on the same data, we report a meta-analysis by comparing our results against previous studies, using their reported results on any of these 11 metrics.

3.5.2 Tuning and Optimization

There are a number of factors about how to run the algorithm that can affect the outcome of our approach. We considered a number of these factors in an optimization experiment on three test projects[13] to obtain the best configuration for our evaluation metric. We mention the main ones called *primary factors* here (the full list of considered factors as well as the detailed process of tuning is available in Appendix 3.A). To optimize our approach regarding these factors, we selected a few possible values for each of those factors –based on literature and also our previous experience in optimizing assignment problems (Manning et al. 2008; Sajedi-Badashian et al. 2016; Servant and Jones 2012):

- **Weighting**: The term-weighting provided by $\boldsymbol{w(t)}$ provides an option to emphasize on specific keywords in our main scoring function. This factor has two options; "do not use term-weighting" and "use term-weighting". The former includes the constant value of "one" for all the terms. In the later case, we consider the term weighting obtained from Stack Overflow as a thesaurus to emphasize on specific keywords.

- **Recency**: The recency factor provided by $\boldsymbol{recency(sd_j)}$ enables emphasis on recent usage of keywords in our main scoring function. It has three options; "no recency",

[13]We divided the projects into three categories; projects with 1k-, 1k to 10k and 10k+ bug-assignments. There are 4, 4 and 5 projects respectively in small, medium and large categories. Then, we selected the smallest of each category for tuning. The idea was to optimize on relatively low number of bugs and leave more bug reports for the main experiment. The selected projects are: lift/framework, fog/fog and adobe/brackets. The percentage of bug reports used for optimization is ∼13%.

"*abs_recency*" and "*rel_recency*". The first option disregards the time of usage of the keywords and applies a constant value of "1" for the $recency(sd_j)$ factor in our scoring function. The next two options are as described before in Equations 3.7 and 3.8.

We used three projects (one from each category; small, medium and large) for optimizing the above parameters, using extensive 2d exploration (Blanco and Lioma 2012; Craswell et al. 2005). We performed a mutation experiment –also called grid search (Tatsis and Parsopoulos 2016) or exhaustive parameter search (Zhai and Lafferty 2002)– to obtain the best configuration for the above parameters. We considered all the possibilities of the parameters and obtained the results for the three test projects on all those combinations. The optimization took two hours in total[14]. The best obtained case is as follows; "use term-weighting" and "*rel_recency*" (see Table 3.1 for their optimization results). The best configuration for primary factors is shown in bold and grayed cells in the table. In addition of those factors, there are other less-important factors, which we call *secondary factors*. These factors only have minor effect on MAP and for brevity we omitted them from this table (see Appendix 3.A for details of this mutation experiment as well as the minor effects of secondary factors).

Table 3.1: Obtaining best configurations for the main factors affecting the accuracy of *TTBA*

Weighting	Recency	MAP
do not use term-weighting	no recency	0.3685
	abs_recency	0.3865
	rel_recency	0.5405
use term-weighting	no recency	0.3728
	abs_recency	0.3933
	rel_recency	**0.5447**

The *recency* makes the most difference in the results. Especially *rel_recency* which is based on relative amount of work is more effective than *abs_recency*, which is solely based on time. Using either of *rel_recency* or *abs_recency* is better than having no recency. The other effective parameter is *term-weighting*. Having term-weighting increases the MAP in all combinations of the other parameter.

We just used the three test projects in the above steps and did not tune our method on other projects since it is a time-consuming task. Also, it would be unfair to optimize on the whole data set. However, doing so, even better overall results might be obtained. In the

[14]This time includes optimization for primary factors (as shown here) and secondary factors (as shown in the Appendix). All the experiments and optimizations are done on a machine with Core i7 CPU and 16GB of RAM.

next section, we use the obtained values from the optimization step as the final experiment configuration. In order to avoid bias, we exclude the three test projects, and run the main experiment on the remaining 10 projects.

3.6 Results

In this section, we compare the performance of our method (a) against two baseline methods on the same data set, and (b) against the performance of other methods as reported in the literature, as a quick meta-analysis.

3.6.1 Comparisons Against Implemented Baseline Methods

We implemented the baseline *TF-IDF* method as well as *Time-TF-IDF* method (Shokripour et al. 2015) and compare our results against them. The final project-based results as well as overall results are shown in Table 3.2. To avoid bias, we excluded the three test projects in this step. The overall MAP over the other 10 projects is even (around 3%) better than the MAP over three test (tuning) projects. Our MAP ranges between 0.51 and 0.71 in different projects with an overall value of 0.57. Full details of all 93k bug-assignments as well as Java source code of the implementation of the three methods are available in our repository[1].

Considering overall results (over 10 projects) regarding all the 11 metrics, our approach, *TTBA*, outperforms the two other implemented methods.

TTBA thoroughly outperforms the baseline *TF-IDF*; regarding all the metrics (i.e., *MAP*, *top-k accuracy*, *precision @k*, *recall @k* and MRR), in all single projects as well as overall values, *TTBA* obtained better results. The per-project MAP values for the baseline *TF-IDF* method are between 0.32 and 0.60 (which are between 6% to 25% lower than the MAP values of *TTBA* in the same projects) and the overall MAP is 0.41 (16% lower than *TTBA*).

Also, *TTBA* outperforms *Time-TF-IDF* (Shokripour et al. 2015), regarding most metrics, and most projects. The project-specific MAP values in *Time-TF-IDF* range from 0.42 to 0.66 (which in 9 projects are between 3% to 12% lower than MAP values of *TTBA* and in one small project 5% higher) and the overall MAP is 0.49 (8% lower than *TTBA*).

Generally, *TTBA* works much better in bigger projects. Excluding Yui3 and a small number of project-specific results, *TTBA* works better regarding all the other metrics. We will discuss more about these results in the Discussions (Section 3.7).

Table 3.2: Comparing *TTBA* results against *TF-IDF* and *Time-TF-IDF* (Shokripour et al. 2015)

Project	Method	Top1 (%)	Top5 (%)	Top10 (%)	p@1 / r@1	p@5 / r@5	p@10 / r@10	MRR	MAP
Khan-exercises	TF-IDF	25.79	68.26	85.06	0.26 / 0.22	0.16 / 0.66	**0.10** / 0.84	0.43	0.42
	Time-TF-IDF	39.56	81.68	87.98	0.40 / 0.35	0.19 / 0.80	**0.10** / 0.88	0.57	0.55
	TTBA	**57.18**	**85.41**	**89.26**	**0.57** / **0.52**	**0.20** / **0.84**	**0.10** / **0.89**	**0.69**	**0.67**
Yui3	TF-IDF	42.35	73.17	83.15	0.42 / 0.37	0.16 / 0.70	**0.10** / 0.81	0.56	0.53
	Time-TF-IDF	**56.76**	**78.71**	**86.25**	**0.57** / **0.50**	**0.18** / **0.76**	**0.10** / **0.85**	**0.66**	**0.64**
	TTBA	50.11	75.83	84.81	0.50 / 0.44	0.17 / 0.73	**0.10** / 0.83	0.62	0.59
Html5 rocks	TF-IDF	46.29	83.07	91.88	0.46 / 0.39	0.20 / 0.79	**0.12** / 0.91	0.63	0.60
	Time-TF-IDF	55.21	86.77	**93.09**	0.55 / 0.47	**0.21** / **0.84**	**0.12** / **0.93**	0.69	0.66
	TTBA	**66.63**	**87.17**	92.69	**0.67** / **0.57**	0.20 / 0.82	0.11 / 0.91	**0.76**	**0.71**
Ghost	TF-IDF	55.42	75.27	86.84	0.55 / 0.43	0.18 / 0.68	0.11 / 0.83	0.65	0.58
	Time-TF-IDF	55.45	83.00	90.90	0.55 / 0.43	**0.22** / 0.79	**0.12** / 0.90	0.68	0.64
	TTBA	**59.39**	**85.27**	**91.66**	**0.59** / **0.46**	**0.22** / **0.82**	**0.12** / **0.91**	**0.70**	**0.67**
Travis-ci	TF-IDF	41.35	71.20	77.23	0.41 / 0.40	0.15 / 0.70	**0.08** / 0.76	0.54	0.53
	Time-TF-IDF	51.83	**75.54**	**79.54**	0.52 / 0.50	**0.16** / **0.74**	**0.08** / **0.79**	0.62	0.61
	TTBA	**57.47**	74.74	0.78	**0.57** / **0.55**	**0.16** / 0.73	**0.08** / 0.77	**0.65**	**0.64**
Angular .js	TF-IDF	22.05	60.32	81.10	0.22 / 0.20	0.13 / 0.57	0.09 / 0.79	0.39	0.37
	Time-TF-IDF	34.50	73.84	85.72	0.35 / 0.31	0.16 / 0.71	**0.10** / 0.84	0.51	0.49
	TTBA	**46.47**	**79.87**	**86.19**	**0.46** / **0.42**	**0.17** / **0.76**	**0.10** / **0.85**	**0.60**	**0.58**
Rails	TF-IDF	21.80	47.00	62.29	0.22 / 0.20	0.10 / 0.43	0.07 / 0.58	0.35	0.32
	Time-TF-IDF	29.73	62.62	75.04	0.30 / 0.26	0.13 / 0.58	0.08 / 0.71	0.44	0.42
	TTBA	**41.42**	**69.91**	**78.69**	**0.41** / **0.36**	**0.15** / **0.66**	**0.09** / **0.76**	**0.54**	**0.51**
Julia	TF-IDF	33.79	62.18	76.36	0.34 / 0.31	0.13 / 0.59	0.08 / 0.73	0.47	0.45
	Time-TF-IDF	34.21	68.61	80.29	0.34 / 0.31	0.15 / 0.65	**0.09** / 0.78	0.49	0.48
	TTBA	**45.67**	**71.80**	**81.02**	**0.46** / **0.41**	**0.16** / **0.69**	**0.09** / **0.80**	**0.57**	**0.55**
Salt	TF-IDF	20.88	58.91	76.06	0.21 / 0.17	0.13 / 0.54	0.09 / 0.72	0.38	0.35
	Time-TF-IDF	29.26	66.06	79.64	0.29 / 0.24	0.15 / 0.61	**0.10** / 0.77	0.46	0.43
	TTBA	**43.28**	**72.63**	**81.14**	**0.43** / **0.36**	**0.17** / **0.69**	**0.10** / **0.79**	**0.56**	**0.53**
Elastic search	TF-IDF	25.28	55.70	72.97	0.25 / 0.23	0.12 / 0.53	0.08 / 0.70	0.40	0.38
	Time-TF-IDF	32.04	67.71	82.19	0.32 / 0.29	0.14 / 0.65	**0.09** / 0.80	0.48	0.46
	TTBA	**47.61**	**73.97**	**84.71**	**0.48** / **0.44**	**0.16** / **0.71**	**0.09** / **0.82**	**0.59**	**0.57**
Total (13 proj)	TF-IDF	28.92	60.05	75.44	0.29 / 0.25	0.13 / 0.56	0.08 / 0.72	0.43	0.41
	Time-TF-IDF	35.80	69.84	81.52	0.36 / 0.31	0.15 / 0.66	0.09 / 0.79	0.51	0.49
	TTBA	**47.50**	**74.73**	**82.93**	**0.48** / **0.42**	**0.17** / **0.72**	**0.10** / **0.81**	**0.59**	**0.57**
Max (13 proj)	TF-IDF	55.42	83.07	91.88	0.55 / 0.43	0.20 / 0.79	**0.12** / 0.91	0.65	0.60
	Time-TF-IDF	56.76	86.77	**93.09**	0.57 / 0.50	**0.22** / **0.84**	**0.12** / **0.93**	0.69	0.66
	TTBA	**66.63**	**87.17**	92.69	**0.67** / **0.57**	**0.22** / **0.84**	**0.12** / 0.91	**0.76**	**0.71**

3.6.2 Comparison Against Results Reported in the Literature

Different bug-assignment studies reported the performance of their methods on different metrics. Moreover, the assumptions and settings of various studies differ a lot (e.g., definition of ground-truth assignee and developer community and filtering inactive developers). These differences make it is hard to compare a new method against previous approaches. In our previous research (Sajedi-Badashian and Stroulia 2018a), we reported 13 exemplary BA studies (the 13 state-of-the-art approaches, mentioned in Section 2.3) as good candidates for

further comparison and meta-analysis. They have fairly good number of bugs and developers in their projects and did not have major data filtering. Here, we compare our results against their published results.

Since we obtained our results based on various metrics in addition to MAP (i.e., *top-k accuracy*, *precision @k*, *recall @k* and MRR), we are able to compare the performance of our method against all those studies (regarding one or more metrics for each study). Each of these studies reported their results on their chosen projects, without providing average results over all bug reports in all their projects. Thus, it is hard to make a pairwise comparison between any two studies. So, we compare our final, overall values (which is obtained over all bug reports of all our projects) against results of each project in those baseline approaches. Table 3.3 shows this comparison.

On all the metrics, the highest values are related to one of projects in *TTBA*. Regarding the overall (total) values, *TTBA* surpasses the results of all the projects of 8 studies; (Čubranić and Murphy 2004), (Canfora and Cerulo 2006), (Tamrawi et al. 2011b), (Shokripour et al. 2012), (W. Zhang et al. 2016b), (Cavalcanti et al. 2014b; Cavalcanti et al. 2016) and (Sun et al. 2017). There are a few cases in other 5 studies that they work partially better than *TTBA*:

(Jeong et al. 2009) obtained around 2% higher top-5 accuracy than our average in Eclipse, but 4% lower in Mozilla. (Matter et al. 2009) has almost the same precision @5 and @10 comparing to *TTBA*, but at least 10% below our average results regarding precision @1, and recall @1, @5 and @10). (Tamrawi et al. 2011a) evaluated their results over 7 projects. Their top-1 accuracy in two projects and top-5 accuracy in four projects are slightly better than our average. *TTBA* outperforms their reported results in the rest of the projects. The maximum and average of their values over all projects are lower than our maximum and average. (Bhattacharya and Neamtiu 2010; Bhattacharya et al. 2012) has slightly (2%) better top-5 accuracy than our average, but our top-1 is around 15% higher than their results in both projects.

3.7 Discussion

We tuned our approach on three projects in our data set and then tested it on the remaining 10 projects. The final results on the 10 projects were even better than the tuning results.

The improvement in the final results (over the results of tuning phase) shows that the

optimization is general and robust enough. On average, our method (*TTBA*) has overall MAP of 0.57. The high obtained MAP in our experiment shows that in most cases the ground-truth assignees are successfully retrieved and ranked at the top few ranks of the recommended list. This is also proved by the high values on the other reported metrics. For example, in 47.5% of the cases, *TTBA* recommends one of the ground-truth assignees in the first rank, in 74.73% of the cases in the first 5, and, in 82.93% of the cases in the top 10 ranks.

Table 3.3: The evaluation measures and other design choices of the selected research

Method / Project	#devs	#bugs	**Top1 (%)**	**Top5 (%)**	**Top10 (%)**	**P@1 / r@1**	**p@5 / r@5**	**p@10 / r@10**	**MRR**	**MAP**
TTBA(our approach)										
Total (13 proj)	75-4,079	566-16,184	47.50	74.73	**82.93**	**0.48/0.42**	**0.17/0.72**	**0.10/0.81**	**0.59**	**0.57**
Max (13 proj)			**66.63**	**87.17**	**92.69**	**0.67/0.57**	**0.22/0.84**	**0.12 / 0.91**	**0.76**	**0.71**
(Čubranić and Murphy 2004)										
Eclipse	162	15,859	30.00							
(Canfora and Cerulo 2006)										
Mozilla	637	12,447				- / 0.12	- / 0.21	- / 0.24		
KDE	373	14,396				- / 0.05	- / 0.10	- / 0.12		
(Jeong et al. 2009)										
Eclipse	1,200	46,426		77.14						
Mozilla	2,400	84,559		70.82						
(Matter et al. 2009)										
Eclipse	210	130,769				0.33 / 0.27	0.16 / 0.59	**0.10** / 0.71		
(Tamrawi et al. 2011a; Tamrawi et al. 2011b)										
Firefox	3,014	188,139	32.1	73.9						
Eclipse	2,144	177,637	42.6	80.1						
Apache	1,695	43,162	39.8	75.0						
Net Beans	380	23,522	31.8	60.4						
FreeDesktop	374	17,084	**51.2**	**81.1**						
GCC	293	19,430	48.6	79.2						
Jazz	156	34,220	31.3	75.3						
(Bhattacharya and Neamtiu 2010; Bhattacharya et al. 2012)										
Mozilla	?	549,962	27.48	77.87						
Eclipse	?	306,297	32.99	77.43						
(Shokripour et al. 2012)										
Eclipse	?	?				- / 0.32	- / 0.71			
Mozilla	?	?				- / 0.27	- / 0.48			
Gnome	?	?				- / 0.10	- / 0.45			
(W. Zhang et al. 2016b)										
Mozilla	∼874	74,100							0.28	0.44
Eclipse	∼544	42,560							0.28	0.56
Ant	∼203	763							0.35	0.36
TomCat6	∼79	489							0.35	0.37
(Cavalcanti et al. 2014b; Cavalcanti et al. 2016)										
New SIAFI - A	70	781	31.40							
New SIAFI - B	70	1031	22.00							
(Sun et al. 2017)										
JEdit	123	?	28.0	60.1	79.8					
Hadoop	82	?	8.5	30.1	50.3					
JDT-Debug	47	?	14.4	46.6	66.4					
Elastic	661	?	13.6	43.6	75.2					
Libgdx	345	?	22.0	51.3	69.6					

3.7.1 Intuitions About Comparison of the Implemented Approaches

Regarding overall results in all the 11 metrics, our approach, outperforms the two other implemented methods (*TF-IDF* and *Time-TF-IDF*). Considering project-specific results, our approach outperforms *TF-IDF* and *Time-TF-IDF* in almost all projects. In 7 projects (*Khan-exercises*, *Ghost*, *Julia*, *Elasticsearch*, *Salt* *Angular* and *Rails*), our approach comprehensively works better than the two other methods. In two projects (*Html5rocks* as a small project, and *Travis-ci* as a medium project), regarding some of the metrics, *Time-TF-IDF* works better, but still in MAP and other evaluation measures our approach works better. Note that between available BA evaluation metrics, only MAP is comprehensive enough to include all the assignments of each bug report (Sajedi-Badashian and Stroulia 2018a). For example, in *Html5rocks* *Time-TF-IDF* is ∼1% higher than *TTBA* regarding precision @5. However, this metric only considers first 5 recommendations and ignores the rest, but regarding MAP, *TTBA*'s results are ∼5% higher. So again, in these two projects (*Html5rocks* and *Travis-ci*), our approach outperforms the other two methods. The only exception is *Yui3* in which *Time-TF-IDF* has 5% higher MAP than *TTBA*. The reason can be regarding its size and duration; *Yui3* is the second smallest project, with the smallest duration, i.e., ∼38 months (while the average and median over all projects are 61 and 67 months respectively). So *TTBA*'s high granularity of expertise of developers over time did not work in this project as good as the other projects.

This fine-grained usage of data is especially evident in bigger projects; In four large projects with 10k+ bug-assignments (*Rails*, *Salt*, *Elasticsearch* and *Julia*), our approach obtained ∼10% and ∼20%[15] better results than *Time-TF-IDF* and *TF-IDF* respectively. The reason can be that in big projects, there are more bug-assignment data available, which are scattered over various dates. It enables our fine-grained scoring function to utilize them as various evidence of expertise distributed over time. On the other hand, in small projects, lack of enough bug reports might bring uncertainty in making decision. *Yui3*, as the only project in which *Time-TF-IDF* works better than our approach, is a small project. *Html5rocks* and *Travis-ci*, in which in some measures (excluding MAP) *Time-TF-IDF* competes against our approach are respectively small and medium projects.

We captured the details of the system we ran our experiments on. On a computer with Core i7 CPU and 16GB of RAM, it took between 1 to 26 seconds to run for each project (average = 12s), depending the number of bugs to be assigned. We found that it is quite fast;

[15]Note that in all the comparisons of this study, the absolute difference is mentioned, not relative difference.

the average time for assigning developers to each bug, considering the data of all projects, is 1.7 milliseconds. In total, it took 138 seconds for doing all the 93k bug-assignments. The two other approaches, *TF-IDF* and *Time-TF-IDF*, took 413 and 465 seconds respectively. There are two reasons for this speed up in TTBA. First, *TTBA* does not re-calculate the *idf* values after processing each bug report (which includes updating indexes and statistics related to frequency of the terms in the corpus); instead, it replaces it by $w(t)$ which is calculated once over the Stack Overflow data as pre-processing (although this does not need to be calculated every time, we measured its time; the pre-processing on Stack Overflow data took 63 seconds, which is again quite fast). Note that our approach still needs to calculate the *tf* values for all sub-documents (i.e., previously assigned bug reports to a developer). Second, *TTBA* filters the non-technical terms (i.e., any term except the Stack Overflow tags), and in fact, shortens the bug reports. As a result of these two enhancements, *TTBA* works more than three times faster than the other two compared methods.

3.7.2 Intuitions About Comparison of Reported Results

From a higher perspective, we compared the *TTBA* results against the reported meta-data of 13 best studies in the field. We showed that *TTBA* completely outperforms 8 studies. Five studies obtained better results regarding some metrics in a few projects, but we achieved much better results regarding several other metrics / projects. Note that over all the metrics, the highest values are obtained in one of our projects.

Regarding the dimensions of variability of the data sets (number of projects, bug reports and developers), our data set includes more challenging situations. It includes abundant of bug reports, more than most previous research in the area. With respect to number of developers per project (which is a key factor in the difficulty of assignee recommendation in a project), our data set includes the most populated projects (in average, 1,132 developers per project). So, the overall situations of our data set is more difficult than most mentioned studies in Table 3.3. Also the data we use is the simplest form of data used for BA (only bug reports' title and descriptions). As a result, competing against the meta-data reported by them as mentioned in the above table is fair and reasonable.

Regarding limitations, unlike some other studies mentioned in Table 3.3 (Matter et al. 2009; Shokripour et al. 2012; Sun et al. 2017), there is no dependency on specific information (e.g., interaction histories between developers, heavy historical changes on the source code files, the component and severity of the bug, etc.), other than title and description of the

bug reports, which are easily accessible in any open-source repository.

Unlike most previous research that we reported their results in Table 3.3(Bhattacharya and Neamtiu 2010; Bhattacharya et al. 2012; Cavalcanti et al. 2014b; Cavalcanti et al. 2016; Čubranić and Murphy 2004; Jeong et al. 2009; Matter et al. 2009; Tamrawi et al. 2011a; Tamrawi et al. 2011b), our approach does not need big training data sets. Most those approaches utilize ML algorithms and require big training steps, which make their approach incapable of recommending assignees –with high accuracy– for the first portion of their data set. We tested our approach from the first bug, to the last one, with no training data. In fact, the training data of our approach includes every bug report is being processed for assignment in our main experiment. For predicting assignee for bug report #n, the information of n-1 previous bug reports were considered. Starting from the first bug report, every bug it processes, it builds the sub-documents (i.e., the text of the bug reports fixed by each developer up to that point in time) as the expertise profiles of developers and uses them in the next ones. Considering the first part of the data set as training set (like those studies), would even increase our accuracy.

Considering the fact that the above research were the most reproducible research selected for comparison and meta-analysis (Sajedi-Badashian and Stroulia 2018a), and aggregating with the obtained results regarding superiority of our approach against two other implemented methods (*TF-IDF* and *Time-TF-IDF*), we argue that *TTBA* outperforms the state-of-the-art methods and is capable of precise assignee recommendation.

3.7.3 Threats to Validity

Like any other research, there are some possible challenges and threats to validity;

Construct validity:
A validity threat is about the definition of ground-truth assignee –e.g., "the developer who is tagged as *assignee* at the closing time, the developer who closes the bug, or any other developer who works towards fixing the bug" (Sajedi-Badashian and Stroulia 2018c). One can argue that in reality, this might not be the best person to fix the bug and there are other developers in the project who may be better choices. To address this issue, we argue that the best way to identify "all" the (possible) ground-truth assignees of a bug would be to discuss it with a project manager and tag all the appropriate developers as right assignees. Unfortunately, this plan needs a lot of effort and is too expensive. So, we defined the "right

assignee" based on the combination of the definitions used in the literature (Sajedi-Badashian and Stroulia 2018c). This is the most comprehensive definition of assignee that has been used up to now.

Internal validity:
In order to capture the ground-truth assignees (i.e., the ground truth to validate our approach), we used projects that use Github's issue tracker. We looked for bugs in Github projects and their certain links to the commits and issue (bug) events, to find the ground-truth assignees. Although it is a common practice to mention and preserve those links in Github's open-source projects (Github 2017a), there might be some missing links (Bachmann et al. 2010) that are not considered here. This can lightly affect the validation of our approach. Note that validation of these cases and inclusion of the full links between commits and bug reports is a tedious task, that needs to be done manually (Bachmann et al. 2010). To the best of our knowledge, no previous research has done this manual process. All the previous research in the field evaluated their approach by comparing their results against heuristic-based ground-truth extracted automatically from available data of software projects.

We found that there are some bug reports that have no, or only a few, Stack Overflow tags. If there are many such bugs in a project, the effectiveness of our approach will suffer. To address this issue, we programmatically counted the tags in each bug report. The bug reports of each project have 20 tags in average (between 11 to 32 tags in average in each project, with median 19). This number of tags is more than enough to convey the idea of a bug report. Only around 1% of the bugs have fewer than three tags and 5% have fewer than five tags. 75% of the bug reports have ten or more tags. For extreme cases where not enough tags are included in the text of the bug reports, we outline some potential means to expand the keywords in Section 3.8 as a future work.

There may be very short tags that are usually used along with other tags (e.g., "r" that usually is used with "rstudio", "plot", "sentiment-analysis", etc). In Stack Overflow, they are also explained by the question description. Their use may be problematic (e.g., the developer may mean the name of a simple variable, not the meaning indicated by the tag). In addition, some tags are generic words that have specific meanings (like the term "this", which is also a *Java* keyword) and may be misrepresenting. To address this issue, first, we checked the length of all the tags; there are in total 46,278 tags. Among those, 11 and 152 tags are one- and two-letter tags that usually come with other tags and it is hard to use

them as indication of meaning outside of Stack Overflow. To avoid false positives, these tags can simply be ignored since their number is low. The rest of them, 46,115 tags in total, are considered to filter the bug reports' textual elements (1,730 of them have three letters and the rest have four or more letters). Note that, as mentioned in the previous issue, usually (in 99% of the cases) there are enough tags in the text of the bug report. Moreover, in the case of misleading tags, again, we refer to the above statistics; since there are enough tags in each bug report, the combination of remaining terms usually is enough to converge toward the general idea of the bug report and usually cover the possible false positives. So, some of them may be eliminated through a stop-word removal step, as we mention in the future works.

Some tags have synonyms in Stack Overflow which are identified by the expert developers. We did not merge those synonyms. For example, "crypto" is a synonym for "cryptography". Merging the two includes changing all occurrences of "crypto" in the textual elements of all the bug reports to "cryptography". Also, it needs to merge the statistics of usages of the two terms into one term in Stack Overflow. In addition, synonym suggestion techniques (Beyer and Pinzger 2015) can be used for even enhancing the accuracy. These updates might change the weights and may lead to better accuracy but needs extra effort. While we agree that this can be considered as future work, we anticipate minor effect on the outcome since the number of synonym pairs in Stack Overflow is quite low, compared to the total number of tags (Beyer and Pinzger 2015).

Some of the tags are combinations of several keywords –e.g., "pull-request", "active-record-query", "for-loop", "http-status-code-403", "elasticsearch-java-api" and "jdk1.8.0", but they are not consistently used by everyone. The developers may combine these keywords the same way Stack Overflow does, or in a different manner (e.g., with capitalization, space, underscore and so on). Consider the tag "jdk1.8.0" as an example; it may be found as "jdk 1.8.0", "jdk_1.8.0", or "jdk 1 8 0". To address this issue, we did a cleaning step before running our approach; considering the textual elements of the bug reports, we applied a set of heuristics that concatenate consequent keywords with/without valid connections (i.e., "." and "-") to build possible composite tags. To avoid false positives, we have put limitations for the length of the combined terms. At the end, we have made joint tags from two, three and four consecutive keywords in the bug reports (respectively 11,999, 1,008 and 51 Stack Overflow tags with 236,905, 5,746 and 228 total occurrences in the bug reports).

One aspect that can diminish the performance of our approach is the changes during

time. For example, the set of developers change and new Stack Overflow tags appears. We answer that this can only have a negative effect on the accuracy of our approach in long term. For example, it takes long time (e.g., several months) to appear a set of new programming concepts and keywords, being adopted by Stack Overflow users as tags, and being used by the developers in the description of bug reports. Moreover, to address this issue in long term, we can re-calculate the term weights once every few months, based on new set of tags in Stack Overflow. Also, if new developers join the project, they will not be assigned any bugs for a short time. However, they will be assigned to new bugs after showing their level of expertise in a few bug reports (e.g., fixing some bugs). Also if a developer is removed from the project, our expertise metric gradually lowers their position in the list since they do not have any recent activity. Besides all these arguments, these changes (to developers or tags) can be considered and applied in our metrics by the project managers to obtain even higher accuracy than we showed in this study.

External validity:
We showed that use of Stack Overflow tags as a rich thesaurus to obtain term weights is efficient and straightforward. There is no doubt about this usefulness in open-source topics and projects. However, one may argue that it might not be as useful in some specific contexts. There may be some terminology in certain pieces of technology (e.g., related to proprietary software) that are not covered much by Stack Overflow tags. We argue that in these cases, the term weighting can be obtained using the alternative networks or Q&A systems (e.g., the developer network of the proprietary software). In fact, the main idea remains the same; utilization of a thesaurus from a developer network to extract a set of mostly referred keywords (referred to as the thesaurus) can help to obtain the required domain-specific term weights.

One may mention that TTBA does not assign any bugs to the newcomers since they have no previously assigned bugs, hence they have no sub-documents. So, they are given a score of zero. We argue that this is true for any bug-assignment method, because at each point in time, they only assign bugs to the developers who have some evidence of expertise in the system, up to that point in time. When predicting the assignee for a bug that is assigned to a newcomer, *d1*, like any other bug-assignment method, *TTBA* fails to predict the correct assignee since there is no previous evidence for *d1*. It gives the score of zero to *d1*. However, after this point in time, there will be one evidence for *d1*, and *TTBA* calculates

a positive score for *d1*. With processing more bugs having *d1* as the ground-truth assignee, more sub-documents are stored in the data set as his evidence of expertise, and his chance of being recommended increases.

3.8 Conclusions and Future Work

In this study, we described *TTBA*, a new BA approach based on developers' previous contributions in project. We used Stack Overflow as a thesaurus of technical terms to identify the importance and specificity of keywords and utilized it along with recency of developers' work in a scoring function for BA. We demonstrated that our method outperforms most state-of-the-art methods. *TTBA* does not require training; at each point in time it uses only the previous bug fixes. It might not produce very good results for the first few bug reports of each project but improves quickly by obtaining some BA data and using them as evidence of expertise.

Our BA metric considers previous bug fixing as evidence of expertise of developers, but our preliminary investigations and results (not included in this study) show that it is expandable to other sources of expertise in the repository like commits. We plan to consider it as a future work.

Our data set is the most comprehensive data set used in this field, to our knowledge. Our data set as well as our code in Github are available online[1] for further researchers to compare their method against our approach on our data set. As an extensive data set of bugs and developers, it can also be used for any other BA experiment.

In the future, we plan to expand the tags in a bug report to infer new keywords with query expansion or graph-based methodologies. This way, we may eliminate the limited number of tags in a few percentage of the bug reports and enhance the method's accuracy. Finally, some additional stop-words removal (e.g., removing *generic* Stack Overflow tags like *this*, *for* or *while*) may increase the quality of the tags and enhance the accuracy of our approach.

Currently, for each tag, we considered weights based on appearance in Stack Overflow. Then used these weights constantly for all the projects. This can change to obtain a better accuracy. One would envision using more specific weighted keywords for each domain or project. This can be done by altering the weights we obtained from Stack Overflow using the information from domain or project. Alternatively, the weights can be obtained using a

completely different schema (e.g., based on occurrence of those terms in that specific domain or project). The promises of this study, however, was to utilize the Stack Overflow as the thesaurus, in addition to a fine-grained utilization of time in the usage of keywords, and we showed that it works for the purpose of BA.

Acknowledgements

The work is supported by Graduate Student Scholarship[16] funded by Alberta Innovates - Technology Futures (AITF)[17] and Queen Elizabeth II Graduate Scholarship [18] funded by Faculty of Graduate Studies and Research (FGSR)[19] at University of Alberta.

3.A Appendix: Details of mutation experiment for tuning 6 parameters of our method

There are a number of factors that can affect the quality of our assignment. We introduce them here, and consider their effect in a small experiment on a few test projects:

- **Weighting**: The term-weighting provided by $\boldsymbol{w(t)}$ provides an option to emphasize on specific keywords in our main scoring function. This factor has two options; "do not use term-weighting" and "use term-weighting". The former includes the constant value of "one" for all the terms. In the later case, we consider the term weighting obtained from Stack Overflow as a thesaurus to emphasize on specific keywords.

- **recency**: The recency factor provided by $\boldsymbol{recency(sd_j)}$ enables emphasis on recent usage of keywords in our main scoring function. It has three options; "no recency", "*abs_recency*" and "*rel_recency*". The first option disregards the time of usage of the keywords and applies a constant value of "1" for the $recency(sd_j)$ factor in our scoring function. The next two options are as described before in Equations 3.7 and 3.8.

- **tf(t, sd$_j$)**: having the text of the sub-document sd_j, term-frequency can be measured in a number of ways (Manning et al. 2008). We chose to test four simple definitions:

[16]https://fund.albertainnovates.ca/Fund/BasicResearch/GraduateStudentScholarships.aspx
[17]https://innotechalberta.ca
[18]https://www.ualberta.ca/graduate-studies/awards-and-funding/scholarships/queen-elizabeth-ii
[19]https://www.ualberta.ca/graduate-studies

- **1**: "one" is the constant number considered when a term appears at least once in the text. In this case the *tf* does not depend on the number of repeats of the term in the text.
- **freq**: this is the frequency of the term in the text.
- **freq/#ofTerms**: this option normalizes the frequency of the term by dividing it to the #ofTerms which is the length of the text (shown as *textLength*). Now, the *textLength* can be itself considered in two cases as mentioned below.
- **log**: another normalization option that is increased logarithmically with increases in the frequency of the term and is equal to $1 + log_{10}(freq)$.

- **textLength**: in case we need to count *#ofTerms* in a piece of text (e.g., the *freq/#ofTerms* option above), we have two options:
  - **before** (b): in this case, the length of the original text (i.e., the text before removing any other keyword that is not Stack Overflow tag) is measured as #ofTerms.
  - **after** (a): in this case, the length of the remaining text (that just includes Stack Overflow tags) is measured.

- **freq(t, q)**: we consider the four options similar to the *tf* mentioned above, except that this is measuring the counts of the words for the query *q*. Like *tf*, in its third option (*freq/#ofTerms*), the two cases for *textLength* will take effect as mentioned above.

- **prioritizing previous assignees**: As an enhancement in our overall assignment process, we can consider a priority for the developers who have fixed at least one bug before, over the rest of developers (i.e., at each point in time, first, rank the developers who have fixed at least one bug up to that point, then rank the other members in the developer community randomly (since they all have the score of *zero*). Note that this way it gives a chance to the developers who did not have a common keyword with the new bug and so obtained score of *zero*, but were active regarding other keywords, over the other developers who were completely inactive). The options are "yes" and "no". Note that the data set is not filtered in any case and at each point in time, only the previous evidence is used.

We used three projects for tuning. In order to select the projects for optimization, we divided the projects into three categories; projects with 1k-, 1k to 10k and 10k+ bug-assignments for small (4 projects), medium (4 projects) and large (5 projects) respectively.

Then, we selected one project per category; we selected the smallest of each category for tuning: lift/framework, fog/fog and adobe/brackets. The idea was to leave the biggest projects with more bug assignments for the main experiment.

We used extensive 2d exploration (Blanco and Lioma 2012; Craswell et al. 2005) and performed a mutation experiment –also called grid search (Tatsis and Parsopoulos 2016) or exhaustive parameter search (Zhai and Lafferty 2002)– for optimizing the parameters and obtaining the best configuration. We considered all the possibilities of the different parameters to create different configurations and obtained the results for the three test projects in each case. In total, there are 276 different cases for our 6 factors to tune the algorithm. So, we ran the program in a loop generating the above 276 cases. We obtained the results of each run, and sorted them (descending) based on MAP, calculated over all the bug reports of all the three projects. We looked the top 10% in the results, and observed that we can decide for two of the parameters which we call *primary factors*. The *primary factors* are those that frequently appear with fixed value in those top 10% records and it is obvious that they have high effect on the output; *weighting* = "use term-weighting" and *recency* = "$rel\_recency$". The rest of parameters ("TF", "prioritizing previous assignees", "$freq(t,\ q)$" and "textLength") are called *secondary factors* for which we cannot conclude anything at this step. For brevity, since the produced output contains too many results, we do not report the detailed results of this step. The 276 configurations are as follows:

(A) Primary factors; *Weighting* and *recency*:
Weighting has 2 cases ("do not use term-weighting" and "use term-weighting") and *recency* has 3 cases ("no recency", "$abs\_recency$" and "$rel\_recency$").
This makes:

2 (two cases of *weighting*) × 3 (three cases of *recency*) = **6 cases**.

(B) Secondary factors; $tf(t, sd_j)$, *prioritizing previous assignees*, *textLength* and *freq(t, q)*:
$tf(t, sd_j)$ has four cases, and, in each case, '*prioritizing previous assignees* has two cases ("yes" and "no").
In three cases of $tf(t, sd_j)$ ("1", "freq" and "log"), for each case of *prioritizing previous assignees*, there are 5 cases for *freq(t, q)*:

"1",
"freq",

"freq/#ofTerms" + *textLength*="before",
"freq/#ofTerms" + *textLength*="after" and
"log".

This makes:

3 (three out of four cases of $tf(t, sd_j)$) $\times$ 2 (two cases of *prioritizing previous assignees*) $\times$ 5 (five cases for *freq(t, q)*) = **30** cases.

In the other case of $tf(t, sd_j)$ (i.e., "freq/#ofTerms"), regarding each *prioritizing previous assignees*, there are 8 cases for *freq(t, q)* (four original cases for "freq", times two for the two cases of *textLength*; "before" and "after").
This also makes:

1 (the last case of $tf(t, sd_j)$) $\times$ 2 (2 cases of *prioritizing previous assignees*) $\times$ 8 (four cases for *freq*, times two for the two cases of *textLength*; "before" and "after") = **16** cases.

So for the secondary factors there are 30 + 16 = **46 cases**.

(C) Combining the above two sets (multiplying primary and secondary factors), in total, there are:

6 (primary cases) $\times$ 46 (secondary cases) = **276 cases** .

These cases took 2 hours in total to run on a machine with Core i7 CPU and 16GB of RAM. After that, we set the primary factors to those values obtained from the previous step (although we still do not know the exact effect of each primary factor and will examine it at the end of this step). Then, run the code with 46 unique cases including all possible combinations of the secondary factors. We obtained the best combination as shown in Table 3.4. The best configuration is shown in bold (and grayed cells) in the table.

It seems that the obtained MAP values are too close to each other. To check if there is a significant difference between the result of different configurations of secondary factors, we calculated the Coefficient of Variation (CV) of the MAP values mentioned in Table 3.4. The CV is too low (0.0005). This indicates that these (secondary) factors have only minor effect on the final MAP and they can be neglected. However, as a systematic way of 2d exploration (Blanco and Lioma 2012; Craswell et al. 2005), we choose the best result because it may

Table 3.4: The results of optimizing secondary factors affecting the accuracy of *TTBA*

$tf(t, sd_j)$	Prioritizing previous assignees	$freq(t,\ q)$		MAP
1	yes	1		0.5443
		freq		0.5447
		freq/#ofTerms	tL = b	0.5447
			tL = a	0.5447
		log		0.5442
	no	1		0.5441
		freq		0.5446
		freq/#ofTerms	tL = b	0.5446
			tL = a	0.5446
		log		0.5440
freq	yes	1		0.5443
		freq		0.5447
		freq/#ofTerms	tL = b	0.5447
			tL = a	0.5447
		log		0.5442
	no	1		0.5441
		freq		0.5446
		freq/#ofTerms	tL = b	0.5446
			tL = a	0.5446
		log		0.5440
freq/#ofTerms	**yes**	1	tL = b	0.5443
			tL = a	0.5443
		freq	**tL = b**	**0.5448**
			tL = a	0.5447
		freq/#ofTerms	tL = b	0.5447
			tL = a	0.5447
		log	tL = b	0.5442
			tL = a	0.5442
	no	1	tL = b	0.5441
			tL = a	0.5441
		freq	tL = b	0.5446
			tL = a	0.5446
		freq/#ofTerms	tL = b	0.5446
			tL = a	0.5446
		log	tL = b	0.5441
			tL = a	0.5441
log	yes	1		0.5443
		freq		0.5447
		freq/#ofTerms	tL = b	0.5447
			tL = a	0.5447
		log		0.5442
	no	1		0.5441
		freq		0.5446
		freq/#ofTerms	tL = b	0.5446
			tL = a	0.5446
		log		0.5441

make a higher difference when we run the code for the main experiment (10 remaining projects) with several times more bug reports. The best results were obtained by setting the secondary factors as: "$tf(t, sd_j)$=freq/#ofTerms", "prioritizing previous assignees=yes", "$freq(t,\ q)$=freq" and "textLength=before(b)". By setting the secondary factors to those best obtained values, the effect of different combinations of primary factors as shown in Table 3.1 can be compared (12 cases as discussed in part "A" above).

Chapter 4

An Investigation Into the Information Value of Different Sources of Evidence on the Developers' Expertise for Bug Assignment

Preface

In this chapter, we investigate the effectiveness of various sources of evidence of the developers' expertise in the context of *TTBA*, including the trace of different activities in the context of a project, as well as the developers' contribution to social software platforms. We extend *TTBA* to consider these multiple sources of expertise, each one with a different weight. Next, we examine the performance of this *Multisource* variant of *TTBA*, and demonstrate that some technical (e.g., text of previous bugs or commit messages) and social (e.g., bug comments) contributions of developers are more important than the others.

This section is prepared for submission to IET Software journal.

Abstract

Bug assignment (BA), the process of ranking developers according to their potential ability to fix a given bug, is an important software-engineering task, which requires two key pre-requisites: (a) the development of an expertise profile for each developer, and (b) the formulation of a similarity metric, based on which to estimate the relevance of a developer to the bug in question. Developers make different types of contributions to their projects, and each of them typically leaves a trace in the team repository, which then raises the following question: "what is the information value of these various traces towards developing an accurate profile for the developers' expertise?" In this chapter, we report on a study designed to address this question. This study makes the following contributions. 1) We investigate the information value of different pieces of information in open-source repositories for BA. We show that in addition to *bug-fixing* contributions, other *technical* and even *social* contributions of the developers within the version-control system are useful information for BA. 2) We provide a curated, up-to-date data set[1] including multiple types of information, all relevant evidence of developers' expertise, for 13 popular open-source projects in Github. To the best of our knowledge, this data set is the most comprehensive one, currently available for bug-assignment research. 3) We demonstrate how the effectiveness of the bug-assignment process may be improved with this comprehensive expertise information, using our TTBA similarity metric on the above data set.

[1]The data set, source code, documentations and detailed output results are available at:
https://github.com/TaskAssignment/MSBA-outline
This data set includes more than 93k BAs as well as other technical and social contributions of 30k developers.

4.1 Introduction

Bug assignment (BA) aims at identifying the most appropriate developers to fix a given bug. The problem is usually formulated as ranking a number of available developers to fix the bug (Bhattacharya and Neamtiu 2010; Matter et al. 2009; Tamrawi et al. 2011b). This was previously done by applying computing methods (like machine learning or similarity methods) on a set of evidence of expertise of developers to recommend the proper developers to fix the available bug. At a high level, two set of factors are important in BA research and can affect the quality of the assignee recommendations: (a) the method for estimating developer-to-bug relevance, and (b) the data elements considered for formulating the developers' expertise.

Methods: Previously, researchers used machine learning, Information Retrieval (IR), Fuzzy, Social Network Analysis and other methods to solve this problem (Bhattacharya and Neamtiu 2010; Matter et al. 2009; Shokripour et al. 2012; Tamrawi et al. 2011b; Xie et al. 2012; T. Zhang et al. 2016). During time, researchers tried to enhance the assignment process by improving the technical methods (Aljarah et al. 2011; Khatun and Sakib 2016; Tamrawi et al. 2011a). The methods became more intelligent and precise by providing more accurate features, better filters, more explicit separation between elements of data (e.g., parts of speech in an NLP method) and so on. Recently, some research appeared to combine two or more of those methods to propose more efficient methods (T. Zhang et al. 2016; W. Zhang et al. 2016b).

Data: Regarding the developers' expertise, almost all the previous researchers used the previous bug fixing history (Anvik et al. 2006; Khatun and Sakib 2016; Matter et al. 2009; Tamrawi et al. 2011b; Xie et al. 2012; W. Zhang et al. 2016b). Some researchers considered more evidence like developers' code (Hossen et al. 2014; Hu et al. 2014), meta data –e.g., product and component– of previously fixed bugs (Aljarah et al. 2011; Anjali et al. 2016) and tossing history of the previous bugs (Bhattacharya and Neamtiu 2010; Jeong et al. 2009). Again, recently some studies combined several sources of data to obtain better results (T. Zhang et al. 2016; W. Zhang et al. 2016b).

As most previous research in this area focused on improving the *methods* (and yet more efficient methods are needed), enhancing upon the usage of *data* still has not been addressed much, although it is not less important in terms of capability of enhancing the quality of assignee recommendations. The previous BA research only addressed some of the basic elements of *data* –mostly the previously fixed bugs or code commits. There is a huge potential about *what* data to utilize and *how*, in order to improve the performance of bug-assignments.

To the best of our knowledge, no major effort has been done in previous research to investigate the effectiveness of different pieces of information in BA.

In our previous research, we devised a new method, called Thesaurus and Time Based Bug-assignment (TTBA) (Sajedi-Badashian and Stroulia 2018c). TTBA relies on two key intuitions. First, instead on analyzing the complete text of bug descriptions and developers' bug-fixing comments, it only examines a thesaurus of software-engineering terms, obtained from the keywords in Stack Overflow. Second, instead of analyzing a developer's record as a whole, it analyzes it as a time-stamped sequence of sub-documents, each one corresponding to a developer contribution to the project; in this manner, it weighs recent expertise more than past expertise. In a previous study, we demonstrated TTBA's effectiveness and showed that it outperforms state-of-the-art methods. In its original formulation, TTBA only considers bug-fixing contributions of developers. In this study, we extend the original TTBA formulation to take into account more types of evidence of expertise, in order to investigate the following research questions:

1. How can we combine various sources of expertise of developers to obtain an accurate bug-fixer recommendation?

2. What is the information value of different sources of expertise of developers in open-source projects for bug-assignment? How useful are social and technical contributions of developers (other than previous bug fixing history)?

The answers to the above questions are useful for future research aiming to extend previous approaches around those grounds and to utilize more effective methods of assigning bugs to developers. While it is impossible to consider all different types of data and combinations, we address the above questions in some simplified settings. In a nutshell, in this study, we make the following contributions:

1. We extend the TTBA method, to include multiple sources of expertise of developers. We show that extending the sources of expertise can improve automatic BA and the obtained enhancement is non-trivial.

2. Using our extended method, we investigate the information value of different pieces of information available in social and technical contributions of developers in open-source projects in Github. We show that while using pull requests' information have minor

improvement over the accuracy of BA, commits, bug comments and their traces to the developers' names have major improvements.

In order to validate the effectiveness of our approach (first part) and to investigate the effect of each piece of information in BA (second part), we provide a data set including social and technical contributions of developers in 13 open-source projects in Github during 7 years. We publish this data set for further researchers to replicate our study, or to be used for other BA studies as a rich data set.

4.2 Background and Related Research

During last decades, software engineering community has performed plenty of research addressing bugs and triaging problems. One of these problems is Bug Assignment (BA); Given a new bug report, identify a ranked list of developers, whose expertise (based on their record of contributions to the project) qualifies them to fix the bug (Bhattacharya and Neamtiu 2010; Hu et al. 2014; Khatun and Sakib 2016; Matter et al. 2009; Shokripour et al. 2015); This is the most prevalent formulation of the BA task, which we address in our BA approach –ignoring other BA-related scenarios and formulations; "team BA" (Jonsson 2013; Jonsson et al. 2016), resolution-time minimization (Nguyen et al. 2014; Park et al. 2011; Park et al. 2016) and multi-objective BA (i.e., maximizing expertise while minimizing time, effort and cost) (Karim et al. 2016; Liu et al. 2016).

As we discussed before in Section 2.3, previous research addressed this problem using different approaches, from machine learning to social network analysis and natural language processing methods. Since most previous studies tried to enhance the methods, less effort been dedicated for expanding the sources of expertise, and –to the best of our knowledge– no previous research tried to investigate the effectiveness of various pieces of information which are easily available in the open-source repositories. In the rest of this section, we summarize the types of information used in the literature.

4.2.1 Knowledge assumptions of Bug-Assignment

Almost all the previous approaches try to somehow match between the needed expertise (i.e., the new bug report) and the developers' expertise (i.e., their previous records), in order to find the best fit. Table 4.1 shows a survey of the information used for obtaining

developers' expertise for this matching in the same 13 studies of Section 2.3. There is a "meta" field which is representative of a wide range of meta-data elements like product and component of the handled bug or other details. Although most of the previous studies used the title and description of new bug (as the needed expertise) and previously fixed bugs (as developers' evidence of expertise), the extent to rely on these textual elements highly differs from method to method.

Table 4.1: A review of the used information for bug assignment in previous research

Method	Developers' expertise info				
	Bug fixing; title / description	Being a committer	Tossing history	Meta	Changed code
(Čubranić and Murphy 2004)	✓				
(Canfora and Cerulo 2006)	✓				
(Jeong et al. 2009)	✓		✓		
(Matter et al. 2009)	✓	✓			✓
(Bhattacharya and Neamtiu 2010; Bhattacharya et al. 2012)	✓		✓	✓	
(Tamrawi et al. 2011a; Tamrawi et al. 2011b)	✓				
(Shokripour et al. 2012)	✓	✓			
(Cavalcanti et al. 2014b; Cavalcanti et al. 2016)	✓			✓	
(W. Zhang et al. 2016b)	✓	✓		✓	
(Sun et al. 2017)		✓		✓	✓

For example, text-based approaches mostly rely on bug reports' direct information. The Naïve Bayes classifier of NB1 (Čubranić and Murphy 2004), Vector Space Model (VSM) of Develect (Matter et al. 2009) and Fuzzy model of Bugzie (Tamrawi et al. 2011a; Tamrawi et al. 2011b) are examples of these approaches. Albeit they may count on limited meta-data (e.g., component and product of each bug report) to make their decisions more local, the main determinant part of the data is the bug reports' textual information (mostly title and description).

On the other hand, there are approaches that use various information other than bugs' textual elements. Tossing-graph based approach of TG1 (Bhattacharya et al. 2012), social-network based model of KSAP (W. Zhang et al. 2016b) and the source code analysis method of iMacPro (Hossen et al. 2014) are examples of those approaches. Those are dependent on a great deal of non-textual information. Even in some cases, like Visheshagya (Anjali

et al. 2016), they do not use the bug reports' textual information at all, but just meta-data information. In all the cases, they utilize that information to localize the bug or to connect it to developers as possible fixers.

Admitting the fact that the second category can be very varied in dependence on those meta-data, we focus on the first category, i.e., text-based approaches. We investigate the textual elements available in bug reports in open-source projects in Github as the most popular version control system (VCS). We consider the textual descriptions of bug reports as evidence of expertise of developers who fixed them. We also consider several other text-based sources of expertise which are available in the VCS.

Figure 4.1 shows a sample bug report and some of its data resources. Each bug has a short title and a longer description. In the page of each bug in Github, the developers can comment and discuss about the bug. Sometimes they mention each other's login names to take a look at the bug. In addition to bug reports, commits and pull requests also contain textual elements that can be representative of expertise of their author. Furthermore, the developers can comment on bug reports, commits and pull requests. All these sources contain useful information that guide us about expertise and interests of developers. Interestingly, there were only a few studies that made a limited use of only some of those elements – e.g., commit messages in IE1 (Shokripour et al. 2012) and number of comments in $\text{REP}_{\text{topic}}$ (T. Zhang et al. 2016). We hypothesize that some of those sources can be valuable in indicating expertise for their author and investigate their usefulness in BA.

Since the scope of available sources of expertise are too ample, and the usefulness of each piece highly depends on the utilized method, addressing the above problem needs a simplified version of data on a more controlled method. So, we narrow down the "method" to our previous method (Sajedi-Badashian and Stroulia 2018c), called TTBA, and consider it as the baseline method –since it was the latest text-based IR model for BA and outperformed previous state-of-the-art approaches. We further revise it to handle multiple sources of expertise and show that the improvement is non-trivial. We also narrow the "data" to developers' previous bug fixing histories, commit and pull request messages and various social contributions (e.g., commenting and linking to developers) inside an open-source project. Then we investigate the information value of each piece by examining their effectiveness in the mentioned method, and how they can enhance the quality of BA.

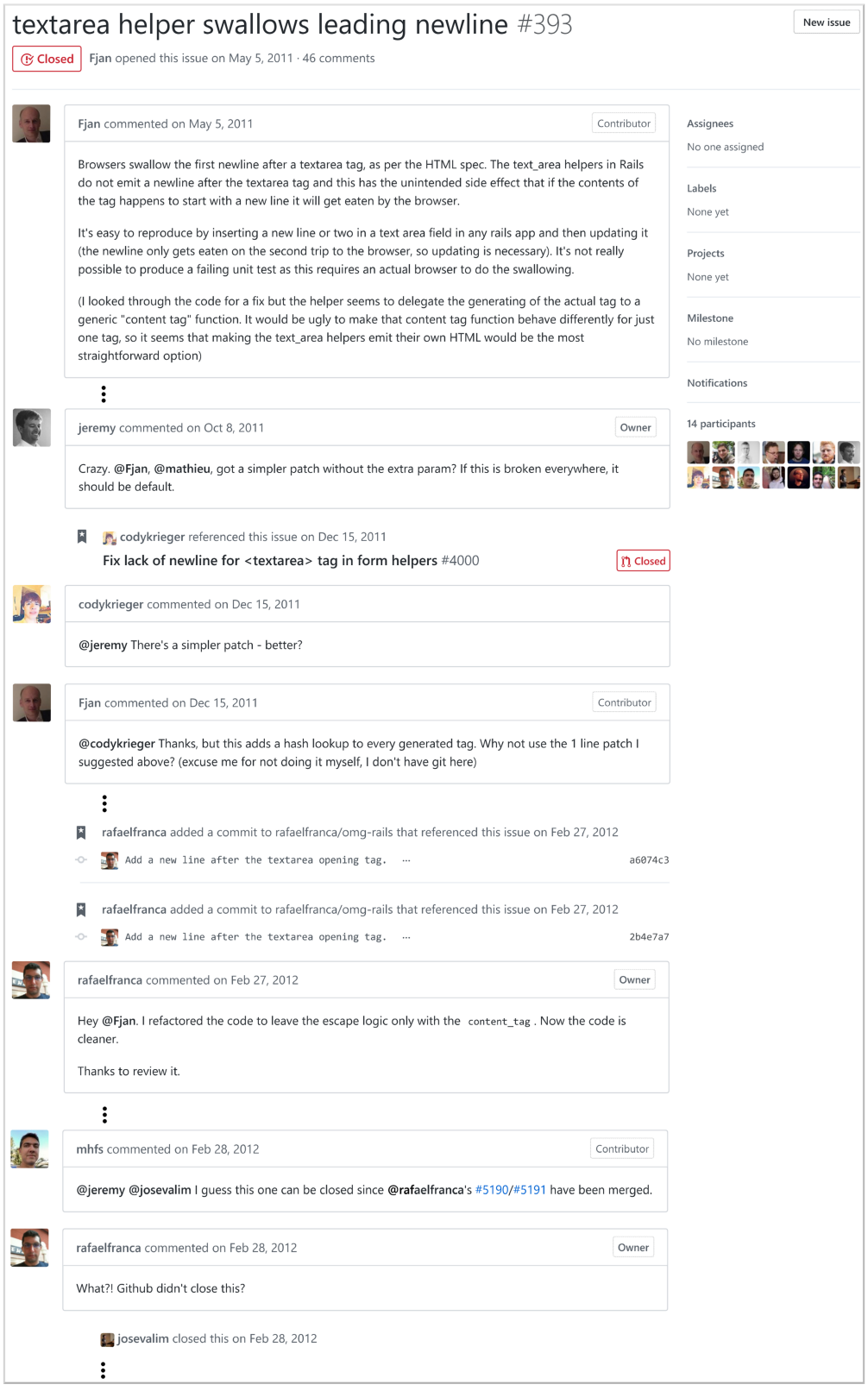

Figure 4.1: A sample bug report and information sources available around it

4.3 Bug Assignment Based on Multiple Sources of Evidence of the Developers' Expertise

In this section, we expand our method introduced in the previous section, *"thesaurus and time-based BA"* (TTBA), to include multiple sources of evidence of developers' expertise. This expansion is the answer to our first research question.

4.3.1 TTBA: A Compositional Similarity Metric for Bug Assignment

In the previous chapter, we proposed a new method called TTBA which is based on original *TF-IDF* and showed that it outperforms many other recent approaches. The premises of *TTBA* is that the expertise of a developer is represented as a document, constructed through the concatenation of several different sub-documents. Each of these sub-documents corresponds to a previous bug-fix by the developer. We considered the title and description of bug reports as the text of these sub-documents. Then, for each of the new bug reports, we used Equation 3.10 to calculate the *TTBA* score of each document (developer's expertise) and sort them from high to low.

Despite *TTBA*'s satisfying job towards reflecting developers' bug fixing history in recommending developers to fix a new bug, it is only capable of capturing bug description as the only source of developers' expertise. While there are abundant of other evidence of expertise in developers' tracks in Version Control Systems (VCS) and other technical social networks, as we mentioned before, it is desired to know the information value and importance of each piece of those information in BA research and also to utilize them in our assignment methods.

In order to achieve this, we adapt *TTBA* to be able to consider multiple sources of expertise. We inject the factor $context(sd_j)$ into our previous scoring function:

$$score(q,\ d) = \sum_{t \in q} freq(t,\ q) \cdot w(t) \cdot \left(\sum_{j=1}^{n_d} tf(t,\ sd_j) \cdot recency(sd_j) \cdot context(sd_j) \right) \tag{4.1}$$

The above formula is also called *Multisource* (multi-source version of *TTBA*). In the above equation, d is the document (developer's expertise), t is any distinct term in q (i.e., query or description of new bug), n_d is the number of sub-documents in d (e.g., the

number of previous assignments of any bugs to the developer d) and $w(t)$ is the weight of the term t (which is a fixed value for that term and is between zero and one). Sd_j is the j^{th} sub-document of document d (j^{th} evidence of expertise of developer d). $Freq(t, q)$ is a parameter that indicates the frequency of the technical term being considered in the query. It is obtained by dividing the frequency of the term in the query by the length of the query. $Tf(t, sd_j)$, term frequency, measures the number of times a term, t, is appeared in the sub-document sd_j, normalized by sub-document length. The factor indicating "time" of sd_j is $recency(sd_j)$. Again, it is between zero and one; it gets close to one for the recent evidence and close to zero for old ones. *Recency* is applied in order to take into account the change in developers' interest and expertise during time, as well as their effective engagement in the project.

Finally, $context(sd_j)$ reflects the importance of sub-document sd_j based on its type (bug description, commit message, pull request message, bug comment message, etc.) and is fixed for all sub-documents of each type. It can be between 0 and 1 based on the importance of the sub-document. It gets higher values for the important sources of expertise and lower for the minor document types. We will obtain proper context weights in the Section 4.5.1. Our approach uses this equation for giving a score to each developer regarding a given bug. For simplicity, we call our approach *Multisource*.

4.4 Experimental Design

We used our *Multisource* approach –as the expansion of our previous method *TTBA*– and ran experiments with two goals: 1) to inspect the information value of various sources of expertise available in open-source projects and 2) to validate the usefulness of this adjustment –as a practical enhancement. We used a big Github data set including thousands of bug reports, developers and their various types of evidence of expertise for this purpose.

In order to rely on realistic results, we separated the first part (which also includes tuning) from the second part (which is our main validation). We selected three projects as tuning / test projects (one from each category; small, medium and large)[2] and performed all the tunings using those projects. To avoid bias, later, we excluded these projects from

[2]We divided the projects into three categories; projects with 1k-, 1k to 10k and 10k+ bug-assignments. There are 4, 4 and 5 projects respectively in small, medium and large categories. Then, we selected the smallest of each category for the first part (including tuning). The idea was to tune on relatively low number of bugs and leave more bug reports for the main experiment. The selected projects are: lift/framework, fog/fog and adobe/brackets. The percentage of bug reports used for the first part is ∼13%.

our second experiment. For both experiments, we used our *Multisource* scoring function (Equation 4.1) to calculate a score for each developer and ranked them for each bug report. The two experiments are as follows;

1) First, we investigated the usefulness of different pieces of information. Since our scoring function can consider several types of sources of expertise of each developer for a bug report, in each step of this experiment, we added one piece of information (i.e., considered one more evidence of expertise) to see if adding the new resource improves the accuracy and if so to what extent. we used extensive 2d exploration (Blanco and Lioma 2012; Craswell et al. 2005) to obtain the order and importance of various sources of information and the best context scores for each type of evidence. We considered the evidence of expertise in two categories; first, we investigated additional textual information –i.e., "previous bug fixing history", committing, submitting pull request and three commenting evidence (on bugs, commits and pull requests). Then, we considered links to developers' Github login names from any source that is related to a bug as an additional source of expertise regarding that bug. As the result of this first experiment, we obtained the most important sources of information for BA, and their weights which are used in the next part.

2) In the second experiment, we validated our approach using all the projects (to avoid bias, we eliminated the test projects for this experiment). We used the same similarity metric, but the best obtained settings (i.e., context scores) from the first experiment to calculate the score of each developer for each bug report. Then, we ranked the developers based on this score, calculated the final evaluation metrics and compared the results against our previous approach, *TTBA*. All the results are available online[1].

4.4.1 Data set

We used a data set for Stack Overflow tags and a main data set for various sources of evidence of expertise of developers.

For obtaining a list of Stack Overflow tags and their weights to be used in our main *Multisource* formula (as the term weights), we used a recent Stack Overflow data set (approximately 100 GB) (I. Stack Exchange 2017) including information of the tags, users and their contributions (e.g., posts). Out of the total 46,278 available technical terms (tags), we excluded the tags with one or two letters (to avoid misleading) and obtained 46,115 tags with three letters or more. Then, using Equation 3.5, calculated their weights based on their recent usage statistics in around 32 million posts.

Our main data set is the extension of the data set we mentioned in Chapter 2. It is obtained from Github and is used for evaluating our multi-source BA method and comparing its results against real data. It includes the same set of projects as in Chapter 2. Figure 4.2 shows the data model for this data set. Details of number of different items related to developers' expertise (including bugs, commits, pull requests and their comments) are shown in Table 4.2. References to developers' login names are contained in bugs, commits and their comments (we will show some statistics about those references in Table 4.4). In terms of number of bug reports, number of developers in the projects and inclusion of variety of sources of expertise, our data set is one of the most extensive data sets used for BA.

We made the preliminary data set (final list of tags and their weights) as well as the above "expertise" data set available online[1] for further processing and usage.

Table 4.2: Data set different evidence of expertise in 13 projects

Project	#of devs	# of bugs	# of assignments	# of commits	# of pull requests	# of bug comments	# of commit comments	# of pull request comments
framework	75	325	566	3,315	405	3,782	175	1,695
html5rocks	159	627	998	4,056	797	1,167	41	750
yui3	175	526	902	25,381	1,154	1,977	344	4,272
khan-exercises	206	624	857	7,148	1,246	1,228	474	58,751
ghost	473	3,578	6,142	6,974	3,912	14,927	129	8,859
fog	770	1,124	1,327	12,183	2,686	7,003	335	10,789
julia	831	9,086	12,748	34,095	8,482	79,165	10,848	56,364
brackets	864	6,255	10,462	17,099	4,921	39,179	365	20,929
travis-ci	1,159	5,473	6,334	3,607	210	32,035	162	517
elasticsearch	1,262	10,423	16,184	24,798	9,366	41,379	690	33,526
salt	2,283	10,237	15,533	74,805	23,076	52,407	1,831	27,591
Angular.js	2,386	7,402	9,658	8,117	7,124	41,356	1,581	30,793
Rails	4,079	8,794	11,305	59,758	17,424	36,335	13,180	43,743
Total	union: 7,438	64,474	93,016	281,336	80,803	351,940	30,155	298,579

4.5 Value of Various Sources of Expertise

The sources of expertise include two sets of information; textual data inside project and references to the developers' login names. Starting from "previous bug fixing history", each time we add an extra type of information and see if we can enhance the final overall MAP over the three test projects –and if so, we obtain the best context score in Equation 4.1 for that type of information. We investigate the value of the two mentioned sets of information in two separate sub-sections.

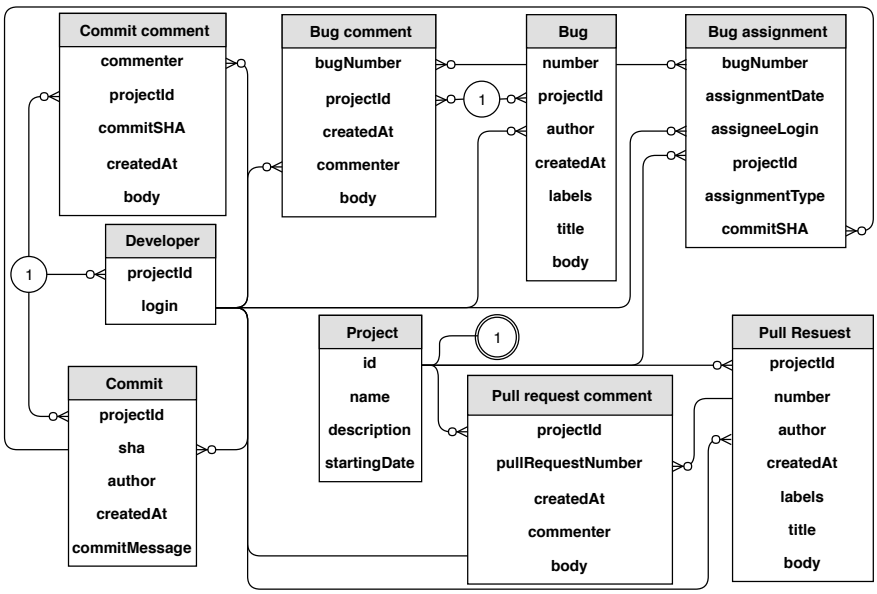

Figure 4.2: The data model of our Github data set

4.5.1 Considering additional textual information as evidence of expertise

We consider bugs, commits, pull requests and the comments of each of them as evidence of expertise of developers regarding the keywords (Stack Overflow tags) mentioned in them[3]. Before we consider the exact effect of each source of information on the resulted MAP, we identify the order of importance of each source in a pre-processing experiment. We start by "previous bug fixing history" and consider each of the evidence as an addition to it. Items that increase final MAP the most are considered the most important sources. The order of importance of these elements is shown in Table 4.3 (from left to right). We found that *bug comment*, *commit* and *pull request* are in order the most important sources.

Then we follow this ordering in obtaining appropriate context scores for each element. Table 4.3 shows the results. The first row has no extra information and just includes the *previous bug fixing history*. Then we add *bug comment* and obtain its best coefficient (which produces the highest MAP). We do this using extensive 2d exploration (Blanco and Lioma 2012; Craswell et al. 2005) in a mutation experiment –also called grid search (Tatsis and Parsopoulos 2016) or exhaustive parameter search (Zhai and Lafferty 2002)– considering values between 0.0 and 1.0 with increments of 0.1[4] as the coefficient of *bug comment*. The

[3]The usefulness of considering only Stack Overflow tags instead of all the keywords was proved in our previous research (Sajedi-Badashian and Stroulia 2018c) and we just rely on it here.

[4]The 0.1 is an arbitrary value. It can be any small value. Note that if it is too small, then too many

Table 4.3: Determining context weights for additional pieces of information in test projects

Coefficient					MAP
bug comment	commit	pull request	pull request comment	commit comment	
-	-	-	-	-	0.5448
0.4	-	-	-	-	**0.5564**
0.4	**1.0**	-	-	-	**0.5640**
0.4	1.0	**0.7**	-	-	**0.5661**
0.4	1.0	0.7	**0.0**	-	0.5661
0.4	1.0	0.7	0.0	**0.4**	**0.5662**

best obtained weight for *bug comment* is 0.4 (see the second row in the table). Lower coefficient values underestimate its effect and higher values inflates noise in the data. In the next step, we consider *bug comment* with its obtained coefficient (0.4) in addition to the *previous bug fixing history* and add *commit* to obtain it best coefficient using 2d exploration. The best obtained weight for *commit* is 1.0. This does not indicate that the commits are more important than bug comments. In fact, we have obtained the order and importance of the fields in the first place (i.e., as shown in Table 4.3 from left to right). The higher coefficient for commit means that there is less noise in the commits since they are brief and concise.

We follow the same approach to obtain appropriate context scores for other elements. The obtained coefficient in each step is shown in bold and the achieved MAP is shown in the last column. In general, *bug comments*, *commits* and *pull requests* which also obtained the highest orders have the most effects on the final MAP. The bug comments are important since they are directly related to the bug to be fixed. They appear like a discussion thread under the bug, including keywords from that bug which makes them valuable. Commits and pull requests are important in that both are directly related to the *code*. A committer and submitter of a pull request have written some code in the project and in forked projects (which are used in the main project) respectively. Both have done some efforts towards the code. The reason commits are found more effective can be regarding the fact that pull requests can be merged into different branches but the commits we captured are all related to *master* branch. Also commit texts are usually shorter and including just clear technical data. This makes the pull requests less important than the commits.

As mentioned above, the value of the weights does not show the item's importance. We

different configurations would be generated to test. If it is too big, then we might loose some good configurations.

obtained the order and importance of each item in the first place by comparing their effect on the final MAP in isolation (i.e., determining how much the MAP increases if we add only one source of expertise to "previous bug fixing history"). The obtained value of weights can be interpreted as the level of noise in each item. For example, the commits include low level of noise since they are brief and to-the-point and have the weight "1.0". Pull requests are longer and have lower weights (0.7). In all the cases, comments of an item (e.g., commit comment) have lower weight than the item itself (e.g., commit). Adding "pull request comments" has no positive effect on the final MAP, so we remove it from the considered evidence (the coefficient "0" is selected for it). This can be because of lower importance of them in presence of better evidence (e.g., bug comments and commits), or because of noise in them.

We can conclude that additional information from open-source projects can be very useful for BA. Especially using the text of bug comments, commits and pull requests as evidence of expertise of developers regarding the keywords mentioned in them enhances the assignee prediction.

4.5.2 Considering References to the developers' names

Next, we consider the references to the developers' login names from sub-documents that are related to a bug report, i.e., the description or the comments of that bug report, the related commits to that bug report or the comments of those related commits. We cannot consider other evidence like pull requests since relating them to a bug is not common (it is common to mention "fixes bug # x" in a commit message, but not in the text of a pull request).

We hypothesize that if a developer is referenced somewhere from the social or technical correspondence around a bug, then that developer has a high chance of being related to that bug and / or work towards fixing it later.

Table 4.4: Statistics of the references to the developers' names in three test projects

Source of reference	# of references	# of positive references	Positive reference ratio (%)	# of referenced developers	# of referencing bugs
Bug	781	141	18.05	79	324
Comment of a bug	14,377	2,558	17.79	833	4,758
Commit related to a bug	2	0	0	2	2
Comment of a related commit	0	0	0	0	0
All types	Sum: 15,160	Sum: 2,699	Union: 17.80	Sum: 914	Union: 4,856

We do a short feasibility study in the three test projects, to check the appropriateness of

the referenced developers to fix the related bug. Table 4.4 shows the statistics about these references from the four mentioned resources (bug, bug comment, commit related to a bug and comment of a commit related to a bug) in the three test projects of our data set. The first three columns show the number of references (to developers) found in each source of information and the number of positive references as well as their ratio with respect to the total references from each resource. Positive references are those references (or links) leading to a bug fix by the referenced developer. As indicated in this table, the developers referenced directly from a bug or its comment, have a high chance of being a fixer of that bug later (∼18%). Note that referencing a developer from a comment of a bug is more common than from the bug itself (compare the 14,377 references from bug comments against 781 references from bugs). This is because the developers communicate in the comments of the bugs for several purposes including suggesting developers to fix the bug.

We noticed that many developers (914) are referenced from many bugs (4,856), but not all the references were dealing with the correct assignee (∼18% of the references were positively linking the correct assignee). We checked the test projects in our data set and found that in many cases the developers are referenced for the purpose of communication and information sharing rather than recommending fixing the bug. As a result, we need a method to avoid false positives. To achieve this, we maintain a list of "positive" referenced developers during time. At each point in time, we just consider references to those developers with a good "reference history" in the past (e.g., at least half of their previous references lead to work towards fixing the related bug).

To take into account the positive references in our recommendations, we prioritize the positively referenced developers (who have been referenced from anywhere related to a bug) over all other developers. In order to do that, we calculate the score for the developers and rank them in two separate groups; first, we consider those positively referenced developers in one group, and then the rest of developers in another group. Note that since the text containing the reference is not written by the referenced developer, we can not consider a separate *sub-document* and a separate $context(sd_j)$ coefficient for the referenced developer in Equation 4.1. That is the reason we prioritized the positive referenced developers against other developers.

Like before, we obtained the order of importance of each type of reference (from bugs, commits or their comments) in a separate pre-processing experiment (not shown here) in our three test projects. Types of references that increase the final MAP more are considered

more important. For each reference type, we examined the value of MAP considering that reference type, as an addition to the best configuration we obtained before in Table 4.3. That best configuration is again shown in the first row in the table 4.5 which shows the order of importance of the references. We found that *bug comments* and *bugs* are in order the most importance references to developers' login names.

Since we used the references to prioritize the developers (i.e., consider the positively referenced developers in a separate pool, before the other developers) rather than changing the developers' score (i.e., adding an entry to Equation 4.1), there is no need to obtain different context scores for the references. So, we do the step-by-step process just to determine whether using the new reference improves the final MAP or not.

The results are shown in Table 4.5. Both the references from *bugs* and *bug comments* are effective and increasing the MAP. The number of references from *commits* and *commit comments* are zero in these three projects, but they might be non-zero for the rest of projects, so we did not omit them from our main experiment; we just considered them in the next priorities.

Table 4.5: Determining whether to prioritize the referenced developers or not

bug comment	bug	commit	commit comment	MAP
-	-	-	-	0.5662
✔	-	-	-	**0.6349**
✓	✔	-	-	**0.6374**

"Bug comments" and "bugs" are the most important resources with regard to references to developers. It is interesting that "bug comments" are even more important than "bugs". As mentioned before, one reason can be that referencing developers from a bug description is not so common. Instead, there are sometimes long discussions under a bug report mentioning who may be a good fit to fix this bug, what is the right way of dealing with it and similar concerns.

4.6 Validating *Multisource* Approach on the Whole Data Set

In this section, we validate our *Multisource* approach. We report its results after final run on all projects (excluding the three test projects), with the configuration we obtained from

our test projects. In order to validate our approach, we used the exact implementation of our previous approaches, *TTBA* (Sajedi-Badashian and Stroulia 2018c) and made the *Multisource* enhancements in it (to include multiple sources with various weights). We compare the results regarding overall and per-project values.

Table 4.6: Comparison of the results of our approach against *TTBA*

Project	**Method**	**Top1 (%)**	**Top5 (%)**	**Top10 (%)**	**p@1 / r@1**	**p@5 / r@5**	**p@10 / r@10**	**MRR**	**MAP**
Html5 rocks	*TTBA*	**66.63**	87.17	92.69	**0.67** / **0.57**	0.20 / 0.82	0.11 / 0.91	0.76	0.71
	Multisource	**66.63**	**90.88**	**95.19**	**0.67** / **0.57**	**0.22** / **0.88**	**0.12** / **0.94**	**0.77**	**0.74**
Yui3	*TTBA*	50.11	75.83	84.81	0.50 / 0.44	0.17 / 0.73	**0.10** / 0.83	0.62	0.59
	Multisource	**50.89**	**79.16**	**86.81**	**0.51** / **0.45**	**0.18** / **0.76**	**0.10** / **0.85**	**0.64**	**0.61**
Khan-exercises	*TTBA*	**57.18**	85.41	89.26	**0.57** / **0.52**	**0.20** / 0.84	**0.10** / **0.89**	**0.69**	0.67
	Multisource	56.83	**86.23**	**89.96**	**0.57** / 0.51	**0.20 / 0.85**	**0.10** / **0.89**	**0.69**	**0.68**
Ghost	*TTBA*	**59.39**	85.27	91.66	**0.59** / **0.46**	0.22 / 0.82	0.12 / 0.91	0.70	0.67
	Multisource	57.52	**89.40**	**94.50**	0.58 / 0.44	**0.23** / **0.87**	**0.13** / **0.94**	**0.71**	**0.68**
Julia	*TTBA*	45.67	71.80	81.02	0.46 / 0.40	0.16 / 0.69	0.09 / 0.80	0.57	0.55
	Multisource	**46.78**	**78.37**	**86.78**	**0.47 / 0.41**	**0.18** / **0.76**	**0.10** / **0.86**	**0.61**	**0.59**
Travis-ci	*TTBA*	**57.47**	74.74	78.09	**0.57** / **0.55**	0.16 / 0.73	0.08 / 0.77	0.65	0.64
	Multisource	56.20	**80.06**	**82.89**	0.56 / 0.54	**0.17** / **0.79**	**0.09** / **0.82**	**0.66**	**0.66**
Elastic search	*TTBA*	47.61	73.97	84.71	0.48 / 0.44	0.16 / 0.71	0.09 / 0.82	0.59	0.57
	Multisource	**52.33**	**80.34**	**88.95**	**0.52 / 0.48**	**0.17** / **0.78**	**0.10** / **0.87**	**0.65**	**0.62**
Salt	*TTBA*	43.28	72.63	81.14	0.43 / 0.36	0.17 / 0.69	0.10 / 0.79	0.56	0.53
	Multisource	**44.78**	**79.52**	**86.89**	**0.45** / **0.37**	**0.19** / **0.76**	**0.11** / **0.85**	**0.60**	**0.57**
Angular .js	*TTBA*	46.47	79.87	86.19	0.46 / 0.42	0.17 / 0.77	**0.10** / 0.85	0.60	0.58
	Multisource	**47.98**	**83.05**	**88.70**	**0.48** / **0.44**	**0.18** / **0.81**	**0.10** / **0.87**	**0.63**	**0.61**
Rails	*TTBA*	41.42	69.91	78.69	0.41 / 0.36	0.15 / 0.66	0.09 / 0.76	0.54	0.51
	Multisource	**44.68**	**75.48**	**83.04**	**0.45** / **0.39**	**0.17** / **0.72**	**0.10** / **0.81**	**0.58**	**0.55**
Total (10 proj)	*TTBA*	47.50	74.73	82.93	0.48 / 0.42	0.17 / 0.72	**0.10** / 0.81	0.59	0.57
	Multisource	**49.31**	**80.36**	**87.36**	**0.49 / 0.43**	**0.18** / **0.78**	**0.10** / **0.86**	**0.63**	**0.60**

Table 4.6 shows the final results. The overall MAP for 93k BAs of all the 10 projects in *Multisource* approach is 0.60. Compared to *TTBA*, it is improved by ∼3%. Regarding project-specific results, the improvement is between 1 to 5 percent (*Multisource* works better in all the 10 projects). Regarding other metrics, in almost all the cases *Multisource* works better. There are only a few exceptions; in two small projects (Html5rocks and Khan-exercises)[5] and two medium projects (Ghost and Travis-ci) *TTBA* works equally or slightly better regarding @1 values. It seems that superiority of *Multisource* against *TTBA* is more evident in large projects rather than small ones. We hypothesize that this difference arises from the size, time duration and availability of "multi-source" evidence.

To investigate on this, we compare the available multi-source evidence in the two groups of projects; group 1 includes the above four projects that have slightly better @1 accura-

[5]Note that based on size of projects we divided them into three categories; small, medium and large.

cies for *TTBA* and group 2 includes the other six projects in which regarding all metrics *Multisource* obtained better results. Group 1 includes two small and two medium projects. In group 2, only Yui3 is small, Angular.js is medium and the other four are large projects. Results of this comparison are shown in Table 4.7.

Table 4.7: Available multi-source data in different projects

	Group 1: 4 projects (Html5rocks, Khan-exercises, Ghost and Travis-ci) Best MAP: *Multisource* Best @1: *TTBA*	Group 2: other 6 projects (Yui3, Julia, Elasticsearch, Salt, Angular.js and Rails) Best on all metrics: *Multisource*
Number of BAs per project	3,582	11,055
Number of references to developers' names per bug	0.95	1.26
Average length of bugs and their comments	26.96	40.57
Average project duration	53	65

As shown in this table, the projects of group 2 have much more multi-source evidence than the other group. Those information, as sources of expertise, are the key to success of *Multisource*. Lack of information especially affects the first recommendation. Without enough multi-source evidence, the success of *Multisource* would be chancy and the @1 results are more vulnerable. Note that even though in 4 projects of group 1, regarding three @1 metrics, *TTBA* is competing against *Multisource*, still regarding MAP and other 7 metrics *Multisource* performs better. The intuition is that the multi-source evidence can sometimes be noisy and misleading regarding the first recommended rank (i.e., when there not enough data) and produce no better @1 accuracies, but even in those cases, the next recommended developers include more ground-truth assignees and hence the final efficiency criterion, MAP is enhanced. In other words, with no extra evidence of expertise, the *Multisource* approach of Equation 4.1 would be the same as *TTBA*. Little extra evidence (as in small projects) can slightly enhance the results of *Multisource* or not (depending on the quality of additional evidence and existence of noisy data). When it comes to large projects with abundance of extra evidence (which are all in group 2), the effect of noise is reduced, and the predictions are more accurate. So, the difference between *Multisource* and *TTBA* is more evident. On the other hand, in large projects, there are too many candidates and it is difficult to identify the best developers to fix a new bug. Since *Multisource* benefits from a fine-grained weighting scheme that differentiates between many existing evidence of expertise, it obtains better results.

Overall, these results, as well as the other comparisons of MAP we established before, shows that our *Multisource* method outperforms *TTBA*.

Our approach is also fast. On a machine with Core i7 CPU and 16GB of RAM, we captured time-to-run for each project; in the most complicated case, which includes evidence from all our additional sources (bug, bug comment, commit, pull request and commit comment as well as links to developers' login names), it takes between 0.1 and 134.1 seconds to run for each project (average=32.9, median=11), depending the number of bugs to be assigned and number of additional sources. The average time for assigning developers to each bug in the whole data set is 4.6 milliseconds.

4.7 Threats to Validity

There are a few points about the sources of expertise and how we used them (to infer about the best developer to fix a new bug) as well as the generalization of our approach for further studies (for other projects or platforms including other types of expertise evidence) that we review and discuss as internal and external threats to validity.

Construct validity:
The most important validity threat is about the definition of assignee we used for the validation of our approach. We used the extensive definition of "the developer who is tagged as *assignee* at the closing time, the developer who closes the bug, or any other developer who works towards fixing the bug" (Sajedi-Badashian and Stroulia 2018c). One can argue that in practice, there might be discrepancies from the proper assignee. A developer may fix a bug, without being the best candidate for fixing that bug. On the other hand, in reality, a developer may have no connection with a given bug, nor have done any attempts towards fixing the bug (e.g., because of being busy with another bug, or due to other project circumstances), but still be a good possible fixer for the bug.

To address this issue, we argue that the best way to identify "all" the (possible) ground-truth assignees of a bug would be to discuss it with a project manager and tag all the appropriate developers as right assignees. Unfortunately, this plan is too expensive; it needs lots of time and resources to investigate about thousands of bugs and developers. To the best of our knowledge, no previous studies have done such an inspection towards validity of their approach. We did not investigate this in our projects either. Instead, we defined the "right

assignee" based on the most comprehensive definition in the literature (Sajedi-Badashian and Stroulia 2018c) to be inclusive of all the possible developers who worked towards fixing the bug and capture the most probable bug fixers. This is the most comprehensive definition of assignee that has been used up to now (Sajedi-Badashian and Stroulia 2018c). Also, we did our best to clean the data set and remove any bugs that are not closed, to prevent misrepresentations.

Internal validity:
The other challenge is related to our Github data. We just considered the data from the main branch of each project, which is called "master" by default. But there are evidence of expertise in other branches that are ignored, and could possibly change our results.

To address this issue, we argue that although this might be true, it would not have much effect on our results since most of the contributions of the developers are usually in the *master* branch. In addition, the contributions of the developers in the *master* branch are usually more important than the secondary branches.

External validity:
While we tested our *Multisource* approach using several open-source projects and their available data, one could suspect its usefulness in other (different) projects, networks or platforms with other types of evidence of expertise. For example, the practicality of our scoring function (Equation 4.1) might be questionable in proprietary software.

We argue that although our final scoring function (Equation 4.1) might not work perfectly in new settings, it has the capability of being tuned for different situations; first, the different context coefficients –$context(sd_j)$– for the new types of evidence can be fine-tuned using 2d-exploration (Blanco and Lioma 2012; Craswell et al. 2005). Second, for the situations with special backgrounds or vocabularies where the Stack Overflow tags are not representative of the most important keywords (like some proprietary software), other *idf*-like weighting schemes with proper indexing can be adopted rather than our term-weighting –$w(t)$. Finally, even the factors $freq(t, q)$ and $tf(t, sd_j)$ can be utilized in a different way that works better for those specific situations. All these adjustments can be met with a small percentage of bug reports in the projects, or a few test projects, to obtain desirable results in different setups. The important aspect is that the general scheme of our approach is easily supporting multiple sources of expertise with different importance (weights). Also, it benefits from a proper granularity of the technical terms and their time of usage by the developers.

4.8 Conclusions and Future works

Based on our previous approach, we developed a bug-assignment method that can consider and apply various sources of expertise all at the same time, but with different levels of importance. With its high granularity over different evidence of expertise and their technical terms, we showed that it is capable of highly accurate assignee recommendation and that it outperforms state-of-the-art methods.

In addition to enhancing our bug-assignment method, in this study, we investigated the information value of different pieces of data that are usually found inside open-source software repositories. We realized that different types of social and technical contributions of developers inside a software repository can be very useful for recommending proper fixers. Using our multi-source experiments, we showed that extending the sources of expertise from *only bug fixing* to diverse sources in open-source projects can propose non-trivial enhancements in automatic assignee recommendation accuracy. These valuable evidence of expertise are easily accessible in open-source projects for public and can be utilized for further research in this area.

Specifically, we found that the text of bug comments, commits or pull requests contain useful information about the expertise of their author (poster). In addition, the links to developers' login names from developers' contributions contain fruitful information about who might be a good candidate to fix a given bug.

We provided the data sets of our experiments online [1]. This is a rich set of textual information about bugs, commits, pull requests and social contributions of developers in the VCS that can be processed for further studies.

Despite the high extent of the provided data set in this study, still extending this study is possible by providing richer data; one can investigate the usage of more textual elements available in the open-source repositories (e.g., information available in other branches).

As another future work to this study, the notion of "ground-truth assignee" can be enhanced to include all the expert developers regarding the given bug. In order to obtain such a list for each bug, the project managers need to review the bug reports and recommend the set of appropriate developers for each one, regardless of other project constraints (and just based on developers' expertise). In a big project with lots of bug reports, this is expensive in terms of time and effort but would help to have a more appropriate evaluation.

Acknowledgements

The work is supported by Graduate Student Scholarship[6] funded by Alberta Innovates - Technology Futures (AITF)[7] and Queen Elizabeth II Graduate Scholarship [8] funded by Faculty of Graduate Studies and Research (FGSR)[9] at University of Alberta.

[6]https://fund.albertainnovates.ca/Fund/BasicResearch/GraduateStudentScholarships.aspx
[7]https://innotechalberta.ca
[8]https://www.ualberta.ca/graduate-studies/awards-and-funding/scholarships/queen-elizabeth-ii
[9]https://www.ualberta.ca/graduate-studies

Chapter 5

Utilizing Beyond-project Sources of Expertise for Bug Assignment

Preface

In this chapter, we investigate the usefulness of additional sources of evidence of developers' expertise beyond their activities in the project where the bug originates. We consider the developer's activity in Stack Overflow and in projects other than the project from which the bug under examination originates. We examine how the additional sources affect the overall bug-assignment process. We provide two scenarios for external sources of expertise:

1) The external sources are considered in the absence of internal sources, which would be realistic, for example, in the case of developers new to the project. In this case, we assume that we have the external contributions for all the developers in isolated settings[1]. The main contents of this chapter are devoted to this case. We also performed a preliminary feasibility study for this case which is shown in Appendix 5.A.

2) The external sources are considered in addition to internal sources to see if they provide any complementary information or not. Appendix 5.B discusses this case.

We show that in the ideal/isolated settings (the first case above in which external contributions of all the considered developers are available), usage of external sources is capable of accurate assignee recommendation. However, in general settings (when only partial external sources of expertise are added to the complete internal data), this does not enhance the prediction accuracy since accurate inside-project sources of expertise are available.

The contents of this chapter has been published at the 19th International Conference on Fundamental Approaches to Software Engineering (FASE 2016).

Appendix 5.A has been published at the 31st International Conference on Software Maintenance and Evolution (ICSME 2015).

[1]In this case, we filter and remove the developers with no external evidence of expertise and any bug report assigned to them. So we described it as "isolated settings".

Abstract

Bug triaging, i.e., assigning a bug report to the "best" person to address it, involves identifying a list of developers that are qualified to understand and address the bug report, and then ranking them according to their expertise. Most research in this area examines the description of the bug report and the developers' prior development and bug-fixing activities. In this chapter, we propose a novel method that exploits a new source of evidence for the developers' expertise, namely their contributions in Stack Overflow, the popular software Question and Answer (Q&A) platform. The key intuition of our method is that the questions a developer asks and answers in Stack Overflow, or more generally in software Q&A platforms, can potentially be an excellent indicator of his/her expertise. Motivated by this idea, our method uses the bug-report description as a guide for selecting relevant Stack Overflow contributions on the basis of which to identify developers with the necessary expertise to close the bug under examination. We evaluated this method in the context of the 20 largest Github projects, considering 7144 bug reports. Our results demonstrate that our method exhibits superior accuracy to other state-of-the-art methods.

5.1 Introduction

Software development, today more than ever, is a community-of-practice activity. Developers often work on multiple projects, hosted on large-scale software repository platforms, such as Github and *BitBucket*. They access and contribute information to open question-answering web sites, such as *Java Forum*, *Yahoo! Answers* and Stack Overflow [2]. Through the developers' participation on these code-sharing and question-answering platforms, rich evidence of their software development expertise is collected. Understanding the developers' expertise is relevant to many software-engineering activities, including "onboarding" of new project members so that their expertise is best utilized in the new context, forming new teams that have the necessary expertise to take on new projects, and bug triaging and assignment to the person that is best skilled to fix it.

In this study we focus on the bug-triaging-and-assignment task, which has already received substantial attention by the software-engineering community (Anvik et al. 2006)(Čubranić and Murphy 2004)(Jeong et al. 2009)(Linares-Vásquez et al. 2012)(Matter et al. 2009)(Nguyen et al. 2014). The typical formulation of **bug triaging** problem aims at ranking a number of developers that could potentially fix a given bug report. Most solutions to date have considered developers' expertise, using their past development and bug-resolving contribution as evidence. In contrast, we describe and report on the effectiveness of a bug-assignment method that uses expertise networks extracted from social software-development platforms.

At a high level, our work makes two novel contributions to the bug-triaging research. First, we demonstrate that as a software focused Q&A web site, Stack Overflow contains valuable information about the expertise of the participating developers, which may be exploited to support bug triaging. Second, we comparatively investigate a family of methods for analyzing Stack Overflow posts to precisely understand how to improve state-of-the-art bug-triaging methods.

The rest of the chapter is as follows. Section 5.2 sets the background context for our work. Section 5.3 describes in detail our new bug-assignment method, ranking the expertise of developers based on a new metric relying on Stack Overflow. Section 5.4 reports on the evaluation of our method. Finally, Section 5.6 concludes with a summary of the take-home lessons of this work.

[2] `http://www.coderanch.com/forums`, `http://answers.yahoo.com`, and `http://stackoverflow.com/`

5.2 Literature Review

There are two categories of previous research relevant to this body of work: (a) expertise identification and recommendation; and (b) bug triaging.

Expertise Identification and Recommendation Venkaratamani *et al.* (Venkataramani et al. 2013) described a system for recommending specific questions to Stack Overflow members qualified to answer them. The system infers the developers' expertise based on the names of the classes and methods to which the developers have contributed. Similarly, Fritz *et al.* (Fritz et al. 2010) developed the "Degree of Knowledge" (DOK) metric to determine the level of a developer's knowledge regarding a code element (class, method or field), based on the developer's contribution to the development of this element. Mockus and Herbsleb (Mockus and Herbsleb 2002) developed the Expertise Browser (EB), a tool that identifies the developers' expertise from their code and documentation, considering system commits and changes to classes, sub-systems, packages, etc.

Zhang *et al.* (J. Zhang et al. 2007) described a method for constructing a "Community Expertise Network" (CEN) from the post-reply relations of Java Forum users. They then ranked the users' expertise using the PageRank (Brin and Page 1998) and HITS (Hyperlink-Induced Topic Search) (Kleinberg 1999) algorithms on this network.

Bug Triaging Previous research in bug-triaging has produced a number of different techniques for selecting the (list of k) most capable developer(s) to resolve a given bug report. Typically the first developer in the list is selected as the bug assignee but, if this developer is unavailable or somehow unsuitable to work on the bug report, the other developers in the recommendations list may be tasked with the bug. Given this problem formulation, most researchers evaluate their methods by reporting *top-k* "accuracy" (Anvik 2006; Bhattacharya et al. 2012; Čubranić and Murphy 2004; Jeong et al. 2009; Lamkanfi et al. 2011; Lin et al. 2009; Shokripour et al. 2012; Tamrawi et al. 2011a; Tamrawi et al. 2011b) (hit ratio in the *top-k* recommended list) or precision-and-recall (Anvik 2006; Anvik et al. 2006; Anvik and Murphy 2011; Canfora and Cerulo 2006; Matter et al. 2009; Shokripour et al. 2013) (precision is the percentage of the suggested developers who were actual bug fixers and recall is the percentage of bug fixers who were actually suggested).

Machine Learning (ML) approaches: Čubranić and Murphy(Čubranić and Murphy 2004) used a Naive Bayes classifier to assign each bug report (a "text document" consisting

of the bug summary and description) to a developer (seen as the "class"). Their classifier was able to predict the bug assignee with a *top-1* accuracy of up to 30%.

Next, Anvik *et al.* (Anvik et al. 2006) proposed a Support Vector Machine (SVM) method as a more effective text classifier for this problem, reporting up to 57%, 64% and 18% *top-3* accuracy. Additionally considering the bug-report severity and priority (Anvik and Murphy 2011) resulted in 75%, 70%, 84%, 98% and 98% *top-5* accuracy. Note that the last two high-accuracy results are were obtained in very small projects, with 6 and 11 developers respectively. A subsequent method, taking also into account information about the components linked to bugs and the list of active developers resulted in 64% and 86% accuracies in two projects (Anvik 2006).

Lin *et al.* (Lin et al. 2009) used SVM and C4.5 classifiers, considering the bug-report textual data (title and description) as well as the bug type, class, priority, submitter and the module IDs, and obtained up to 77% accuracy.

Considering severity and component of the bug reports in addition to the textual descriptions, Lamkanfi *et al.* (Lamkanfi et al. 2011) compared the effectiveness of four ML approaches, Naive Bayes, Multinomial Naive Bayes, 1NN and SVM in predicting the ground-truth assignee. They reported Multinomial Naive Bayes as the most accurate method with 79% accuracy.

Naguib *et al.* (Naguib et al. 2013) used LDA to assign the bug reports to topics. Then, mining the activity profiles of the developers in a bug-tracking repository, they associate topics to developers. Finally, they suggest the developers with the most topics matching with the bug-report topics. They obtained up to 75% *top-5* accuracy.

Information Retrieval (IR) approaches: Canfora and Cerulo (Canfora and Cerulo 2006) consider each developer as a document by aggregating the textual descriptions of the change requests that the developer has addressed. Given a new bug report, the textual description of the new request is used as a query to the document repository to retrieve the candidate developer. This method achieved 62% and 85% accuracy in two projects.

Develect, by Matter *et al.* (Matter et al. 2009), employs the Vector Space Model (VSM) and relies on a vocabulary of "technical terms" collected from the developers' source-code commits and the bug-report keywords. The developer's expertise is modeled as a term vector, based on that developer's commit history. Given a new bug report, the closest –according to the cosine distance– developer is identified. This method achieved up to 34% and 71% *top-1* and *top-10* accuracies.

Linares-Vásquez *et al.* (Linares-Vásquez et al. 2012) applied IR-based concept-location techniques (Poshyvanyk and Marcus 2007) to locate the source code files relevant to the text-change request. Source-code authorship information of these files was used to recommend expert developers and they obtained up to 65% precision.

Shokripour *et al.* (Shokripour et al. 2012) proposed an assignee recommender for the bug reports based on information extracted from the developers' source code, comments, previously fixed bugs, and source code change locations. A subsequent study (Shokripour et al. 2013) improved these results using additional data, such as the source-code files, commits and comments of the developers, names of classes, methods, fields and parameters in the source code. The maximum *top-5* accuracy of their approach on three different projects was 62%. They obtained 48% and 48% *top-1* and 60% and 89% *top-5* accuracies on two projects (between 57 and 9 developers respectively).

Other approaches: Tamrawi *et al.* (Tamrawi et al. 2011b) introduced a fuzzy approach that computes a score for each "developer - technical term" based on the technical terms available in previous bug reports and their fixing history by the developers. Considering the new bug report, they calculate a score for each developer as a candidate assignee by combining his/her scores for all the technical terms associated with the bug report in question. This method was shown to achieve between b40% and 75% for *top-1* and *top-5* accuracy over 7 projects.

A number of studies have examined bug reassignments, the reasons that cause them, and ways to reduce them (Baysal et al. 2009) (Zimmermann et al. 2012). To reduce bug reassignments, Jeong *et al.* (Jeong et al. 2009) introduced "tossing graphs" of developers (as nodes) and edges between them, weighed by the number of times the destination developer was assigned a bug originally assigned to the source developer. Then, beginning with the first prediction (developer candidate) in hand, they used this graph to predict the next developer by consulting this graph. They obtained up to 77% *top-5* accuracy.

All the above studies some combination of the bug textual and categorical attributes, the bug code components, and the developers' coding and bug-fixing contributions. Our method is unique in that it uses the developers' Stack Overflow questions and answers, as well as their previous bug assignments, and correlates these contributions to relevant bugs based on the semantic tags they share.

5.3 A Social Bug-Triaging Model

Software developers today contribute to a variety of social platforms, including social software-development platforms, question-and-answering communities, technical blogs, and presentation-sharing web sites. The key intuition of our work is that these contributions constitute evidence of expertise that can be exploited in the context of bug triaging. More specifically, in this study, we analyze the developers' contributions in Stack Overflow for assigning them to Github bug reports. Focusing on the overlap of the two social platforms (Sajedi-Badashian et al. 2014), our approach examines the questions and answers in Stack Overflow that pertain to the terms mentioned in a bug report's title and description. It uses Stack Overflow tags for cross-referencing Github bug reports with Stack Overflow questions and answers. Tags categorize the questions and their corresponding answers in terms of a few well-known technical terms. The community curates these tags to improve their quality: the person asking a question selects the initial tags for the question (out of around 40,000 available but evolving tags) and expert community members, who enjoy a reputation above some threshold, can edit them. Tags are also used as indication of expertise; for example, the person answering a question tagged with *Android* and *Java* is assumed to be knowledgeable in these two domains. Furthermore, the more upVotes this answers collects, the more knowledgeable this answerer is assumed to be.

Figure 5.1 summarizes the elements of interest in a real bug report[3]. Some of the words in the bug-report's title and description are shown as *italic* because they also appear as tags in the Stack Overflow questions reported in Table 5.1, where they are shown in bold.

Bug report title: TooManyOpenFiles might cause data-loss in *ElasticSearch Lucene*
Bug report body: Under certain circumstances a TooManyOpenFiles exception in *Java* thrown as FileNotFoundException might cause *data* loss where entire shards *lucene* indices are deleted. This is mainly caused by Lucene-4870 *https* issues.apache.org *jira* browse LUCENE-4870 - currently all *Elasticsearch* releases are affected by this.
Project title: *elasticSearch*
Project description: *Open Source* Distributed RESTful *Search* Engine
Project language: *Java*

Figure 5.1: An example bug report (selected fields)

Table 5.1 reports partial information about five questions in Stack Overflow and the

[3] `https://github.com/elasticsearch/elasticsearch/issues/2812` 2014-08-20

answers provided by seven developers. Each question is associated with the developer who asked it, the number of upVotes it received, and its thematic tags. The questions are sorted based on the number of their tags that match with the bug-report textual information (in Figure 5.1) and are shown in bold under each question. The more tags the question shares with the bug-report terms, the more relevant it is to the bug report. We will use these tags to characterize the *expertise areas* required to address the bug report in question.

Table 5.1: An example of scoring regarding the bug shown in Figure 5.1 based on the users' activities in Stack Overflow

Question/Answerer	$^{up}_{Votes}$	Bob	Ali	Joe	Mike	Jane	Tom	Ben
Q1/Mike; version control, **open source**	3	46	5	53		28		
Q2/Jane; ajax, php, **data**, **search**, jquery	1	20	16	22	6			
Q3/Mike; **elasticsearch**, php, **java**, **lucene**	21	11	14	29		10		
Q4/Ali; **https**, css, **java**, **jira**, **data**	0	27			0		86	
Q5/Ben; **search**, **lucene**, **https**, **java**, **elasticsearch**	70	1	18	42	-4	14	98	
AnswerNum		5	4	4	3	3	2	0
Z_score		2.24	1.34	2	0.45	1	1.41	-1
A_score		$\frac{(46+1)\cdot1+(20+1)\cdot2+(11+1)\cdot3+(27+1)\cdot4+(1+1)\cdot5}{247}$	$\frac{(5+1)\cdot1+(16+1)\cdot2+(14+1)\cdot3+(18+1)\cdot5}{180}$	$\frac{(53+1)\cdot1+(22+1)\cdot2+(29+1)\cdot3+(42+1)\cdot5}{405}$	$\frac{(6+1)\cdot2+(0+1)\cdot4+(-4+1)\cdot5}{}$= 3	$\frac{(28+1)\cdot1+(10+1)\cdot3+(14+1)\cdot5}{137}$	$\frac{(86+1)\cdot4+(98+1)\cdot5}{843}$	0
Q_score ($\mu = 20$)		0	$20\cdot(\frac{4}{0+1})$ =80	0	$20\cdot(\frac{1}{3+1}+\frac{3}{21+1})$ =7.7	$20\cdot(\frac{2}{1+1})$ =20	0	$20\cdot(\frac{5}{70+1})$ =1.41
SSA_Z_score		15.72	6.20	20.12	-1.44	9.34	29.03	-1.19

5.3.1 Social Metrics of Expertise

Zhang *et al.* (J. Zhang et al. 2007) introduced a family of metrics for measuring expertise in social networks. The simplest one is *AnswerNum*, the number of answers contributed by a user. However, while answering a question is an indication of expertise, asking a question is an indication of lack of expertise. Z_score is a more sophisticated metric that considers both

questions and answers: $Z = (a - q)/\sqrt{(a + q)}$. In this formula, q and a are the numbers of the questions and answers correspondingly posted by user u. If a user asks as many questions as he answers, his Z_score will be close to 0. Developers who answer more questions than they ask have positive Z_scores, and vice versa. The *Z_score* is undefined for users who have not asked nor answered a question. The developers in Table 5.1 are ordered (left to right) in descending *AnswerNum* order.

5.3.2 A Bug-Specific Social Metric of Expertise

The *Z_score* would likely identify the most active question answerers as the preferred bug assignees every time, consistently ignoring all other developers. To prevent this phenomenon, we have chosen to refine the *Z_score* with bug-specific information. As discussed before, we use Stack Overflow tags as a cross-referencing mechanism between Github bug reports and Stack Overflow questions and answers. Developers facing problems with their tasks, use these tags, which are indexed by search engines (Stack Exchange Team n.d.), to search for earlier questions and their answers that could be helpful to them. Tags are generic enough to convey semantic topics and, yet, specific enough to relate to programming concepts and expertise needed to fix Github bugs. As a sanity check against the possibility that tags may drastically limit the relevant information between Github and Stack Overflow, we examined the bug reports in three selected Github projects (out of the 20 projects considered in this study) and found that the textual information of each bug report (including projectLanguage, projectDescription, issueTitle and issueBody) mentions between 2 to 89 Stack Overflow tags (avg=14.9, var=132 and σ=11.5). In effect, the Stack Overflow tags define a common vocabulary for developers to exchange information. This vocabulary has a fundamental advantage over natural languages; all tags are useful and there is no need for stop-word and noise-word removal from the bug-report texts.

Our approach limits the search for potential bug assignees to the Stack Overflow members that have asked questions or provided answers with at least one tag in common with the text of the bug report under examination, b. To that end, we define the following terms.

$$A\_score_{u,b} = \sum_{a \in u's\ answers} (upVa + 1) \cdot (match\_tags_{a,b}) \tag{5.1}$$

$$Q\_score_{u,b} = \mu \cdot \sum_{q \in u's\ questions} \frac{(match\_tags_{q,b})}{(upVq + 1)} \tag{5.2}$$

$$Z\_score_{u} = \frac{(a - q)}{\sqrt{(a + q)}} \tag{5.3}$$

$$SSA\_Z\_score_{u,b} = \frac{(A\_score_{u,b} - Q\_score_{u,b})}{\sqrt{(A\_score_{u,b} + Q\_score_{u,b})}} \tag{5.4}$$

- **match_tags**$_{SO,b}$: all the Stack Overflow tags that appear in the title and description of the bug report; these are, in effect, the Stack Overflow topics that are important for the bug report in hand.
- **match_tags**$_{q,b}$: the shared tags between a question (q) and b.
- **match_tags**$_{a,b}$: the tags that annotate the question of an answer (a) that also appear in b.

Based on the above definitions, we have developed a measure of the expertise of the project developers in the areas defined by the $match\_tags_{SO,b}$ set. As we have discussed above, our inspection of numerous bug reports has established that the textual information of each bug report is usually matched with several tags. As a result, the relevant subsets of q and a for each developer, $match\_tags_{q,b}$ and $match\_tags_{a,b}$, frequently contain more than one elements.

Our expertise metric is specific to a particular bug report, b. It is *subject-aware* in that it considers two sets of tags –$match\_tags_{q,b}$ and $match\_tags_{a,b}$– relevant to the bug under examination. Finally, it is *social* in that it relies on social assessments of the Stack Overflow content, taking into account the numbers of upVotes and downVotes associated with the developer's Stack Overflow questions and answers.

Let us now describe our expertise metric for developer (user)[4] u on bug report b. We define the $A\_score_{u,b}$ (see Equation 5.1) and $Q\_score_{u,b}$ (see Equation 5.2) to replace a and q

[4]We used the notation u (user) instead of d (developer) in this section since it represents a Stack Overflow user, which is also representing a Github developer. This is the same notation that was used in our published paper.

respectively in the original definition of the *Z_score*. At any point in time, for every answer the user has contributed in the past that is relevant to the bug in question (i.e., is associated with a tag that appears in the bug report), the number of $match\_tags_{a,b}$ is multiplied with the number of the answer's upVotes (plus one, for the answer itself). In effect, each answer contributes to the calculation of the user's expertise, taking into account the number of upVotes that the answer has received, which reflects the community's judgement on the answer's quality and usefulness. The sum of these terms make up $A\_score_{u,b}$. Each question is considered as evidence of lack of relevant expertise but this weakness is compensated by promotion of the question by other users (upVotes). To reflect the intuition that the "asker of a naive question is less knowledgeable than asker of a good one", we divide $match\_tags_{a,b}$ by the number of upVotes (plus one for the question itself). This tends to make the value of $Q\_score_{u,b}$ very small relative to $A\_score_{u,b}$, which is why we use the μ *normalization factor* to adjust it. The *social subject-aware Z_score* (*SSA_Z_score*) can then be defined as shown in Equation 5.4. This formula involves the terms relevant to the user's expertise (as $match\_tags_{q,b}$ and $match\_tags_{a,b}$ used in $A\_score_{u,b}$ and $Q\_score_{u,b}$ for different questions and answers). Furthermore, it takes into account the votes of the users to the answers and questions to advance good ones. The $SSA\_Z\_score_{u,b}$ focuses on answers and questions related to the topics relevant to the bug under examination.

Table 5.1 shows different scores for the users. Each cell at the intersection of a question and a developer contains the number of upVotes for the answer posted by that developer to the question. *Tom* has the best $SSA\_Z\_score_{u,b}$: he provided two answers to questions relevant to the bug, which received many upVotes.

Note that our implementation of the above score is aware of the temporal aspect of a developer's expertise. The activity of a developer in Stack Overflow accumulates over time but the estimation of the developer's expertise for a given bug report, reported in time t, is based only on his contributions up to date: the $SSA\_Z\_score_{u,b}$ considers questions and answers of the user u posted in time $t1 < t$.

A Recency-Aware SSA_Z_score

The expertise of the developers shifts over time as they work on different projects with potentially different technologies (Matter et al. 2009). Developers actively working in a particular domain are more appropriate to be assigned to a bug in this domain. This is why Mayter *et al.* consider a decay factor in their model of developers' expertise. Shokripour

et al. (Shokripour et al. 2012) also consider this idea in their bug-assignment method: the older the evidence for a particular expertise is, the less relevant it is for current expertise needs. Anvik *et al*. (Anvik et al. 2006) used filtering approaches to capture the recency of work.

Motivated by the intuition that "more recent evidence of expertise is more relevant", we define the *recency-aware, social, subject-aware* $RA\_SSA\_Z\_score_{u,b}$ as follows.

$$RA\_SSA\_Z\_score_{u,b} = \alpha \cdot (SSA\_Z\_score_{u,b}) + \beta \cdot \Big(\sum_{\substack{i \in \text{previous bugs} \\ \text{assigned to } u}} \frac{1}{1 + \begin{array}{c} \text{number of bugs occurred} \\ \text{between } i \text{ and } b \end{array}} \Big) \quad (5.5)$$

In this formula, α and β are tuning parameters and we explain how we tuned them in Section 5.4.4. Having the $RA\_SSA\_Z\_score_{u,b}$ for all users in the community over a bug report, our algorithm sorts the users and reports the top k as the most capable developers to fix the bug.

5.4 Evaluation

We obtained two Stack Overflow data sets (S. E. C. Stack Exchange 2014)(Stack Exchange, Inc n.d.) (approximately 65GB and 90GB). They consist of several XML files including information of 2,332,403 and 3,080,577 users, their posts, tags, votes, etc. In order to link these users to Github, their emailHash is needed (Sajedi-Badashian et al. 2014)(Vasilescu et al. 2013), which is provided by the older data set. We merged these two data sets to get a large data set including the newer posts with old users.

We used a mySQL dump (The GHTorrent Project n.d.) (with a size of about 21GB) containing information of 4,212,377 Github users and their project memberships. However, this data set did not include the textual information of the bug reports. We obtained this information from a set of MongoDB dumps provided by the same web site (The GHTorrent Project n.d.) (with a size of about 210GB) including information of 2,908,292 users. Again, we also merged the two data sets and obtained a large data set including information about Github users, projects and bug reports.

As our method assigns bugs to developers with a presence in both Github and Stack Overflow, we used identity merging (Sajedi-Badashian et al. 2014)(Vasilescu et al. 2013) to identity the common users in Github and Stack Overflow. The Github data set contains

the e-mails of the users, but Stack Overflow data set includes e-mail hash. So for each Github user, using MD-5 function, we obtained the e-mail hash and compared it with e-mail hashes in Stack Overflow. With this approach, we found 358,472 common users.

5.4.1 Experiment Setup

For each Github project, we first calculated the union of the sets of project members, committers, bug reporters and bug assignees, and we removed from this set all developers without any Stack Overflow activity, to calculate the project's community-members set. Next, we sorted the projects based on the cardinality of their community-member sets and we identified the top 20 projects[5] with the highest number of community members and the highest number of bug assignees.

For the selected 20 projects, the number of community members vary from 28 to 822 (average=127, median=87). Out of 14,172 bug reports in all the selected projects, we examined 7144 bug reports that have been assigned to one of the project's community members. Note that we could not use the rest of bug reports since they were assigned to developers with no Stack Overflow activity. We used bug reports from three of these projects for training and tuning purposes and 17 for final evaluation. For each bug report in each of the 20 chosen projects, we ran our algorithm to recognize the RA_SSA_Z_score$_{u,b}$ score of all project-community members. Then, we ranked the users from the highest score to the lowest.

We report the average *top-k* recommendation accuracies. We compare our results for k=1 and k=5 with several implemented methods, as well as previously published results. We also report our results based on MAP (Mean Average Precision) as a precise, synthesized, rank-based evaluation measure.

5.4.2 Comparison to State of the Art

Direct comparison with earlier methods is not possible since none of the previous studies we reviewed above have made available their bug-assignment algorithm implementation and data sets. To approximate this comparison, we experimented with the `scikit-learn` [6]

[5] rails/rails, scala/scala, adobe/brackets, JuliaLang/julia, mozilla/rust, mozilla-b2g/gaia, angular/angular.js, bundler/bundler, lift/framework, dotcloud/docker, edx/edx-platform, elasticsearch/elasticsearch, fog/fog, html5rocks/www.html5rocks.com, Khan/khan-exercises, saltstack/salt, travis-ci/travis-ci, NServiceBus/NServiceBus, TryGhost/Ghost and yui/yui3

[6] `http://scikit-learn.org/stable/`

implementations of a number of algorithms classifying bugs to developers, which we applied to our own data set. Considering the previous bug reports and the ground-truth assignee for each one, these algorithms use word-based features of the bug reports to predict the most probable developer who would fix the bug.

1NN, 3NN and 5NN In this family of classifier methods, each bug report is considered a point in a multi-dimensional space, each dimension defined by a distinct word. Each developer (class) corresponds to a hyper-plane in this space, consisting of all the bugs closed by the developer. Then, given a new bug report and a corresponding new point in the space, the closest existing point is selected. The class of the selected point (bug report) is the recommendation for the new bug report. This process is called Nearest Neighbor (1NN). In 3NN and 5NN, we look for 3 or 5 nearest points (bug reports) to that point and simply get their average to determine a hyper-plane and its class (developer) as the recommendation. Lamkanfi, *et al.* (Lamkanfi et al. 2011) and Anvik (Anvik and Murphy 2011) used this method for their predictions about bug reports.

Naive Bayes (NB) and Multinomial Naive Bayes (MNB) In this family of algorithms, the developers' features are the words included in the textual elements of the bug reports they have handled before. These features are considered by the learner as a bag of words. Given a new bug report, the classifier returns the classes (developers) with the highest number features in common with the bug. Bhatacharya *et al.* (Bhattacharya et al. 2012), Čubranić and Murphy (Čubranić and Murphy 2004) and Anvik (Anvik et al. 2006) are from those researchers who used this method for bug triaging.

Building on the above method, a group of Naive Bayes classifiers, one per developer, may be constructed to decide the developer to which a given bug report belongs, and to calculate the probability of that being the case. Then, this probability is compared over all the developers to infer the most probable bug fixers. Lamkanfi *et al.* (Lamkanfi et al. 2011) and Anvik (Anvik and Murphy 2011) used this method for bug triaging.

SVM This approach represents bug reports as vectors in a multi-dimensional space –similar to 1NN, 3NN and 5NN. With each word being a dimension, this classifier considers each bug report a point in this multidimensional space. Then, considering all the bug reports that are already assigned to each developer as a *category*, the optimal hyper-planes between these points to separate different categories is inferred. This method also assigns a label (name of a developer) to each category. Then, given a new bug report, it reports the label of its category. Lin *et al.* (Lin et al. 2009), Anvik *et al.* (Anvik et al. 2006) and Bhattacharya *et*

al. (Bhattacharya et al. 2012) used this method for bug triaging.

5.4.3 Implementation

The Java implementation of our approach as well as our data sets (3 training and tuning and 17 final evaluation projects and their bug reports) and output results are available online at `https://github.com/anonymous-user-1/BugTriaging` for consideration or future comparisons.

Regarding the implemented Machine-Learning approaches, given that no open implementations were available for the previous bug-assignment methods reported in the literature, we made fair effort toward the best implementation of the competitor algorithms. We processed bug reports' title and body words with TFIDF, producing TFIDF word vectors. In order to make the process competitive enough to our approach, we made the process online; train them on first n-1 bug reports and then test on the n^{th}. Then train on first n bug reports and test on $n+1^{th}$ and so on.

We used the followings parameters for `scikit-learn` machine learners. For KNN, we chose k as the parameter (1, 3 or 5), weights='uniform', algorithm='auto', leaf_size=30, p=2, metric='minkowski' and metric_params=None. For Multinomial Naive Bayes, we used Laplace smoothing priors ($\alpha = 1.0$) fit to prior distribution using OneVsRestClassifier classifier strategy. Similarly for Naive Bayes, but it uses multiclass classification. For SVM, we used Support Vector Classification (SVC) class. We chose RBF kernel type, used shrinking heuristic, with gamma kernel coefficient $1/n$ for n features, error penalty=1 and probability=true. More details as well as the the Python implementation of the mentioned approaches are available online at `https://github.com/anonymous-user-1/ML-bug-triager-scikit/blob/master/dumpbayes.py`.

5.4.4 Performance of Variant Social Metrics of Expertise

In Section 5.3 we incrementally developed our *Triage_score* starting with the simple social measures of expertise a and q. To gain an insight on how each aspect of this measure contributes to the bug-assignment effectiveness, we applied several intermediate variants of the metric, representing different intuitions in its evolutionary construction process, to three test projects with 490 bug reports in total, randomly selected from the 20 projects of our study.

The performance of the simplest measure, i.e., the number of answers, *AnswerNum* (J. Zhang et al. 2007), tagged with at least one of those $match\_tags_{SO,b}$ is shown in Table 5.2. The triaging accuracy is poor and does not recommend this naive measure for the bug-assignment task.

The original Z_score (J. Zhang et al. 2007), which considers answers as indication of expertise and questions as indication of lack of expertise, does not perform much better. The problem was that the Z_score metric measures general expertise rather than expertise specific to the bug under examination, and, as a result, it is inadequate to compete with the approaches reported in the literature.

Next we evaluated the subject-aware Z_score, *SA_Z_score*, which measures expertise of the developers in $match\_tags_{SO,b}$, without considering upVotes. This score is in effect equivalent to *SSA_Z_score*, but with μ=1 and without considering upVotes. μ was the *normalization factor* which we used to balance the values of $Q\_score_{u,b}$ with $A\_score_{u,b}$ when it was divided by "1+number of upVotes of the question". In other words, we set $\mu = 1$ for *SA_Z_score* because it does not consider upVotes. Again, a small improvement was observed in the performance, evidence that, not surprisingly, awareness of the bug under examination is useful in selecting the right bug assignee. Still this score is not competitive with the literature results.

Our next step was to consider the community's curation of the questions and answers. Instead of uniformly considering all Stack Overflow answers of a developer as evidence of expertise and all questions as evidence of lack of expertise, we evaluated whether weighing "good" answers and questions more than "bad" ones would make a difference. The Stack Overflow users' upVotes are evidence for the quality of the questions and answers and the *social subject-aware Z_score* (*SSA_Z_score*) was designed to take them into account, as well as being aware of the bug context. This metric involves the μ *normalization factor* that determines the importance of considering "asking" as "lack of expertise" with respect to answers. It can be assigned a static value, or, it may be tuned for different projects. For all projects, we set it to *"1+Harmonic Mean of upVotes of all related questions"* (all questions containing at least one $match\_tags_{SO,b}$) which has slightly better performance. The tuning results are shown in Table 5.2. Note that for the example of Table 5.1, we have μ=20, obtained simply based on the average of upVotes of the questions mentioned in the first column.

The final improvement leading to our triage score was to make it sensitive to the recency

Table 5.2: Accuracy results for preliminary approaches and tuning

Method		Top-1	Top-5	MAP
AnswerNum		3.40	21.00	0.1384
Z_score		3.49	21.05	0.1453
SA_Z_score (μ=1, upVotes=0)		9.12	23.59	0.1801
SSA_Z_score	μ=1	**12.33**	56.97	0.3216
	μ=10	12.06	52.68	0.3153
	μ=20	11.79	50.67	0.3128
	μ=1+avg(upVotes)	12.06	53.61	0.3166
	μ=1+avg(upVotes)^2	11.66	53.73	0.3130
	μ=1+HM(upVotes)	**12.33**	**58.45**	**0.3223**
recency-aware SSA_Z_score	α=0.001	42.65	88.37	0.618
	α=0.01	**43.06**	**88.57**	**0.621**
	α=0.1	41.84	86.33	0.609
	α=1	39.59	77.96	0.565
	α=10	38.98	77.14	0.559

of the relevant Stack Overflow activity. The key intuition here is that "the fixing activity has locality" meaning that "the recent fixing developers are likely to fix bug reports in the near future" (Tamrawi et al. 2011b). Inspired by this idea, we considered the recency of the developers" activities, highlighting recent ones more than past ones. As we anticipated, the results improved further.

Finally, we examined the impact of the various parameters of our metrics to the bug-triaging performance. For the purpose of tuning and calibrating our method, we needed to determine the values for α and β in the $RA\_SSA\_Z\_score_{u,b}$ (Equation 5.5). We set the value of β to 1 in order to reduce the variables to one. Then, changed α and measured the accuracy and MAP on three test projects. The best results obtained with α=0.01. This is because of very large numbers attained for *Social_Z_score* (i.e., number of upVotes multiplied by number of tags, summed over all answers of each user). Later in this section, we apply the parameter values (μ, α and β) obtained from the three projects into the remaining 17 projects in our final evaluation.

5.4.5 Performance of the RA_SSA_Z_score$_{u,b}$

As the final evaluation, we ran our algorithm over 17 projects (holding out the three projects used for tuning) including 6654 bug reports and sorted the recommended developers for each bug report. We measured the average *top-k* accuracies as well as MAP. The average *top-k*

accuracies of our approach for k from 1 to 5 are 45.17%, 66.41%, 77.50%, 84.79% and 89.43% respectively. We also obtained the MAP as 0.633, which is very strong and shows that the harmonic mean of the ground-truth assignee is 1.58 over all the bug reports.

We also implemented the other approaches discussed in Section 5.4.2. We ran those experiments to compare the results of our method with other approaches on the same data set. The results for average *top-1* and *top-5* accuracies as well as MAP are shown in Table 5.3.

Table 5.3: Accuracy results for different simulated approaches compared with ours

Evaluation Measure \ Method	1NN	3NN	5NN	Naive Bayes	Multinomial Naive Bayes	SVM	Our approach
Top 1 Accuracy (%)	43.09	**46.48**	45.60	43.77	42.75	45.46	45.17
Top 5 Accuracy (%)	70.46	75.63	75.00	78.98	75.97	81.82	**89.43**
MAP	0.575	0.610	0.596	0.609	0.606	0.617	**0.633**

Note that all the values reported in Table 5.3 are averages over all the 17 projects examined. We also examined the detailed results for each project and found them close to the mean (var=60.97 and σ=7.81 for *top-5* accuracies). Our results demonstrate that our $RA\_SSA\_Z\_score_{u,b}$, relying on evidence of developers' expertise from their Stack Overflow activities, is very effective in selecting the right assignee for the right bug, much more so than all competing machine-learning algorithms relying exclusively on Github data. In the next section, we analyze these results and compare the details with the other methods.

5.5 Analysis

First, we compare our approach against implemented machine-learning methods. The results in Table 5.3 show that our method outperforms all of the other machine learning methods in terms of *top-5* accuracy and MAP. 3NN, 5NN and SVM do well for top-1 accuracy, slightly better than our approach. Our average *top-5* accuracy is between 8 to 19 percent better than other approaches. The MAP value of our approach, 0.633, corresponds to the harmonic mean 1.58 for the rank of the ground-truth assignee (implying that the ground-truth assignee frequently appeared in the rank-1 and rank-2 positions in the results). MAP varies from 0.575 (for 1NN) to 0.617 (for SVM as the best approach after ours). Comparing the different algorithms on the same data set demonstrates the usefulness of our method. The improved MAP and accuracy of our approach over these other methods shows that our

approach is trustworthy and capable of precise assignee recommendation.

Let us now compare the accuracy of our approach against the accuracy reported in previous published contributions. Due to differences in the experimental design and collected metrics of the various studies, it is impossible to have an exact and fair comparison. Some of these earlier methods reported the maximum accuracy over different projects instead of the average accuracy. Also they differ in reported values for k in *top-k* accuracies, with *top-1* and especially *top-5* being the most frequently used. As one of the best obtained accuracies in the previous studies, Shokripour *et al*. obtained 48% *top-1* and 60% and 89% *top-5* accuracies on two projects (between 57 and 9 developers respectively). Our *top-5* accuracy outperforms theirs, but their approach performs 3% better on *top-1*. Note that their best results were obtained in a project with only 9 candidate developers (our projects included between 28 and 822 developers). Also note that their approach was tested only on 80 and 85 bug reports, as opposed to our 7144 bug reports. In fact, some of the features and meta-data that are required for their method (e.g., product and component of the bug reports) are difficult to obtain (Lamkanfi et al. 2011), which makes this study challenging to replicate.

To summarize our comparison findings, it is important to mention the following. Our evaluation of our metric is the most thorough reported in the literature (with 20 projects and 7144 bug reports). Our metric highly outperforms all previously reported methods in terms of average *top-5* accuracy, and most of them in terms of average *top-1* accuracy. More importantly, our metric exhibits the highest MAP.

Limitations and Threats to Validity The most important concern with respect to the validity of our method is that the common users (between Stack Overflow and Github) who constitute the project community are a small part of the complete set of developers associated with each project. The common users between Stack Overflow and Github represent up to 20% of the total number of users, in each of these networks. There are many users about whom we do not have information, because we could not match their profile in the two networks. However, to mitigate this limitation, unlike most previous studies, we examined our approach on a large number (i.e., 20) of big projects with thousands of users and bug reports which is fairly substantial, limiting threats to external validity. We can even argue that this phenomenon may be an advantage of our approach that focuses on high-quality evidence of developers' expertise established in the actively curated Stack Overflow commu-

nity and ignores developers who do not have such credentials. If our method performs well by accessing parts of the developers' contributions, it should improve when accessing the complete information.

Currently, for privacy reasons, much of the Q&A content at the software social networks is provided anonymously. One could envision however that project managers could request their developers to provide their Stack Overflow IDs. Thus, the step of identifying users common across the two networks through their e-mails should become unnecessary and a larger community of developers, with far more extensive Q&A contributions, will become available to the bug-assignment process.

One concern, is the practice of some developers answering their own questions on Stack Overflow for announcing a commonly encountered issue with some API, library, etc. However, we investigated the questions and answers of members of three (out of the 20) chosen projects and found that only 3% of their answers are answers to one's own questions, and only in around half of these cases the question is up-voted, meaning that the case did not indicate expertise, but lack of expertise (as we assumed).

5.6 Conclusions and Future Work

The fundamental novelty of our work lies in that it is the first bug-assignment method to consider evidence of developers' expertise beyond their contributions to software development, examining instead their contributions to a Q&A platform. Our method takes advantage of the fact that many developers participate in both platforms. Relying on the expertise of the community to recognize good (and bad) questions and answers, our method taps into a rich, and as yet unexploited, social source of expertise information. To consider this information in the context of the software-development task at hand, our method relies on the intersection between Github bug-report text and tags of the Stack Overflow questions and answers. We believe Stack Overflow is a rich source of expertise for software engineering purposes since the privilege of important Stack Overflow contributions like up/downVoting is only available to community members who have established a minimum reputation.

We have thoroughly evaluated our method with 20 popular Github projects, comparing its performance (a) against six traditional machine-learning approaches that have been widely used for bug assignment before, and (b) against the reported accuracies of previous bug-triaging publications. Our approach exploits expertise information found in Stack

Overflow and readily outperforms the competition. We believe that in order to achieve even better performance, a project manager may ask the ID of his developers in the software social networks and identify their full Q&A contributions.

Generalizing beyond Stack Overflow, how helpful it is for bug assignment, and what limitations it suffers, we envision a new research agenda studying the application of third-party expertise networks to bug triaging. The biggest open question is how to generalize this approach to multiple expertise networks. As well as various Q&A networks and code forums, perhaps there are wikis, project documentation, or developer performance histories that could be mined for expertise networks to exploit for bug triage.

In addition to considering multiple social platforms, we also plan to consider tag synonyms: Stack Overflow introduces lists of tag synonyms and suggests the users to use the primary definitions (e.g., "servlets" instead of "webservlet", "authentication" instead of "login"), but does not enforce the practice. In the future, we plan to consider integration of the synonyms in their primary definitions in code and data sets.

Acknowledgments

This work has been partially funded by IBM, the Natural Sciences and Engineering Research Council of Canada (NSERC) and the GRAND NCE.

5.A Appendix: A Preliminary study for Usage of External Beyond-project Sources of Expertise in Isolated Settings

Abstract

Bug triaging and assignment is a time-consuming task in big projects. Most research in this area examines the developers' prior development and bug-fixing activities in order to recognize their areas of expertise and assign to them relevant bug fixes. We propose a novel method that exploits a new source of evidence for the developers' expertise, namely their contributions to Q&A platforms such as Stack Overflow. We evaluated this method in the context of the 20 largest Github projects, considering 7144 bug reports. Our results demonstrate that our method exhibits superior accuracy to other state-of-the-art methods, and that future bug-assignment algorithms should consider exploring other sources of expertise, beyond the project's version-control system and bug tracker.

5.A.1 Introduction and Background

Bug-triaging-and-assignment has received substantial attention by the software-engineering community (Anvik et al. 2006; Čubranić and Murphy 2004; Hossen et al. 2014; Jeong et al. 2009; Kagdi et al. 2012; Linares-Vásquez et al. 2012; Matter et al. 2009; Nguyen et al. 2014; Zanjani et al. 2015). Given a bug report, the goal is to select and rank relevant project developers, who would have the relevant knowledge to fix it. This problem touches on two relevant research areas: (a) expertise identification and recommendation, and (b) bug triaging.

Relevant to *expertise-recommendation*, Venkataramani *et al*. (Venkataramani et al. 2013) described a system for recommending Stack Overflow members qualified to answer specific questions. The system considers the names of the classes and methods to which the developers have contributed to infer their expertise. Similarly, Fritz *et al*. (Fritz et al. 2010) posed the "Degree of Knowledge" (DOK) metric to determine the level of a developer's knowledge regarding a code element –class, method or field– based on the developer's contribution to the development of this element. Mockus and Herbsleb (Mockus and Herbsleb 2002) described "Expertise Browser" (EB), a tool that identifies the developers' expertise about code and documentation, considering system commits and changes to classes, sub-systems, packages, etc. Teyton *et al*. (Teyton et al. 2014) developed XTic, a system taht requires as input a set of skills of interest and provides an automatic process that extracts skills and experience levels from source code repositories. Zhang *et al*. (J. Zhang et al. 2007) described a method for constructing a "Community Expertise Network" (CEN), from the post-reply relations of Java Forum users. Assuming that asking questions is evidence of ignorance and providing answers is evidence of expertise, they defined and demonstrated the usefulnes of the Z-score as an expertise indicator: $Z = (a - q)/\sqrt{(a + q)}$, where q and a are respectively the number of questions asked and answered by a community member.

There has already been substantial research on *bug triaging*, which has produced a number of different techniques to select the (*top-k*, where k is typically 1 and 5) most capable developer(s) to resolve a given bug report. Table 5.4 summarizes some key results of this work.

In relation to this earlier research, the method we propose in this paper is unique in that it uses the developer's expertise, as demonstrated by the developer's contributions to Stack Overflow, as a source of evidence regarding the competence of a developer to fix a bug. We describe our method in Section 5.A.2 and we report on our experimental-evaluation results

Table 5.4: Recent Bug-Triaging Methods

Authors	Basic method / Information used	Effectiveness
(Čubranić and Murphy 2004)	Naive Bayes classification of bug reports (i.e., "text documents") to developers (i.e., "classes"); uses bug summary and description.	up to 30% *top-1*
(Anvik et al. 2006)	Support Vector Machine (SVM) classification of bug reports (i.e., "text documents") to developers (i.e., "classes"); uses bug summary and description.	up to 57%, 64% and 18% *top-3* accuracy
(Tamrawi et al. 2011b)	A fuzzy-set representation of the relations between developers and the bug reports' technical terms; uses bug summary and description.	average 40% and 75% for *top-1* and *top-5* accuracy over 7 projects
(Lamkanfi et al. 2011)	Multinomial Naive Bayes and some other ML approaches; uses bug report summary, description, severity and component.	79% accuracy in predicting severity of bug reports
(Lin et al. 2009)	SVM and C4.5 classifiers; uses bug summary and description, type, class, priority, submitter and the module ID.	up to 77% top-10 accuracy
(Nguyen et al. 2014)	Regression model based on LDA topic modeling; uses bug description.	Just estimated the time to fix for each developer ±2.3 days
(Canfora and Cerulo 2006)	A probabilistic IR method to query the new bug report's text and find the best developer (considered as a document); uses descriptions of the change requests.	62% and 85% accuracy in two projects
(Matter et al. 2009)	Vector Space Model (an IR method); uses source-code commits and the bug-report keywords;	34% and 71% *top-1* and *top-10* accuracies
(Linares-Vásquez et al. 2012)	IR-based concept location techniques; uses text of a change request and source code files.	85% *top-5* accuracy
(Shokripour et al. 2013)	A method based on information extraction; uses bug summary and description, detailed source code info (comments, names of classes, methods, fields, etc.).	48% and 48% *top-1* and 60% and 89% *top-5* accuracies on two projects (57 and 9 developers)
(Jeong et al. 2009)	Introduced "tossing graphs" of developers to reduce bug reassignment; uses bug report title and description	up to 77% *top-5* accuracy

in Section 5.A.3. Reflecting on these results and the corresponding results of earlier studies, we argue for the merits of our method in Section 5.A.3. Finally, in Section 5.A.4, we conclude with the lessons we hope to share with the community and our plans for future work.

5.A.2 A Social Bug-Triaging Method

Motivated by the overlap in the activities of developers in Github and Stack Overflow (Sajedi-Badashian et al. 2014; Vasilescu et al. 2013), in this work, we ask *what if we combine expertise-recommendation based on networks like Stack Overflow with triaging of issues?* and we describe a method that exploits evidence of expertise in the developers' Stack Overflow activity traces, for identifying candidate bug fixers in Github. Let us describe the key

intuition of our method with a simple example: if a developer has answered several questions tagged with the *jquery* keyword, and her answers have received the community's approval with many upVotes, she has a "proven" expertise record in *jquery*; therefore, she should be a likely candidate for fixing bugs whose description includes the *jquery* keyword.

Consider, for example, the activity around five Stack Overflow questions, shown in Table 5.5. The bold tags indicate keywords that also appear in the bug report, which needs to be addressed. The middle section of Table 5.5 reports the simple *AnswerNum* score, namely the total number of questions answered by the developer, and the developer's *Z-score*(J. Zhang et al. 2007) as described above. These two expertise indicators completely ignore the bug at hand, which is why we developed the three additional scores, reported at the bottom of Table 5.5 and described in detail in Section 5.A.2 below. We use Stack Overflow tags for cross-referencing Stack Overflow and Github. As a sanity check for the applicability of tags in Github bug reports, we examined the bug reports in 3 selected projects (out of the 20 projects considered in this study) and found that the textual information of each bug report (including *project language*, *project description*, *issue title* and *issue body*) mentions between 2 to 89 tags (avg=14.9 and σ=11.5). In effect, tags are keywords, curated by the community, that define a common set of vocabularies for developers to exchange information without the need for stop-word and noise-word removal from the bug-report texts.

A Bug-Specific Social Metric of Expertise

Given a bug report, b, the objective is to estimate a developer's expertise and potential ability to fix it. To that end, we define $\mathbf{matched\_tags}_{q,b}$ and $\mathbf{matched\_tags}_{a,b}$, as the set of tags of a specific question (q) and its answers (a) that appear in the textual information of the bug report (b). These metrics are calculated for each pairwise combination of bug reports and questions (and answers) provided by the project developers.

We next define $A_{u,b}$, relative weighted answers, and $Q_{u,b}$, relevant weighted questions, to replace a and q respectively in the original definition of *Z-score*, taking into account the community's assessment of the "quality" of a developer's contributions. The definition of $A_{u,b}$ is shown in Equation 5.6. At any point in time, for every answer a developer has contributed in the past that is relevant to the bug under consideration, the number of $matched\_tags_{a,b}$ is multiplied with the number of the answer's upVotes (plus one, for the answer itself).

Table 5.5: Example of different scores for users

Answerer / Question	Bob	Ali	Taylor	Yakob	Jane	Brian	Harpreet
Q1 by Yakob, 3 upVotes tags: [version control], [**open source**]	46	5	53		28		
Q2 by Jane 1 upVotes tags: [ajax], [**data**], [**search**], [jquery], [php]	20	16	22	6			
Q3 by Yakob, 21 upVotes tags: [**lucene**], [**elasticsearch**], [php], [**java**]	11	14	29		10		
Q4 by Ali, 0 upVotes tags: [**https**], [css], [**java**], [**jira**], [**data**]	27			0		86	
Q5 by Harpreet, 70 upVotes tags: [**java**], [**ajax**], [**https**], [**xml**], [**lucene**]	1	18	42	-4	14	98	
AnswerNum	5	4	4	3	3	2	0
Z-score	2.24	1.34	2	0.45	1	1.41	-1
A	$\frac{(46+1)\cdot 1 + (20+1)\cdot 2 + (11+1)\cdot 3 + (27+1)\cdot 4 + (1+1)\cdot 5}{247} =$	$\frac{(5+1)\cdot 1 + (16+1)\cdot 2 + (14+1)\cdot 3 + (18+1)\cdot 5}{180} =$	$\frac{(53+1)\cdot 1 + (22+1)\cdot 2 + (29+1)\cdot 3 + (42+1)\cdot 5}{405} =$	$\frac{(6+1)\cdot 2 + (0+1)\cdot 4 + (-4+1)\cdot 5}{3} =$	$\frac{(28+1)\cdot 1 + (10+1)\cdot 3 + (14+1)\cdot 5}{137} =$	$\frac{(86+1)\cdot 4 + (98+1)\cdot 5}{843} =$	0
Q ($\mu = 20$)	0	$20\cdot(\frac{4}{0+1}) = 80$	0	$20\cdot(\frac{1}{3+1} + \frac{3}{21+1}) = 7.7$	$20\cdot(\frac{2}{1+1}) = 20$	0	$20\cdot(\frac{5}{70+1}) = 1.41$
SSA_Z-score	15.72	6.20	20.12	-1.44	9.34	29.03	-1.19

$$A_{u,b} = \sum_{\substack{a \in answers \\ posted\ by\ u}} (upVotes_a + 1) \cdot (matched\_tags_{a,b}) \tag{5.6}$$

The $Q_{u,b}$ is calculated as shown in Equation 5.7. In principle, questions are considered as evidence of lack of expertise (J. Zhang et al. 2007). However, to mitigate the adverse effects of asking "good" questions, we divide $matched\_tags_{a,b}$ by the number of upVotes (plus one for the question itself). This tends to make the value of $Q_{u,b}$ very small, in comparison to $A_{u,b}$, which is why we use the μ *normalization factor* to adjust it.

$$Q_{u,b} = \mu \cdot \sum_{\substack{q \in questions \\ posted\ by\ u}} \frac{(matched\_tags_{q,b})}{(upVotes_q + 1)} \tag{5.7}$$

The *Social Subject-Aware Z-score* (*SSA_Z-score*) can then be defined, as shown in Equa-

tion 5.8.

$$SSA\_Z\text{-}score_{u,b} = \frac{(A_{u,b} - Q_{u,b})}{\sqrt{(A_{u,b} + Q_{u,b})}} \tag{5.8}$$

Note that, in order to capture a temporally-aware measure of a developer's expertise, this formula only involves questions and answers posted before the time when the bug was reported (i.e., $t_q < t_b$ and $t_a < t_b$).

A Recency-Sensitive SSA_Z-score

Developers' expertise shifts over time as they work on different projects with potentially different technologies. This is why, many related expertise-modeling methdologies (Anvik et al. 2006; Matter et al. 2009; Shokripour et al. 2013) include a decay factor for older evidence of expertise and weigh it less than more recent one. To capture this intuition, that "recent evidence of expertise is more valuable", we defined *Recency_of_activity* as shown in Equation 5.9 below.

$$Recency\_of\_activity_{u,b} = \sum_{\substack{i \in \\ bugs\ of\ u}} \frac{1}{1 + \ |\{d | d \in bugs\ of\ u \wedge t_d > t_i \ \wedge \ t_d < t_b\ \}|} \tag{5.9}$$

Note that t is the time that the bug report was submitted, and the denominator counts the number of bug reports that occurred between i and b. The intuition here is that bug-fixing exhibits locality, namely that "the developers that have been fixing bugs recently are likely to fix bug reports in the near future" (Tamrawi et al. 2011b). We combine the two last metrics to assign each user a new score regarding the bug currently being triaged:

$$Triage\_score_{u,b} = \alpha \cdot (SSA\_Z\text{-}score_{u,b}) + \beta \cdot (Recency\_of\_activity_{u,b}) \tag{5.10}$$

In this formula, α and β are parameters, which are tuned following a systematic process explained in Section 5.A.3. Having the *Triage_score* for all users in the community over a bug report, our bug-triaging algorithm sorts the users and reports the top k developers to fix the bug.

5.A.3 Evaluation

We obtained two Stack Overflow data sets (Stack Exchange, Inc n.d.; S. E. C. Stack Exchange 2014) (approximately 65GB and 90GB). They consist of several XML files including

information about 2,332,403 and 3,080,577 users, their posts, tags, votes, etc.). In order to link these users to Github, their *email hash* is needed (Sajedi-Badashian et al. 2014), which is provided by the older data set. We merged these two data sets to get a large data set including users of old data set with newer posts.

We used the GHTorrent mySQL dump (Gousios 2013) (with a size of about 21GB) containing information about 4,212,377 Github users and their project memberships. However, this data set did not include the textual information of the bug reports. We obtained this information from a set of MongoDB dumps from the GHTorrent site (210GB) including information of 2,908,292 users. Again, we merged the two data sets and obtained a large data set including information about Github users, projects and bug reports. Both the Github and Stack Overflow data sets include information of the users and their activities from 2008 to 2014. As our method assigns bugs to developers with a presence in both Github and Stack Overflow, we encoded users's emails in Github with MD-5 function and compared them with every e-mail hash available in Stack Overflow (Sajedi-Badashian et al. 2014; Vasilescu et al. 2013). With this approach, we found 358,472 common users[7].

Experiment Setup and Implementation

We first extracted the *community members* of each project as the union of the sets of project members, committers, bug reporters, and bug assignees. We then refined this community to include only developers who had posted some questions or answers on Stack Overflow. Next, we identified the top 20 ranked projects based on the number of their community members[7].

For the selected 20 projects, the number of community members vary from 28 to 822 (average=127, σ=169, median=87). Out of 14,172 bug reports in all the selected projects, we examined 7144 bug reports that have been assigned to one of the community members in the related project. Note that we could not use the rest of bug reports since they were assigned to developers with no Stack Overflow activity. However, if this approach is applied in the workplace, alternative networks should be tested and used. We used bug reports from three of these projects for training and tuning purposes and 17 for final evaluation. For each bug report in each project, we ran our algorithm to compute the expertise score of all project-community members and ranked the users from the highest score to the lowest. We report *top-1* and *top-5* accuracies as well as Mean Average Precision (MAP) as a synthesized,

[7]Our data sets, information of 3+17 projects, Java implementation of our approach and output and tuning results are available online at: `http://github.com/alisajedi/BugTriaging`

rank-based evaluation measure (Wong et al. 2014).

To compare with other state-of-the-art methods, we experimented with the `scikit-learn` [8] implementations of a number of machine-learning algorithms used as the basis for the above research (Anvik et al. 2006; Čubranić and Murphy 2004; Lamkanfi et al. 2011; Lin et al. 2009) which we applied[9] to our own data set: (1) 1NN, 3NN and 5NN; (2) Naive Bayes; (3) Multinomial Naive Bayes; and (4) SVM.

We used the words in the title and description of the bug reports as the TFIDF feature vectors. We developed an online train-and-test method; train them on first n-1 bug reports and then test on the n^{th}. Then recursively train on first n bug reports and test on $n+1^{th}$.

We used the following parameters for `scikit-learn` machine learners. For KNN, we chose k as the parameter (1, 3 or 5), weights='uniform', algorithm='auto', leaf_size=30, p=2, metric='minkowski' and metric_params=None. For Multinomial Naive Bayes, we used Laplace smoothing priors ($\alpha = 1.0$) fit to prior distribution using OneVsRestClassifier classifier strategy. Similarly for Naive Bayes, but it uses multiclass classification. For SVM, we used Support Vector Classification (SVC). We chose RBF kernel, used shrinking heuristic, with gamma kernel coefficient $1/n$ for n features, error penalty=1 and probability=true.

Results

Out of 20 projects, we selected three projects (including 490 bug reports). We then measured the performance metrics of different approaches from *AnswerNum* to original Z-score, to Subject-Aware Z-score (*SA_Z-score*), to *Social Subject-Aware Z-score* (*SSA_Z-score*), to the final recency-aware *SSA_Z-score* (*Triage_score*). In each step, we observed an improvement in accuracy. This validates our effort toward considering Stack Overflow upVotes while being aware of bug content. For the purpose of tuning and calibrating our method, we needed to determine the best values for μ, α and β (Equations 5.7 and 5.10). The best obtained values are as follows: μ (normalization factor) is set to *"Harmonic Mean plus 1"*, β=1 and α=0.01. The reason for small α value can be because of very large numbers attained for *Social_Z-score* (i.e., number of upVotes multiplied by number of tags, summed over all answers of each user). We apply the parameter values (μ, α and β) obtained from the three test projects into the remaining 17 projects in our final evaluation.

As the final evaluation, we ran our algorithm over 17 projects (holding out the three

[8] `http://scikit-learn.org/stable/`

[9] Our Python implementations: `http://github.com/abramhindle/bug-triager-scikit/blob/ali/dumpbayes.py` More explanation of the ML methods are also available at the repository.

projects used for tuning) including 6654 bug reports and sorted the recommended developers for each bug report. The average *top-k* accuracies of our approach for k from 1 to 5 are 45.17%, 66.41%, 77.50%, 84.79% and 89.43% respectively. We also obtained a Mean Average Precision (MAP) of 0.633, which is very strong and shows that the harmonic mean of the ground-truth assignee is 1.58 over all the bug reports. Note that in bug triaging, MAP is equal to Mean Reciprocal Rank (MRR) of the ground-truth assignee over all the recommendations.

We also implemented the other approaches discussed in Section 5.A.3. We ran those experiments to compare the results of our method with other approaches on the same data set. The results for average *top-1* and *top-5* accuracies as well as MAP are shown in Table 5.6.

Table 5.6: Results of different simulated approaches compared with ours

Method / Evaluation Measure	1NN	3NN	5NN	Naive Bayes	Multinomial Naive Bayes	SVM	Our approach
Top 1 Accuracy (%)	43.09	**46.48**	45.60	43.77	42.75	45.46	45.17
Top 5 Accuracy (%)	70.46	75.63	75.00	78.98	75.97	81.82	**89.43**
MAP	0.575	0.610	0.596	0.609	0.606	0.617	**0.633**

The values reported in Table 5.6 are averages over all 17 projects examined. However, we examined the detailed results for each project and found them close to the mean (median=90.19 and σ=7.81 for *top-5* accuracies). Our results demonstrate that our *Triage_score*, relying on evidence of developers' expertise from their Stack Overflow activities, is very effective in selecting the right assignee for the right bug, much more so than all competing machine-learning algorithms relying exclusively on Github data. It is noteworthy that our approach is fast and efficient enough since it avoids the typical text pre-processing of most IR-based methods, such as stemming and indexing. Each bug report was triaged in almost a second, which is fast enough for real-time use.

Analysis

The results in Table 5.6 demonstrate that our method exhibits the best performance, outperforming all other machine-learning methods in terms of *top-5* accuracy and MAP. 3NN, 5NN and SVM do well for top-1 accuracy, slightly better than our approach. Our average *top-5* accuracy is between 8 to 19 percent better than other approaches. The MAP value of our approach —0.633— corresponds to the harmonic mean 1.58 for the rank of the ground-truth assignee (implying that the ground-truth assignee frequently appeared in the rank-1

and rank-2 positions in the results). MAP varies from 0.575 (for 1NN) to 0.617 (for SVM as the best approach after our's). Comparing the different algorithms on the same data set demonstrates in the usefulness of our method.

We also compared our results with previously published results. In short, as one of the best obtained accuracies in the previous studies, Shokripour *et al.* (Shokripour et al. 2013) obtained 48% *top-1* and 60% and 89% *top-5* accuracies on two projects (with 57 and 9 developers respectively). Our *top-5* accuracy outperforms theirs, but their approach performs 3% better on *top-1*. Note that their best results were obtained in a project with only 9 candidate developers (our projects included between 28 and 822 developers). Also note that their approach was tested only on 80 and 85 bug reports, as opposed to our 7144 bug reports, which constitutes strong evidence on the robustness of our approach.

To summarize our comparison findings, it is important to mention the following three key points. Our evaluation of our metric is the most thorough reported in the literature (with 20 projects and 7144 bug reports). Our metric highly outperforms all previously reported methods in terms of average *top-5* accuracy, and most of them in terms of average *top-1* accuracy. More importantly, our metric exhibits the highest MAP/MRR.

Limitations and Threats to Validity

An external validity threat is that the common users (between Stack Overflow and Github) constitute up to 20% of the total number of users in each of these networks. Currently, for privacy reasons, much of the Q&A content at the software social networks is provided anonymously. However, the large number (i.e., thousands) of developers and bug reports on which we tested our approach mitigates this limitation. One could envision that project managers could easily request their developers to provide their IDs in Q&A networks like Stack Overflow, as part of their CV. As a result, more extensive Q&A contributions (or alternative sources of information) will become available to the bug-assignment process.

Another concern is how to treat the phenomenon of developers answering their own questions to announce a commonly encountered issue with some API, library, etc. However, we investigated the questions and answers of members of three projects out of 20 and found that only 3% of their answers are answers to one's own question, and only in around half of these cases, the question is upVoted, meaning that the case did not indicate expertise, but lack of expertise (as we assumed).

5.A.4 Conclusions and Future Work

In this paper, we described a method that effectively utilizes the expertise networks, such as Stack Overflow contributions of developers and their previous bug-assignment history to decide the best candidate developer for fixing a bug. We have thoroughly evaluated our method with 20 popular Github projects, comparing its performance (a) against six traditional machine-learning approaches that have been widely used for bug assignment before, and (b) against the reported accuracies of previous bug-triaging publications. Our approach outperforms the competition.

The fundamental novelty of our work is that it takes advantage of contributions of the developers in software Q&A networks as a rich, unexploited socio-technical source of expertise information, beyond their code. This leads us to a more interesting insight: applying various **third-party expertise networks** in bug triaging envisions new horizons for software maintenance community. In fact, the socio-technical information available on the web is a great source of expertise. While some developers contribute in Stack Overflow, many others may prefer Java Forum, Ask Ubuntu, Experts Exchange, Code Project, Web developer, SUN Forums, MSDN Forums and so on. As part of their development process, developers may provide their IDs in their desired software social platforms, to better inform our method regarding their expertise and thus improve the triaging process.

In the future, we plan to handle tag synonyms. Different synonym tags can be integrated in their primary definition addressing in bug reports and Q&A contents. Finally, capturing level of similarity of the keywords in bug reports with tags (e.g., "xml-parser", "xml parser", "xmlparsing" and "xml parsing" compared to tag "xml-parsing") can be a useful extension. Curation, noise reduction, and tag recommender approaches (Shaowei Wang et al. 2014) may also be useful in this case.

Acknowledgments

This work has been partially funded by IBM, the Natural Sciences and Engineering Research Council of Canada (NSERC) and the GRAND NCE.

5.B Appendix: Can External Beyond-project Sources of Expertise be Useful in General Settings?

We showed in the main part of this chapter that the external sources of expertise are useful in isolated settings (in which any developer with no external contribution is eliminated). This is useful when internal sources are not available –e.g., in new projects.

Here, we investigate the case in more generic settings, without removing any developers or bugs (with or without internal evidence of expertise). Since our *Multisource* approach (Chapter 4) is capable of combining various sources of expertise and its data set contains both internal and external sources of expertise of developers, we build up this section based on the *Multisource* approach.

We consider the usage of two external sources of information as evidence of expertise of developers; evidence from other projects and other social networks. Due to limitations in finding shared users between our projects and sub-projects, or other networks (as we mentioned before in Section 4.4), we consider these external evidence for assigning bugs in our two biggest projects, "Angular.js" and "Rails". Starting from the best configuration we obtained using the within-project sources, each time we add an extra type of external beyond-project information and see if we can improve the overall accuracy. Then we evaluate the usefulness of these external sources of expertise.

5.B.1 Information From Project Families

While specific expertise information –like comments and commits– inside a project is useful in determining the proper bug fixers, the similar information obtained from sub-projects can also be of use in the main project.

To understand the extent to which information from sub-projects can be useful for the main projects ("Angular.js" and "Rails"), we consider the same types of evidence we found useful in Section 4.5. According to the statistics we showed in Table 5.7, there are only 184 and 472 users respectively in Angular.js and Rails who are also in their sub-projects as well. This is a small portion of the users (7.7% and 11.6% respectively). The rest of the users have no multi-project evidence.

We used Equation 4.1 for evaluating appropriateness of a developer for a bug, and considered the same weights we obtained in Table 4.3 for $context(sd_j)$ coefficients. Then we repeated the experiment with additional data (bug comments, commits, pull requests and

Table 5.7: Statistics of the shared users with sub-projects or Stack Overflow

	Angular.js		Rails	
	#	Ratio to all developers	#	Ratio to all developers
Total # of developers	2,386	100%	4,079	100%
# of developers shared with at least one sub-project	184	7.7%	472	11.6%
# of developers shared with Stack Overflow	136	5.7%	265	6.5%

commit comments[10]). We added *one* piece of *additional source* at a time. The highest improvement in final MAP was regarding bug comments but was trivial (less than 0.1%) and hence we do not show them here separately. We explain this as follows;

The addition of evidence from sub-projects are not that effective when within-project evidence are available; this can be due to locality of project expertise. In fact, the local information is much more fruitful than the project families. In a sense, each piece of data that is added, can bring some redundant data or false positives. Once rich local data is available (like in our experiment), addition of external data will not help much. The other reason is that finding external evidence for all the users in a project is very difficult (or impossible). In our experiment, the number of users with multi-project evidence is a small fraction of the all users in the two projects ($\sim$10%).

But the external beyond-project data can be useful when there is no within-project data available (e.g., starting phase of a new project). To investigate this, we check the effectiveness of evidence from sub-projects, without considering within-project evidence. In other words, we assign bugs to developers in two main projects, by considering the evidence of expertise just from their sub-projects. We do not do any filtering to removal of any users (e.g., non-shared users between the main project and their sub-projects). So we have evidence for 7.7% and 11.6% of the users respectively in *Angular.js* and *Rails*. For the rest of the users, we consider no activity, which means no evidence of expertise.

The results of this experiment are shown in Table 5.8. Surprisingly, the overall MAP is 17.43% which is higher than expected. Note that we are using evidence from sub-projects to assign bugs in the main project, and these evidence are related to only around 10% of the users (there is no evidence for the rest of the users). Having more shared users –or more

[10]Note that we cannot use the references to developers' login names from resources of project families. A reference to developer *d* (shown as "*@d*") for bug #n in "Angular.js" is originated from somewhere in the description or comments of that bug in "Angular.js", not other projects.

evidence from those users– can lead to better recommendations. Comparing the amount of evidence of expertise in "Angular.js" against "Rails" confirms this claim; as mentioned before in the last two rows of Table 4.2, the evidence of sub-projects of *Angular.js* is several times more than the evidence of *Rails*. This can interpret the higher accuracy in *Angular.js*.

Table 5.8: Results of using external sources of expertise

	Angular.js		Rails		total	
	# of assignments	MAP	# of assignments	MAP	# of assignments	MAP
Regular (the best obtained result with all within-project evidence)	9,658	60.92	11,305	55.21	20,963	57.84
Just using evidence from project families		19.39		15.76		17.43
Just using evidence from Stack Overflow		10.85		2.64		6.42

The total *top-1*, *top-5* and *top-10* accuracies (for brevity, not shown in Table 5.8) are respectively 6.8%, 32.7% and 45.9% which show that many of the recommendations are in the top of the list. Note that there are ∼2.3k and ∼4k developers respectively in Angular.js and Rails and it is very tough to identify the ground-truth assignee out of those high number of users. So, the sources of expertise in project families are capable of producing fair recommendations, without considering internal evidence of expertise. Although this (using only evidence from external sources) cannot compete against state-of-the-art methods, we argue that those external evidence can be useful in special situations. For example, in early stages of the projects, when there is not enough evidence of expertise available, any additional source is very helpful. Another example is regarding a new developer to an existing project (i.e., we can obtain the evidence of expertise for those specific developers from their contributions in other projects). Needless to say, more shared users, or more evidence of expertise of those shared users between the project families and main project will establish more accurate recommendations.

5.B.2 Information from Other Technical Networks

Next, we consider usefulness of another external source, Stack Overflow, for bug-assignment in software projects. In the main contents of this chapter, we examined usage of Stack Overflow in an "isolated settings" for bug-assignment; we showed that "filtering the developers to the shared users between Stack Overflow and Github" and using our previous method (Sajedi-Badashian et al. 2016) can assign bugs to developers with high accuracy. In

the current study, since our *Multisource* scoring function is capable of considering multiple sources of expertise, we extend our experiments to examine the level of suitability of Stack Overflow posts in bug-assignment, without eliminating non-shared users.

To understand the extent to which the expertise from Stack Overflow is useful, we considered Stack Overflow posts of developers of Angular.js and Rails. According to the statistics we showed in Table 5.7, there are only 136 and 265 users respectively in Angular.js and Rails, who are also in Stack Overflow. This is a small portion of the users (5.7% and 6.5% respectively). The rest of the users have no Stack Overflow evidence.

Again, we use Equation 4.1 for considering appropriateness of a developer for a bug. In addition to the evidence used in our main experiment, we considered Stack Overflow answers as an additional evidence of expertise –regarding the areas mentioned in the tags of the related question. We tunned the coefficient of Stack Overflow answers with different values. Similar to the multi-project experiment, we found that this external source of expertise could not improve the results of *Multisource* (for brevity, we did not show those detailed results here). The reason is similar to the multi-project experiment; once there are local information about the project contributors, the additional external sources are not fruitful. Also note that the number of users with Stack Overflow evidence is a very small fraction of all users ($\sim$ 6%). Adding the questions of Stack Overflow as an additional source of expertise even decreases the accuracy. This is consistent with our previous findings about Stack Overflow questions being evidence of lack of expertise (Sajedi-Badashian et al. 2016), unlike Stack Overflow answers.

In the next step, like previous experiment, we limit the sources of expertise to only the external information (i.e., Stack Overflow evidence). In other words, we assign bugs to the developers in the same two projects of the previous step –Angular.js and Rails– by considering the evidence of expertise just from Stack Overflow, in absence of any evidence from these two projects. Like previous experiment, we do not filter or remove the users (e.g., non-shared users between our two projects and Stack Overflow are not removed but assumed to have no evidence of expertise).

The results are shown in Table 5.8. The overall MAP is 6.42%. Again, *Angular.js* shows much better performance than *Rails*. The results are even lower than the multi-project experiment. We argue that this low accuracy is related to lack of external information. The number of shared developers between Stack Overflow and our projects are 136 and 265 users for *Angular.js* and *Rails* (5.7% and 6.5%) respectively. Considering the fact that the

evidence of expertise is coming from around 6% of the developers to assign all the bugs in the two projects, the results are still convincing. Also note that there are ∼2.3k and ∼4k developers respectively in Angular.js and Rails which makes the assignee prediction very hard.

We believe that having more information (i.e., more shared developers with Stack Overflow or more answers by those shared developers) can produce much better recommendations. To elaborate about this argument, we point to the higher MAP in *Angular.js* compared to *Rails* and examine the shared developers between Stack Overflow and these two projects. As shown in Table 5.9, the percentage of shared developers with Stack Overflow are in both projects around 5% of the total users, but the average and median number of Stack Overflow answers posted by *Angular.js* members are 78.5 and 23 respectively, both around 1.5 times the equivalent numbers in *Rails*. This means that the users of *Angular.js* have more Stack Overflow contributions and hence our method gives better recommendations in it. In other words, having more evidence from Stack Overflow can lead to more accurate results.

Table 5.9: Statistics of the answers posted by the developers in Stack Overflow

		Angular.js	Rails
# of developers with at least one answer (%)		120 (5.03%)	239 (5.86%)
# of answers posted by each developer	min	1	1
	max	1,065	2,036
	average	78.5	54.2
	median	23	15

Again, aligned with results of last sub-section, we argue that Stack Overflow contributions of developers, like the other evidence from project families, can be useful in early stages of the projects (as external sources of expertise), when there is not enough local evidence of expertise available in the project.

Connecting these findings with the ones we obtained in the main contents of this chapter, regarding usefulness of Q&A contributions in isolated settings (by considering only shared users with our projects), we argue that using Q&A contributions as an addition to the sources of expertise is useful only if the shared user base is big enough. The companies' internal developer networks are a good resource for these types of evidence.

Chapter 6

Conclusions and Future Works

In this thesis, we have made the following contributions:

1) We comprehensively surveyed the previous research on bug-assignment and reviewed different objectives, methods, metrics, information and dimensions of variability. Then we proposed a framework for evaluation of bug-assignment research, covering three main inspirations. First, MAP is the most reliable metric for evaluation of bug-assignment experiments. It reflects the effectiveness of the bug-assignment method and is least sensitive to the dimensions of variability in different projects. Second, in order to validate the assignee recommendations realistically, the *definition of ground-truth assignee*, which is used as the ground truth in bug-assignment research, should comprehensively include every bug-fix effort by developers. Because of the general lack of consistency around the processes of recording who fixed a bug, it is reasonable to consider that all developers who contributed to a bug being fixed (whether they are mentioned to have closed it or they are mentioned as having fixed it) or have been nominated by a project manager as assignee should be good choices as assignees to this bug. Third, the *developer community*, who are the potential assignees used to validate a bug-assignment approach, should be inclusive of all project members. We demonstrate that filtering the candidate developers may artificially inflate the reported accuracy. From a higher perspective, validating a new bug-assignment approach needs some spirit of equity and fairness. The important aspects of evaluation and reporting in our proposed framework enables replication of the bug-assignment studies and supports reproducible research. This promotes its usage in other research or industrial applications. This work is reflected in Chapter 2.

2) We developed *TTBA*, a new bug-assignment method that uses a fine-grained term-

weighting scheme that enables utilization of "time" and "importance" of the keywords by developers. It features two main aspects. First, it utilizes the Stack Overflow as the thesaurus of technical terms and identifies the specificity and technicality of the keywords from their appearance statistics in Stack Overflow. Second, it highlights the time of usage of the keywords by developers to diminish the effect of old evidence of expertise. *TTBA* is capable of accurately recommending appropriate developers to fix the given bug reports. We showed that it outperforms other bug-assignment approaches. This work is described in Chapter 3.

3) Preserving *TTBA*'s high granularity of the time of usage of technical terms by developers, we further enhanced it to include multiple sources of expertise with different importance. Unlike most previous bug-assignment studies that focused on enhancing methods, our *Multisource TTBA* approach tries to enhance data by engaging a variety of sources of expertise of developers. We showed that using the sources of expertise from diverse sources in open-source projects can propose non-trivial enhancements in automatic bug-fixing accuracy. Using this *Multisource TTBA* approach, we investigated the information value of different pieces of information that are usually found in open-source software repositories. We realized that different types of social and technical contributions of developers can be very useful for recommending proper fixers. Specifically, we found that the text of bug comments, commits and pull requests (in this order) contain useful information about the expertise of their developer. In addition, the links to developers' login names from developers' contributions contain useful information about who might be a good candidate to fix a given bug. Regarding external beyond-project sources, we found that they generally cannot provide enhancement to the within-project sources. However, they can be useful only in specific conditions (like starting phase of new projects) or when the external sources of all the developers are available (like in some proprietary software). The *Multisource TTBA* approach is discussed in Chapter 4 and the investigation of information value of various sources of expertise is performed in Chapters 4 and 5 (respectively for internal and external sources).

4) We made all our data sets available online for further researchers to replicate our studies, or develop and evaluate their bug-assignment methods on (Chapters 2, 3, 4 and 5). The most comprehensive data set[1] we extracted and published is regarding our latest experiments (Chapter 4 and Appendix 5.B) which includes both technical and social contributions of developers in 13 big open-source projects in Github during +5 years. This data set is one of the most comprehensive and recent data sets available for further bug-assignment research.

6.1 Future Works

Extending different parts of current thesis is possible as future directions:

As one of the contributions of this thesis was to investigate the effect of various information on bug-assignment, still richer data can be used regarding two aspects. First, providing more textual elements available in open-source repositories (e.g., information available in other branches or communications of developers in private developer networks), which would be straightforward using our *Multisource* approach. Second, utilization of varied types of information (e.g., meta-data elements) which might need some adjustments in the scoring function and further tunings.

As another future work to this study, the notion of "assignee" can be enhanced to include all the expert developers regarding each given bug. In order to obtain such a list for each bug, the project managers need to review the bug reports and recommend the set of appropriate developers for each one. In a big project with lots of bug reports, this is expensive in terms of time and effort but would help to have a more appropriate evaluation.

The other amendment would be a query expansion method for inferring some new keywords for each bug report, after eliminating the non-Stack Overflow tags. This may be needed since our *TTBA* and *Multisource* methods eliminate other keywords and the remaining keywords might be only a few keywords. In our data sets, only a few percentage of bug reports had this problem and we did not do any enhancement for them. In any case, one might expand the tags to infer new keywords and enhance the prediction. This may enhance the method's final accuracy. In addition, some stop-words removal (e.g., removing *generic* Stack Overflow tags like *this*, *for* or *while*) may also be useful.

The other possible update is regarding tag synonyms. Stack Overflow preserves a list of tag synonyms which show couples of related tags (the main tag and the secondary one). Merging the secondary tags into the main one in all the bug reports can lead to obtaining better results.

Finally, in our *TTBA* and *Multisource* approaches, for each tag, we considered weights based on its appearance in Stack Overflow. Then we used those weights constantly for all the projects. This can be changed in order to obtain a better accuracy. One would envision using more specific weights for keywords obtained from each domain or project. In any case, the main idea of granularity of time of usage of the keywords and capturing the importance of the technical terms from a thesaurus remains the same.

References

Aggarwal, Karan et al. (2017). "Detecting duplicate bug reports with software engineering domain knowledge." In: *Journal of Software: Evolution and Process* 29.3. 30

Ahsan, Syed Nadeem, Javed Ferzund, and Franz Wotawa (2009). "Automatic software bug triage system (bts) based on latent semantic indexing and support vector machine." In: *Software Engineering Advances, 2009. ICSEA'09. Fourth International Conference on*. IEEE, pp. 216–221. 16, 46

Akbarinasaji, Shirin, Bora Caglayan, and Ayse Bener (2017). "Predicting bug-fixing time: A replication study using an open source software project." In: *Journal of Systems and Software*. 14, 57

Alipour, Anahita (2013). "A CONTEXTUAL APPROACH TOWARDS MORE ACCURATE DUPLICATE BUG REPORT DETECTION." MA thesis. Canada: University of Alberta. 29, 30

Aljarah, Ibrahim et al. (2011). "Selecting discriminating terms for bug assignment: a formal analysis." In: *Proceedings of the 7th International Conference on Predictive Models in Software Engineering*. ACM, p. 12. 1, 10, 16, 20, 4

Anjali, Devina Mohan, Neetu Sardana, et al. (2016). "Visheshagya: Time based expertise model for bug report assignment." In: *Contemporary Computing (IC3), 2016 Ninth International Conference on*. IEEE, pp. 1–6. 17, 20, 92, 95

Anvik, John (2006). "Automating bug report assignment." In: *Proceedings of the 28th international conference on Software engineering*. ACM, pp. 937–940. 16, 117, 118

Anvik, John, Lyndon Hiew, and Gail Murphy (2006). "Who should fix this bug?" In: *Proceedings of the 28th international conference on Software engineering*. ACM, pp. 361–370. 16, 28, 33, 37, 46, 68, 92, 116–118, 125, 127, 137, 138, 141, 143

Anvik, John and Gail Murphy (2011). "Reducing the effort of bug report triage: Recommenders for development-oriented decisions." In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 20.3, p. 10. 16, 46, 117, 11

Bachmann, Adrian et al. (2010). "The missing links: bugs and bug-fix commits." In: *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, pp. 97–106. 51, 79

Banerjee, Sean et al. (2016). "Automated triaging of very large bug repositories." In: *Information and Software Technology* 89. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2016.09.006. URL: http://www.sciencedirect.com/science/article/pii/S0950584916301653. 14, 57

Banitaan, Shadi and Mamdouh Alenezi (2013). "Tram: An approach for assigning bug reports using their metadata." In: *Communications and Information Technology (ICCIT), 2013 Third International Conference on*. IEEE, pp. 215–219. 16

Baysal, Olga, Michael W Godfrey, and Robin Cohen (2009). "A bug you like: A framework for automated assignment of bugs." In: *Program Comprehension, 2009. ICPC'09. IEEE 17th International Conference on*. IEEE, pp. 297–298. 1, 10, 16, 119

Baysal, Olga, Reid Holmes, and Michael W Godfrey (2012). "Revisiting bug triage and resolution practices." In: *Proceedings of the First International Workshop on User Evaluation for Software Engineering Researchers*. IEEE Press, pp. 29–30. 14

Beyer, Stefanie and Martin Pinzger (2015). "Synonym suggestion for tags on stack overflow." In: *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*. IEEE Press, pp. 94–103. 80

Bhattacharya, Pamela and Iulian Neamtiu (2010). "Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging." In: *Software Maintenance (ICSM), 2010 IEEE International Conference on*. IEEE, pp. 1–10. 1, 10, 15, 16, 2

Bhattacharya, Pamela, Iulian Neamtiu, and Christian R Shelton (2012). "Automated, highly-accurate, bug assignment using machine learning and tossing graphs." In: *Journal of Systems and Software* 85.10, pp. 2275–2292. 1, 10, 11, 16, 2

Blanco, Roi and Christina Lioma (2012). "Graph-based term weighting for information retrieval." In: *Information retrieval* 15.1, pp. 54–92. 71, 85, 86, 100

Borg, Markus (2014). "Embrace your issues: compassing the software engineering landscape using bug reports." In: *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, pp. 891–894. 16

Bortis, Gerald and André van der Hoek (2013). "Porchlight: A tag-based approach to bug triaging." In: *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, pp. 342–351. 14

Brin, Sergey and Lawrence Page (1998). "The anatomy of a large-scale hypertextual Web search engine." In: *Computer networks and ISDN systems* 30.1, pp. 107–117. 117

Budgen, David and Pearl Brereton (2006). "Performing systematic literature reviews in software engineering." In: *Proceedings of the 28th international conference on Software engineering*. ACM, pp. 1051–1052. URL: `https://dl.acm.org/citation.cfm?id=1134500`. 13

Canfora, Gerardo and Luigi Cerulo (2006). "Supporting change request assignment in open source development." In: *Proceedings of the 2006 ACM symposium on Applied computing*. ACM, pp. 1767–1772. 16, 22, 25, 26,

Carlson, Patrick Eric (2015). "Engaging developers in open source software projects: Harnessing social and technical data mining to improve software development." PhD thesis. Iowa State University. 67

Cavalcanti, Yguaratã et al. (2014a). "Challenges and opportunities for software change request repositories: a systematic mapping study." In: *Journal of Software: Evolution and Process* 26.7, pp. 620–653. 14, 38

Cavalcanti, Yguaratã et al. (2014b). "Combining rule-based and information retrieval techniques to assign software change requests." In: *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, pp. 325–330. 16, 22, 24–26,

Cavalcanti, Yguaratã et al. (2016). "Towards semi-automated assignment of software change requests." In: *Journal of Systems and Software* 115, pp. 82–101. 17, 24–26, 36,

Chaparro, Oscar (2017). "Improving bug reporting, duplicate detection, and localization." In: *Proceedings of the 39th International Conference on Software Engineering Companion*. IEEE Press, pp. 421–424. 14

Chen, Liguo, Xiaobo Wang, and Chao Liu (2010). "Improving bug assignment with bug tossing graphs and bug similarities." In: *Biomedical Engineering and Computer Science (ICBECS), 2010 International Conference on*. IEEE, pp. 1–5. 16

Cormack, Gordon V and Thomas R Lynam (2006). "Statistical precision of information retrieval evaluation." In: *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*. ACM, pp. 533–540. 31

Craswell, Nick et al. (2005). "Relevance weighting for query independent evidence." In: *Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval*. ACM, pp. 416–423. 71, 85, 86, 100

Čubranić, Davor and Gail Murphy (2004). "Automatic bug triage using text categorization." In: *In SEKE 2004: Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering*. Citeseer. Banff, Alberta, Canada. 16, 22, 25, 26,

Dedík, Václav and Bruno Rossi (2016). "Automated bug triaging in an industrial context." In: *Software Engineering and Advanced Applications (SEAA), 2016 42th Euromicro Conference on*. IEEE, pp. 363–367. 17

ElSalamouny, Ehab, Karl Tikjøb Krukow, and Vladimiro Sassone (2009). "An analysis of the exponential decay principle in probabilistic trust models." In: *Theoretical computer science* 410.41, pp. 4067–4084. 67

Florea, Adrian-Cătălin, John Anvik, and Răzvan Andonie (2017a). "Parallel Implementation of a Bug Report Assignment Recommender Using Deep Learning." In: *International Conference on Artificial Neural Networks*. Springer, pp. 64–71. 17

— (2017b). "Spark-based cluster implementation of a bug report assignment recommender system." In: *International Conference on Artificial Intelligence and Soft Computing*. Springer, pp. 31–42. 17

Fomel, Sergey and Jon F. Claerbout (2009). "Guest Editors' Introduction: Reproducible Research." In: *Computing in Science Engineering* 11.1, pp. 5–7. ISSN: 1521-9615. DOI: 10.1109/MCSE.2009.14. 11, 49

Fritz, Thomas et al. (2010). "A Degree-of-knowledge Model to Capture Source Code Familiarity." In: *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*. ICSE '10. Cape Town, South Africa: ACM, pp. 385–394. ISBN: 978-1-60558-719-6. 117, 137

Github (2017a). *Closing issues using keywords*. URL: https://help.github.com/articles/closing-issues-using-keywords/ (visited on 10/31/2011). 40, 51, 79

— (2017b). *Collaborators*. URL: https://developer.github.com/v3/repos/collaborators/ (visited on 10/31/2011). 37

Gousios, Georgios (2013). "The GHTorent dataset and tool suite." In: *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press, pp. 233–236. 142

Goyal, Anjali (2017). "Effective bug triage for non reproducible bugs." In: *Proceedings of the 39th International Conference on Software Engineering Companion*. IEEE Press, pp. 487–488. 17

Guo, Philip J et al. (2011). "Not my bug! and other reasons for software bug report reassignments." In: *Proceedings of the ACM 2011 conference on Computer supported cooperative work*. ACM, pp. 395–404. 1

Helming, Jonas et al. (2010). "Automatic assignment of work items." In: *International Conference on Evaluation of Novel Approaches to Software Engineering*. Springer, pp. 236–250. 14

Hosseini, Hadi, Raymond Nguyen, and Michael W Godfrey (2012). "A market-based bug allocation mechanism using predictive bug lifetimes." In: *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*. IEEE, pp. 149–158. 16, 19, 21

Hossen, Md Kamal, Huzefa Kagdi, and Denys Poshyvanyk (2014). "Amalgamating source code authors, maintainers, and change proneness to triage change requests." In: *Proceedings of the 22nd International Conference on Program Comprehension*. ACM, pp. 130–141. 16, 20, 25, 37,

Hu, Hao et al. (2014). "Effective bug triage based on historical bug-fix information." In: *Software Reliability Engineering (ISSRE), 2014 IEEE 25th International Symposium on*. IEEE, pp. 122–132. 15, 16, 20, 58,

Jain, Swati and Swapna Rose Wilson (2016). "Automated bug assortment system in datasets." In: *Inventive Computation Technologies (ICICT), International Conference on*. Vol. 2. IEEE, pp. 1–7. 17

Jain, Vibhor, Anand Rath, and Srini Ramaswamy (2012). "Field weighting for automatic bug triaging systems." In: *Systems, Man, and Cybernetics (SMC), 2012 IEEE International Conference on*. IEEE, pp. 2845–2848. 16

Jeong, Gaeul, Sunghun Kim, and Thomas Zimmermann (2009). "Improving Bug Triage with Bug Tossing Graphs." In: *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*. ESEC/FSE '09. Amsterdam, The Netherlands: ACM, pp. 111–120. ISBN: 978-1-60558-001-2. 1, 10, 16, 20, 2

Jie, Zhang et al. (2015). "A survey on bug-report analysis." In: *SCIENCE CHINA Information Sciences* 58, pp. 1–24. DOI: 10.1007/s11432-014-5241-2. 10

Jonsson, Leif (2013). "Increasing anomaly handling efficiency in large organizations using applied machine learning." In: *Software Engineering (ICSE), 2013 35th International Conference on*. IEEE, pp. 1361–1364. 16, 17, 21, 59,

Jonsson, Leif et al. (2012). "Towards automated anomaly report assignment in large complex systems using stacked generalization." In: *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*. IEEE, pp. 437–446. 16, 17, 21

Jonsson, Leif et al. (2016). "Automated bug assignment: Ensemble-based machine learning in large scale industrial contexts." In: *Empirical Software Engineering* 21.4, pp. 1533–1578. 1, 10, 11, 17, 5

Kagdi, Huzefa, Maen Hammad, and Jonathan I Maletic (2008). "Who can help me with this source code change?" In: *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*. IEEE, pp. 157–166. 16, 37

Kagdi, Huzefa and Denys Poshyvanyk (2009). "Who can help me with this change request?" In: *Program Comprehension, 2009. ICPC'09. IEEE 17th International Conference on*. IEEE, pp. 273–277. 16, 20

Kagdi, Huzefa et al. (2012). "Assigning change requests to software developers." In: *Journal of Software: Evolution and Process* 24.1, pp. 3–33. 16, 137

Karim, Muhammad Rezaul et al. (2016). "An empirical investigation of single-objective and multiobjective evolutionary algorithms for developer's assignment to bugs." In: *Journal of Software: Evolution and Process* 28.12, pp. 1025–1060. 17, 18, 21, 59,

Kevic, Katja et al. (2013). "Collaborative bug triaging using textual similarities and change set analysis." In: *Cooperative and Human Aspects of Software Engineering (CHASE), 2013 6th International Workshop on*. IEEE, pp. 17–24. 16

Khalil, Elias, Mustafa Assaf, and Abdel Salam Sayyad (2017). "Human resource optimization for bug fixing: balancing short-term and long-term objectives." In: *International Symposium on Search Based Software Engineering*. Springer, pp. 124–129. 17, 19, 21, 59

Khatun, Afrina and Kazi Sakib (2016). "A bug assignment technique based on bug fixing expertise and source commit recency of developers." In: *Computer and Information Technology (ICCIT), 2016 19th International Conference on*. IEEE, pp. 592–597. 15, 17, 37, 38,

Kim, Sunghun and E James Whitehead Jr (2006). "How long did it take to fix bugs?" In: *Proceedings of the 2006 international workshop on Mining software repositories*. ACM, pp. 173–174. 1, 10

Kitchenham, Barbara. and Stuart Charters (2007). "Guidelines for performing Systematic Literature Reviews in Software Engineering." In: *Software Engineering Group, School of Computer Science and Mathematics, Keele University and Department of Computer Science, University of Durham, Tech. Rep., EBSE*. URL: `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.117.471`. 13

Kleinberg, Jon M (1999). "Hubs, authorities, and communities." In: *ACM Computing Surveys (CSUR)* 31.4es, p. 5. 117

Lamkanfi, Ahmed et al. (2011). "Comparing mining algorithms for predicting the severity of a reported bug." In: *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*. IEEE, pp. 249–258. 117, 118, 127,

Lee, SunRo et al. (2017). "Applying deep learning based automatic bug triager to industrial projects." In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, pp. 926–931. 17, 46

Li, Guo et al. (2015). "Is It Good to Be Like Wikipedia?: Exploring the Trade-offs of Introducing Collaborative Editing Model to Q&A Sites." In: *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing*. ACM, pp. 1080–1091. 60

Lin, Zhongpeng et al. (2009). "An empirical study on bug assignment automation using Chinese bug data." In: *Empirical software engineering and measurement, 2009. ESEM 2009. 3rd international symposium on*. IEEE, pp. 451–455. 16, 117, 118, 1

Linares-Vásquez, Mario et al. (2012). "Triaging incoming change requests: Bug or commit history, or code authorship?" In: *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. IEEE, pp. 451–460. 1, 10, 16, 57, 1

Liu, Jin et al. (2016). "A Multi-Source Approach for Bug Triage." In: *International Journal of Software Engineering and Knowledge Engineering* 26.09n10, pp. 1593–1604. 1, 10, 17, 18, 5

Lukins, Stacy K, Nicholas A Kraft, and Letha H Etzkorn (2010). "Bug localization using latent dirichlet allocation." In: *Information and Software Technology* 52.9, pp. 972–990. 14

Manning, Christopher, Parbhakar Raghavan, and Hinrich Schütze (2008). *Introduction to Information Retrieval*. Cambridge University Press. 28–31, 38, 39,

Manning, Christopher, Hinrich Schütze, et al. (1999). *Foundations of statistical natural language processing*. Vol. 999. MIT Press. 65

Matter, Dominique, Adrian Kuhn, and Oscar Nierstrasz (2009). "Assigning bug reports using a vocabulary-based expertise model of developers." In: *Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on*. IEEE, pp. 131–140. 15, 16, 23, 25,

Meldrum, Sarah, Sherlock A Licorish, and Bastin Tony Roy Savarimuthu (2017). "Crowdsourced Knowledge on Stack Overflow: A Systematic Mapping Study." In: *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*. ACM, pp. 180–185. 60

Mockus, Audris and James D. Herbsleb (2002). "Expertise Browser: A Quantitative Approach to Identifying Expertise." In: *Proceedings of the 24th International Conference on Software Engineering*. ICSE '02. Orlando, Florida, USA: ACM, pp. 503–512. ISBN: 1-58113-472-X. 117, 137

Naguib, Hoda et al. (2013). "Bug report assignee recommendation using activity profiles." In: *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*. IEEE. 16, 118

Nasim, Sana, Saad Razzaq, and Javed Ferzund (2011). "Automated change request triage using alpha frequency matrix." In: *Frontiers of Information Technology (FIT), 2011*. IEEE, pp. 298–302. 14, 16

Nguyen, Tung Thanh, Anh Tuan Nguyen, and Tien N. Nguyen (2014). "Topic-based, Time-aware Bug Assignment." In: *SIGSOFT Softw. Eng. Notes* 39.1, pp. 1–4. ISSN: 0163-5948. DOI: 10.1145/2557833.2560585. URL: http://doi.acm.org/10.1145/2557833.2560585. 1, 10, 16, 18, 2

Park, Jinwoo et al. (2011). "Costriage: A cost-aware triage algorithm for bug reporting systems." In: *Proceedings of the National Conference on Artificial Intelligence*, p. 139. 16, 18, 20, 22,

— (2016). "Cost-aware triage ranking algorithms for bug reporting systems." In: *Knowledge and Information Systems* 48.3, pp. 679–705. 17, 18, 22, 94

Peng, Roger D (2011). "Reproducible research in computational science." In: *Science* 334.6060, pp. 1226–1227. 11, 49

Ponzanelli, Luca, Andrea Mocci, and Michele Lanza (2015). "Stormed: Stack overflow ready made data." In: *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on*. IEEE, pp. 474–477. 60

Poshyvanyk, Denys and Andrian Marcus (2007). "Combining formal concept analysis with information retrieval for concept location in source code." In: *Program Comprehension, 2007. ICPC'07. 15th IEEE International Conference on*. IEEE, pp. 37–48. 119

Rahman, Md Mainur, Guenther Ruhe, and Thomas Zimmermann (2009). "Optimized assignment of developers for fixing bugs an initial evaluation for eclipse projects." In: *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*. IEEE Computer Society, pp. 439–442. 16, 37

Rahmana, Md Mainur et al. (2012). "An empirical investigation of a genetic algorithm for developer's assignment to bugs." In: *Proceedings of the First North American Search based Symposium*. 16, 19, 21

Saha, Ripon K, Sarfraz Khurshid, and Dewayne E Perry (2015). "Understanding the triaging and fixing processes of long lived bugs." In: *Information and Software Technology* 65, pp. 114–128. 1, 10, 57

Sahu, Tirath Prasad, Naresh Kumar Nagwani, and Shrish Verma (2016). "An empirical analysis on reducing open source software development tasks using stack overflow." In: *Indian Journal of Science and Technology* 9.21. 22

Sajedi-Badashian, Ali (2016). "Realistic bug triaging." In: *Doctoral Symposium, Software Engineering Companion (ICSE-C), IEEE/ACM International Conference on*. IEEE, pp. 847–850. v

Sajedi-Badashian, Ali, Abram Hindle, and Eleni Stroulia (2015). "Crowdsourced Bug Triaging." In: *ICSME '15*. Bremen, Germany: IEEE. 17, 37, 67

— (2016). "Crowdsourced Bug Triaging: Leveraging Q&A Platforms for Bug Assignment." In: *Proceedings of 19th International Conference on Fundamental Approaches to Software Engineering (FASE)*. FASE '16. Eindhoven, The Netherlands: Springer. 17, 20, 22, 31,

Sajedi-Badashian, Ali, Vraj Shah, and Eleni Stroulia (2015). "GitHub's big data adaptor: an eclipse plugin." In: *Proceedings of the 25th Annual International Conference on Computer Science and Software Engineering*. IBM Corp., pp. 265–268. v

Sajedi-Badashian, Ali and Eleni Stroulia (2016). "Measuring user influence in github: the million follower fallacy." In: *CrowdSourcing in Software Engineering (CSI-SE), 2016 IEEE/ACM 3rd International Workshop on*. IEEE, pp. 15–21. v

— (2018a). "A Systematic Framework for Evaluating Bug-assignment Research." In: *Manuscript under review*. iv, 63, 68, 73,

— (2018b). "The Information Value of Different Sources of Evidence of Developers' Expertise for Bug Assignment." In: *Manuscript under review*. iv

— (2018c). "TTBA: Thesaurus and Time Based Bug-Assignment." In: *Manuscript under review*. iv, 78, 79, 93,

Sajedi-Badashian, Ali et al. (2014). "Involvement, Contribution and Influence in GitHub and Stack Overflow." In: *Proceedings of 24th International Conference on Computer Science and Software Engineering*. CASCON '14. Markham, Ontario, Canada: IBM Corp., pp. 19–33. URL: `http://dl.acm.org/citation.cfm?id=2735522.2735527`. v, 120, 125, 13

Schwab, Matthias, Martin Karrenbach, and Jon Claerbout (2000). "Making scientific computations reproducible." In: *Computing in Science & Engineering* 2.6, pp. 61–67. 11

Seacord, Robert C, Daniel Plakosh, and Grace A Lewis (2003). *Modernizing legacy systems: software technologies, engineering processes, and business practices*. Addison-Wesley Professional. 1, 10

Sean, JR, M McNee, and JA Konstan. "Accurate is not always good: How accuracy metrics have hurt recommender systems." In: *extended abstracts on Human factors in computing systems (CHI06) p*, pp. 1097–1101. 47

Servant, Francisco and James A Jones (2012). "WhoseFault: automatic developer-to-fault assignment through fault localization." In: *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, pp. 36–46. 16, 67, 70

Shani, Guy and Asela Gunawardana (2011). "Evaluating recommendation systems." In: *Recommender systems handbook*. Springer, pp. 257–297. 28, 29, 31

Sharma, Meera, Madhu Kumari, and VB Singh (2015). "Bug assignee prediction using association rule mining." In: *International Conference on Computational Science and Its Applications*. Springer, pp. 444–457. 17

Shi, Yue et al. (2012). "TFMAP: optimizing MAP for top-n context-aware recommendation." In: *Proceedings of the 35th international ACM SIGIR conference on Research and development in information retrieval*. ACM, pp. 155–164. 29, 31

Shokripour, Ramin et al. (2012). "Automatic Bug Assignment Using Information Extraction Methods." In: *Advanced Computer Science Applications and Technologies (ACSAT), 2012 International Conference on*, pp. 144–149. DOI: 10.1109/ACSAT.2012.56. 16, 19, 23, 25,

Shokripour, Ramin et al. (2013). "Why so complicated? simple term filtering and weighting for location-based bug report assignment recommendation." In: *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press, pp. 2–11. 1, 10, 16, 20, 5

— (2014). "Improving automatic bug assignment using time-metadata in term-weighting." In: *IET Software* 8.6, pp. 269–278. 17

— (2015). "A time-based approach to automatic bug report assignment." In: *Journal of Systems and Software* 102, pp. 109–122. 15, 17, 25, 38,

Somasundaram, Kalyanasundaram and Gail Murphy (2012). "Automatic categorization of bug reports using latent dirichlet allocation." In: *Proceedings of the 5th India software engineering conference*. ACM, pp. 125–130. 14

Stack Exchange, Inc. *Stack Exchange Data Dump.* "https://archive.org/details/stackexchange", Visited on 2014/08/20. 125, 141

Stack Exchange Team. *What are tags, and how should I use them?* "http://stackoverflow.com/help/tagging" Visited on 2015/03/17. 122

Stack Exchange, Inc (2017). *Stack Exchange Data Dump*. URL: https://archive.org/details/stackexchange (visited on 11/10/2016). 69, 100

Stack Exchange, Stack Exchange Community (2014). *Is there a direct download link with a raw data dump of Stack Overflow?* URL: http://meta.stackexchange.com/questions/198915/is-there-a-direct-download-link-with-a-raw-data-dump-of-stack-overflow-not-a-t (visited on 08/20/2014). 125, 141

Sun, Xiaobing et al. (2014). "Empirical studies on the nlp techniques for source code data preprocessing." In: *Proceedings of the 2014 3rd International Workshop on Evidential Assessment of Software Technologies*. ACM, pp. 32–39. 25, 59

Sun, Xiaobing et al. (2017). "Enhancing developer recommendation with supplementary information via mining historical commits." In: *Journal of Systems and Software* 134, pp. 355–368. 17, 24–26, 32,

Tamrawi, Ahmed et al. (2011a). "Fuzzy set and cache-based approach for bug triaging." In: *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, pp. 365–375. 16, 23, 25, 26,

— (2011b). "Fuzzy set-based automatic bug triaging: NIER track." In: *Software Engineering (ICSE), 2011 33rd International Conference on*. IEEE, pp. 884–887. 16, 20, 22, 23,

Tatsis, Vasileios A and Konstantinos E Parsopoulos (2016). "Grid search for operator and parameter control in differential evolution." In: *Proceedings of the 9th Hellenic Conference on Artificial Intelligence*. ACM, p. 7. 71, 85, 102

Teyton, Cédric et al. (2014). "Automatic extraction of developer expertise." In: *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*. ACM, p. 8. 137

The GHTorrent Project. *MySQL database dumps.* "http://GHTorrent.org/downloads/mysql-2014-08-18.sql.gz", Visited on 2014/08/20. 125

Tian, Yuan et al. (2016). "Learning to rank for bug report assignee recommendation." In: *Program Comprehension (ICPC), 2016 IEEE 24th International Conference on*. IEEE, pp. 1–10. 1, 10, 17, 57, 5

Vasilescu, Bogdan, Vladimir Filkov, and Alexander Serebrenik (2013). "StackOverflow and GitHub: associations between software development and crowdsourced knowledge." In: *Social Computing (SocialCom), 2013 International Conference on*. IEEE, pp. 188–195. 125, 138, 142

Venkataramani, Rahul et al. (2013). "Discovery of Technical Expertise from Open Source Code Repositories." In: *Proceedings of the 22Nd International Conference on World Wide Web Companion*. WWW '13 Companion. Rio de Janeiro, Brazil: International World Wide Web Conferences Steering Committee, pp. 97–98. ISBN: 978-1-4503-2038-2. 117, 137

Wang, Shaowei et al. (2014). "Entagrec: an enhanced tag recommendation system for software information sites." In: *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*. IEEE, pp. 291–300. 146

Wang, Song, Wen Zhang, and Qing Wang (2014). "FixerCache: Unsupervised caching active developers for diverse bug triage." In: *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, p. 25. 17, 22, 47

Wang, Xiaoyin et al. (2008). "An approach to detecting duplicate bug reports using natural language and execution information." In: *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*. IEEE, pp. 461–470. 14

Weidt, Frâncila and Rodrigo Silva (2016). "Systematic Literature Review in Computer Science-A Practical Guide." In: *Relatórios Técnicos do DCC/UFJF* 1. 13

Wong, Chu-Pan et al. (2014). "Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis." In: *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*. IEEE, pp. 181–190. 143

Wu, Wenjin et al. (2011). "Drex: Developer recommendation with k-nearest-neighbor search and expertise ranking." In: *Software Engineering Conference (APSEC), 2011 18th Asia Pacific*. IEEE, pp. 389–396. 16

Xia, Xin et al. (2017). "Improving automated bug triaging with specialized topic model." In: *IEEE Transactions on Software Engineering* 43.3, pp. 272–297. 17

Xie, Xihao et al. (2012). "Dretom: Developer recommendation based on topic models for bug resolution." In: *Proceedings of the 8th international conference on predictive models in software engineering*. ACM, pp. 19–28. 16, 92

Xu, Guandong, Yanchun Zhang, and Lin Li (2010). *Web mining and social networking: techniques and applications*. Vol. 6. Springer Science & Business Media. 28

Xuan, Jifeng et al. (2012). "Developer prioritization in bug repositories." In: *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, pp. 25–35. 16, 20

Yan, Meng et al. (2016). "A component recommender for bug reports using Discriminative Probability Latent Semantic Analysis." In: *Information and Software Technology* 73, pp. 37–51. 14

Yang, Geunseok, Tao Zhang, and Byungjeong Lee (2014). "Utilizing a multi-developer network-based developer recommendation algorithm to fix bugs effectively." In: *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. ACM, pp. 1134–1139. 16

Yu, Yue et al. (2016). "Reviewer recommendation for pull-requests in GitHub: What can we learn from code review and bug assignment?" In: *Information and Software Technology* 74, pp. 204–218. 14

Zanjani, Motahareh Bahrami (2016). "Effective assignment and assistance to software developers and reviewers." In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, pp. 1091–1093. 17

Zanjani, Motahareh Bahrami, Huzefa Kagdi, and Christian Bird (2015). "Using developer-interaction trails to triage change requests." In: *Proceedings of the 12th Working Conference on Mining Software Repositories*. IEEE Press, pp. 88–98. 17, 20, 59, 137

Zhai, ChengXiang and John Lafferty (2002). "Two-stage language models for information retrieval." In: *Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval*. ACM, pp. 49–56. 71, 85, 102

Zhang, Hongyu, Liang Gong, and Steve Versteeg (2013). "Predicting bug-fixing time: an empirical study of commercial software projects." In: *Proceedings of the 2013 international conference on software engineering*. IEEE Press, pp. 1042–1051. 14

Zhang, Jun, Mark S. Ackerman, and Lada Adamic (2007). "Expertise Networks in Online Communities: Structure and Algorithms." In: *Proceedings of the 16th International Conference on World Wide Web*. WWW '07. Banff, Alberta, Canada: ACM, pp. 221–230. ISBN: 978-1-59593-654-7. 117, 121, 129,

Zhang, Mi and Neil Hurley (2008). "Avoiding monotony: improving the diversity of recommendation lists." In: *Proceedings of the 2008 ACM conference on Recommender systems*. ACM, pp. 123–130. 47

Zhang, Tao and Byungjeong Lee (2012). "An automated bug triage approach: A concept profile and social network based developer recommendation." In: *International Conference on Intelligent Computing*. Springer, pp. 505–512. 16

— (2013). "A hybrid bug triage algorithm for developer recommendation." In: *Proceedings of the 28th annual ACM symposium on applied computing*. ACM, pp. 1088–1094. 16

Zhang, Tao et al. (2016). "Towards more accurate severity prediction and fixer recommendation of software bugs." In: *Journal of Systems and Software* 117, pp. 166–184. 2, 10, 17, 20, 3

Zhang, Tao et al. (2017). "Bug report enrichment with application of automated fixer recommendation." In: *Proceedings of the 25th International Conference on Program Comprehension*. IEEE Press, pp. 230–240. 17, 22

Zhang, Wen, Song Wang, and Qing Wang (2016a). "BAHA: A Novel Approach to Automatic Bug Report Assignment with Topic Modeling and Heterogeneous Network Analysis." In: *Chinese Journal of Electronics* 25.6, pp. 1011–1018. 17

— (2016b). "KSAP: An approach to bug report assignment using KNN search and heterogeneous proximity." In: *Information and Software Technology* 70, pp. 68–84. 2, 10, 17, 20, 2

Zhang, Xunhui et al. (2017). "DevRec: A Developer Recommendation System for Open Source Repositories." In: *International Conference on Software Reuse*. Springer, pp. 3–11. 22

Zimmermann, Thomas et al. (2012). "Characterizing and predicting which bugs get reopened." In: *34th International Conference on Software Engineering (ICSE), 2012*. IEEE, pp. 1074–1083. 119