# EFFECTIVE ASSIGNMENT AND ASSISTANCE TO SOFTWARE DEVELOPERS AND REVIEWERS

A Dissertation by

Motahareh Bahrami Zanjani

Master of Science, Islamic Azad University, 2010

Submitted to the Department of Electrical Engineering and Computer Science
and the faculty of the Graduate School of
Wichita State University
in partial fulfillment of
the requirements for the degree of
Doctor of Philosophy

May 2017

**EFFECTIVE ASSIGNMENT AND ASSISTANCE TO SOFTWARE DEVELOPERS AND REVIEWERS**

The following faculty members have examined the final copy of this dissertation for form and content, and recommend that it be accepted in partial fulfillment of the requirement for the degree of Doctor of Philosophy with a major in Electrical Engineering and Computer Science.

Huzefa Kagdi, Committee Chair

Christian Bird, Committee Member

Vinod Namboodiri, Committee Member

Ehsan Salari, Committee Member

Sergio Salinas Monroy, Committee Member

Kaushik Sinha, Committee Member

Accepted for the College of Engineering

Royce Bowden, Dean

Accepted for the Graduate School

Dennis Livesay, Dean

## DEDICATION

To my parents and my sister.

## ACKNOWLEDGEMENTS

First of all I would like to express my gratitude to my supervisor and dissertation committee chair, Dr. Huzefa Kagdi, whose expertise added considerably to my graduate experience. His knowledge and expertise in area of Software evolution is admirable. Without his guidance this dissertation would not be possible. I would also like to thank Dr. Christian Bird from Empirical Software Engineering (ESE) group at Microsoft Research. He provided ample guidance and suggestion related to this dissertation work. His extensive knowledge in this research area enlightened me in the right direction. I would also like to thank other dissertation committee members who put their valuable time to review my dissertation and make it a successful one. My sincere gratitude goes to the faculty members that I came across during my entire graduate program at Wichita State University. Last but not least, the extraordinary support I received from my parents, family, and friends has been the most important part of my journey throughout my graduate studies. I would like to extend my special gratitude to them for supporting me in every step of the way.

## ABSTRACT

The conducted research is within the realm of software maintenance and evolution. Human reliance and dominance are ubiquitous in sustaining a high-quality large software system. Automatically assigning the right solution providers to the maintenance task at hand is arguably as important as providing the right tool support for it, especially in the far too commonly found state of inadequate or obsolete documentation of large-scale software systems. Several maintenance tasks related to assignment and assistance to software developers and reviewers are addressed, and multiple solutions are presented. The key insight behind the presented solutions is the analysis and use of micro-levels of human-to-code and human-to-human interactions. The formulated methodology consists of the restrained use of machine learning techniques, lightweight source code analysis, and mathematical quantification of different markers of developer and reviewer expertise from these micro interactions.

In this dissertation, we first present the automated solutions for Software Change Impact Analysis based on interaction and code review activities of developers. Then we provide explanation for two separate developer expertise models which use the micro-levels of human-to-code and human-to-human interactions from the previous code review and interaction activities of developers. Next, we present a reviewer expertise model based on code review activities of developers and show how this expertise model can be used for Code Reviewer Recommendation. At the end we examine the influential features that characterize the acceptance probability of a submitted patch (implemented code change) by developers. We present a predictive model that classifies whether a patch will be accepted or not as soon as it is submitted for code review in order to assist developers and reviewers in prioritizing and focusing their efforts.

A rigorous empirical validation on large open source and commercial systems shows that the solutions based on the presented methodology outperform several existing solutions. The quantitative gains of our solutions across a spectrum of evaluation metrics along with their statistical significance are reported.

## TABLE OF CONTENTS

**Chapter** **Page**


1 Introduction . . . . . 1

1.1 Definitions . . . . . 2
1.2 Conducted Research and Projected Contributions . . . . . 3
1.3 Overview of The Presented Approaches . . . . . 4
1.4 Dissertation Findings . . . . . 6
1.5 Dissertation Organization . . . . . 7

2 Background and Related Researches . . . . . 8

2.1 Software Maintenance Background Contextualizing the Presented Work . . . . . 8
2.2 Micro-Evolution Repositories . . . . . 10
2.2.1 Interaction Histories from IDEs . . . . . 11
2.2.2 Code Review Histories . . . . . 11
2.3 Macro-Evolution Repositories . . . . . 14
2.4 Related Work . . . . . 15
2.4.1 Interaction History . . . . . 15
2.4.2 Code Review . . . . . 16
2.4.3 Software Change Impact Analysis . . . . . 18
2.4.4 Developer Recommendation . . . . . 20
2.4.5 Reviewer Recommendation . . . . . 22
2.4.6 Predicting the outcome of code review . . . . . 23

3 Change Impact Analysis (*InComIA*) . . . . . 24

3.1 Introduction . . . . . 24
3.2 The *InComIA* Approach . . . . . 26
3.2.1 Interactions and Commits for Change Request Resolution . . . . . 27
3.2.2 Extracting Interacted Entities . . . . . 29
3.2.3 Extracting Committed Entities . . . . . 30
3.2.4 Obtaining Source Code of Entities from Interacted and Committed Revisions 31
3.2.5 Creating a Corpus from Source Code and Textual Descriptions . . . . . 32
3.2.6 Indexing and Querying the Corpus . . . . . 34
3.2.7 An Example from *Mylyn* . . . . . 36
3.3 Empirical Evaluation . . . . . 37
3.3.1 Research Questions . . . . . 38
3.3.2 Experiment Setup . . . . . 39
3.3.3 Subject Software System . . . . . 39
3.3.4 Dataset . . . . . 39
3.3.5 Training and Testing Sets . . . . . 40
3.3.6 Performance Metrics . . . . . 40
3.3.7 Hypotheses Testing . . . . . 41
3.3.8 Case Study Results . . . . . 42
3.4 Threats to Validity . . . . . 45

**TABLE OF CONTENTS (continued)**

**Chapter** **Page**


4 Change Impact Analysis (*RevIA*) . . . . . 47

4.1 Introduction . . . . . 47
4.2 The *RevIA* Approach . . . . . 49
4.2.1 Extracting Code Review Comments and Creating a Corpus . . . . . 50
4.2.2 Re-Ranking the Recommended Source Code Files with Issue Change Proneness . . . . . 51
4.3 Empirical Evaluation . . . . . 52
4.3.1 Research Questions . . . . . 52
4.3.2 Experiment Setup . . . . . 52
4.3.3 Subject Software System . . . . . 53
4.3.4 Training and Testing Sets . . . . . 53
4.3.5 Performance Metrics . . . . . 53
4.3.6 Hypotheses Testing . . . . . 54
4.3.7 Case Study Results . . . . . 54
4.4 Threats to Validity . . . . . 57

5 Developer Recommendation (*iHDev*) . . . . . 58

5.1 Introduction . . . . . 58
5.2 Approach . . . . . 61
5.2.1 Key Terms and Definition . . . . . 61
5.2.2 Locating Relevant Entities to Change Request . . . . . 63
5.2.3 Mining Interaction Histories to Recommend Developers . . . . . 64
5.2.4 An Example from *Mylyn* . . . . . 70
5.3 Case Study . . . . . 71
5.3.1 Compared Approaches: *xFinder*, *xFinder'*, and *iMacPro* . . . . . 72
5.3.2 Subject Software Systems . . . . . 72
5.3.3 Benchmarks: Training and Testing Datasets . . . . . 74
5.3.4 Metrics and Statistical Analyses . . . . . 75
5.3.5 Results . . . . . 77
5.3.6 Discussion . . . . . 80
5.4 Threats to Validity . . . . . 81

6 Developer Recommendation (*rDevX*) . . . . . 84

6.1 Introduction . . . . . 84
6.2 The Developer Expertise Model . . . . . 86
6.2.1 Why Code Reviews to Build a New Model for Developer Expertise? . . . . 87
6.2.2 Markers and Expertise Model . . . . . 91
6.3 Application for Recommending Appropriate Developers in Change Request Triaging 94
6.3.1 Locating Relevant Entities to Change Request . . . . . 95
6.3.2 Recommending developers based on SoE scores . . . . . 97
6.3.3 An Example from *Mylyn* . . . . . 98

## TABLE OF CONTENTS (continued)

**Chapter** **Page**


6.4 Case Study . . . 100
6.4.1 *xFinder* . . . 101
6.4.2 *DevCom* . . . 102
6.4.3 Benchmarks: Bugs, Commits, and Reviews . . . 103
6.4.4 Metrics and Hypotheses . . . 105
6.4.5 Results . . . 107
6.4.6 Qualitative Analysis . . . 110
6.5 Threats to Validity . . . 111

7 Code Review Recommendation (*cHRev*) . . . 113

7.1 Introduction . . . 113
7.2 Background on Modern Code Review . . . 116
7.3 The *cHRev* Approach . . . 117
7.3.1 Formulating Reviewer Expertise Model . . . 118
7.3.2 Scoring and Recommending reviewers . . . 120
7.3.3 Implementation of *cHRev* . . . 122
7.3.4 A Motivating Example from *Mylyn* . . . 122
7.4 Case Study . . . 124
7.4.1 Design . . . 124
7.4.2 Compared Approaches: *REVFINDER*, *xFinder* and *RevCom* . . . 125
7.4.3 Subject Systems and Evaluation Datasets . . . 126
7.4.4 Evaluation Protocol for *cHRev* . . . 128
7.4.5 Accuracy Metrics and Hypothesis Testing . . . 129
7.4.6 Results . . . 131
7.4.7 Discussion . . . 136
7.5 Threats to Validity . . . 139
7.6 Related Work . . . 140
7.6.1 Developer Recommendation . . . 141

8 Patch Acceptance Predictor . . . 143

8.1 Introduction . . . 144
8.2 Background on Modern Code Review (MCR) and Subject Systems . . . 146
8.2.1 Definitions of Key Terms . . . 146
8.2.2 Patch Lifecycle in *Gerrit* . . . 147
8.2.3 Merged and Abandoned patches . . . 149
8.2.4 Subject Software Systems . . . 149
8.3 Features Used in the Study . . . 151
8.3.1 Bug Features . . . 151
8.3.2 Patch Features . . . 155
8.3.3 Human Features . . . 159
8.4 Patch Acceptance Descriptive model . . . 164
8.4.1 Logistic Regression . . . 165

## TABLE OF CONTENTS (continued)

**Chapter** **Page**


8.4.2 Data Extraction . . . 165
8.4.3 Results . . . 167
8.5 Patch Acceptance Predictive Model . . . 175
8.5.1 Support Vector Machines . . . 175
8.5.2 Predictive Featured Used . . . 176
8.5.3 Training and Test Sets . . . 176
8.5.4 Evaluation Metrics . . . 176
8.6 Predictive Results . . . 177
8.7 Threats to Validity . . . 180

9 Conclusions and Future Work . . . 182

9.1 Contribution and Findings . . . 182
9.1.1 *InComIA*, an Approach for Change Impact Analysis . . . 182
9.1.2 *RevIA*, an Approach for Change Impact Analysis . . . 183
9.1.3 *iHDev*, an Approach for Developer Recommendation . . . 183
9.1.4 *rDevX*, the Developer Expertise Model . . . 184
9.1.5 *cHRev*, the Code Reviewer Recommendation Model . . . 184
9.1.6 Patch Acceptance Predictor . . . 185
9.2 Opportunities for Future Research . . . 185
9.2.1 Combining Static and Dynamic Analysis with Textual Information . . . 185
9.2.2 a Developer Recommendation Approach Considering the Workload Balancing . . . 186
9.2.3 Considering the Contribution of Developers of Similar Change Requests . 186
9.2.4 Developer and Reviewers Expertise Markers from Code Review Contents . 186
9.2.5 Revisiting Code Reviewer Recommendation . . . 186
9.2.6 Additional Empirical Studies Considering Different MCR Tools . . . 187
9.2.7 Revisiting the Patch Acceptance Predictor . . . 187

REFERENCES . . . 188

## LIST OF TABLES


| Table | | Page |
|---|---|---|
| 3.1 | Predicted files for bug# 201151 by three different approaches | 36 |
| 3.2 | Different comments and expressions extracted from file *AbstractRepositorySettingsPage.java* in different revision numbers and associated commit messages | 37 |
| 3.3 | Different comments and expressions extracted from file *BugzillaRepositorySettingsPage.java* in different revision numbers and associated commit messages | 37 |
| 3.4 | *Mylyn* project interaction and commit histories from June 18, 2007 to July 01,2011. | 40 |
| 3.5 | A heat-map summarizing hypotheses test results across all the three models for two different values of k. | 42 |
| 3.6 | Recall@10 and 20 and Precision@10 and 20 of three models *InComIA*, *ComIA*, and *SIA* | 44 |
| 5.1 | The attachers extracted with *iHDev* from each of the top ten files relevant to Bug# 313712. | 68 |
| 5.2 | Top five attachers (developers) recommended to resolve bug #313712 by *iHDev*. | 70 |
| 5.3 | Top five recommended developers and their associated ranks for the compared approaches. *iHDev*, *xFinder*, *xFinder0* and *iMacPro*. | 71 |
| 5.4 | The frequency distributions of developers resolving issues in the benchmarks for *Mylyn* and *Eclipse Platform*. | 75 |
| 5.5 | Average of recall @1, 2, 3 and 5 values of the approaches *iHDev*, *xFinder*, *xFinder0*, and *iMacPro* measured on the *Mylyn* and *Eclipse Platform* benchmarks. | 78 |
| 5.6 | Mean Reciprocal Rank of the approaches *iHDev*, *xFinder*, *xFinder0*, and *iMacPro* measured on the *Mylyn* and *Eclipse Platform* benchmarks. | 80 |
| 6.1 | The developers extracted with *rDevX* from each of the top ten files relevant to Bug# 428544. | 99 |
| 6.2 | Top five developers recommended to resolve bug #428544 by *rDevX* | 99 |
| 6.3 | Top five recommended developers and their associated ranks for the compared approaches | 100 |

## LIST OF TABLES (continued)

**Table** **Page**


6.4 The frequency distributions of developers and summary statistics (IQR, unique developers) resolving issues in the benchmarks for *Mylyn*, *Eclipse Platform* and *OpenStack Nova*. . . . . 105

6.5 Descriptive statistics of the review and commit history of the three open source subject systems . . . . . 105

6.6 Average, gains and p-values of recall @1, 2, 3 and 5 values of the approaches. . . . . 109

6.7 Mean Reciprocal Rank of the approaches *rDevX* , *xFinder*, and *DevCom* measured on the benchmarks. . . . . 110

6.8 p-values from applying one way ANOVA on MRR values for each subject system. . . 110

7.1 The reviewers extracted with *cHRev* from each of the files related to code change in the review # 33689. . . . . 122

7.2 Top four reviewers recommended to review the review #33689 with their associated ranks and score by *cHRev*, *REVFINDER*, *xFinder*, and *RevCom*. . . . . .124

7.3 Evaluation benchmarks and the distribution of reviewers per review (code change). . .131

7.4 Average of precision, recall, and F-score @1, 2, 3 and 5 values of the approaches *cHRev*, *REVFINDER*, *xFinder*, and *RevCom* measured on the benchmarks. . . . . .132

7.5 Average of precision, recall, and F-score gains @1, 2, 3 and 5 values of the approaches *cHRev*, *REVFINDER*, *xFinder*, and *RevCom* measured on the benchmarks. . . . . .132

7.6 Mean Reciprocal Rank of the approaches *cHRev*, *REVFINDER*, *xFinder*, and *RevCom* measured on the benchmarks. . . . . 133

7.7 p-values from applying one way ANOVA on Precision@m and Recall@m values for each subject system. . . . . 133

7.8 p-values from applying one way ANOVA on MRR values for each subject system. . . 133

8.1 Descriptive statistics of the reviews considered from three open source projects in our Study. . . . . 150

## LIST OF TABLES (continued)

**Table** **Page**


8.2 Descriptive statistics of the feature components considered from *Mylyn* and *Eclipse Platform*. . . . . 152

8.3 Explanation of bug severity values . . . . . 153

8.4 Percentage distribution of merged and abandoned reviews based on owner bug assignee match feature in *Eclipse Platform* and *Mylyn*. . . . . 155

8.6 P-values from statistical significance test for features used in the descriptive model with Logistic Regression. . . . . 168

8.7 Performance of Logistic Regression (LR). . . . . 178

8.8 Performance of Support Vector Machine (SVM). . . . . 179

8.9 Performance of Dummy Predictor and Zero model . . . . . 179

## LIST OF FIGURES

**Figure** **Page**


2.1 A Schematic diagram of presented approaches . . . . . 8

2.2 A snippet of an interaction event recorded by *Mylyn* for bug issue #330695 with trace ID #221752. File TaskListView.java is edited. . . . . 12

2.3 An example from *Gerrit* code review# 74981 related to *Eclipse Platform* project . . . . . 13

2.4 The patch life cycle in *Gerrit* . . . . . 14

3.1 A snippet of 4 interaction events (labelled 1-4) recorded by *Mylyn* for bug issue #175229 with trace ID #71687. . . . . 28

3.2 Bug ID #175229 from Eclipse bug tracking and commit history. . . . . 29

3.3 A Schematic diagram of *InComIA*. . . . . 31

3.4 A snippet of the file TaskHistoryTest.java . . . . . 33

4.1 A snippet of the code review comment in *Gerrit* which is written by *Sam Davis* for file *TaskReviewRelationshipListener.java* related to *Mylyn* project . . . . . .48

4.2 Recall, Precision, Mean Average Precision, and the calculated metric gains of two models *RevIA* and *SIA* with and without re-ranking mechanisms for K=10 and K=20. . 56

5.1 A snippet of an interaction event recorded by *Mylyn* for bug issue #330695 with trace ID #221752. File TaskListView.java is edited. . . . . 63

5.2 A snippet of the bug #315184 interaction log entry from its interaction log file. . . . . 65

8.1 An example from *Gerrit* code review# 74981 related to *Eclipse Platform* project . . . . 147

8.2 The patch life cycle in *Gerrit* . . . . . 148

8.3 Bug severity frequency distribution for merged and abandoned reviews in *Eclipse Platform* and *Mylyn* . . . . . 153

8.4 Bug priority frequency distribution for merged and abandoned reviews in *Eclipse Platform* and *Mylyn* . . . . . .154

## LIST OF FIGURES (continued)

**Figure** **Page**


8.5 Patch size group frequency distribution for merged and abandoned reviews in *Android Platform*, *Eclipse Platform*, and *Mylyn* . . . . . 156

8.6 File size Box Plot for merged and abandoned patches in *Android Platform*, *Eclipse Platform*, and *Mylyn* . . . . . 157

8.7 Number of patch revisions Box Plot for merged and abandoned patches in *Android Platform*, *Eclipse Platform*, and *Mylyn*. . . . . 157

8.8 Entropy score Box Plot for merged and abandoned patches belongs to *Android Platform*, *Eclipse Platform*, and *Mylyn*. . . . . 159

8.9 (a) Owner reputation Box Plot for merged and abandoned patches belongs to *Android Platform*, *Eclipse Platform*, and *Mylyn*. (b) Percentage distribution of merged reviews based on owner reputation . . . . . 161

8.10 Percentage distribution of merged reviews based on number of assigned/contributing reviewers . . . . . 162

8.11 A Box Plot for the number of assigned/contributing reviewers for merged and abandoned patches . . . . . 162

8.12 Percentage distribution of merged reviews based on assigned/contributing reviewers reputation . . . . . 163

8.13 A Box Plot for the assigned/contributing reviewers reputation for merged and abandoned patches . . . . . 164

# CHAPTER 1

## Introduction

Software products are constantly growing in terms of size, complexity, and application domains, among other things. It is not uncommon in large open source projects to receive several bug reports and new feature requests daily [9]. These change requests need to be effectively triaged and resolved in an efficient and effective manner to sustain (or even retain) the viability of the product in the marketplace – not a trivial ask by any means. The development units, i.e., individuals and teams, need to perform several tasks such as validating the change requests, assigning them to the developers(s), implement the necessary changes to the source code, review the code changes, and then assembling them into a (new) release to the user base.

Software engineering is very much human driven and prone, including their ingenuity and egregiousness. Predominantly, humans (with tool support) need to manage and perform these activities in a distributed software change management. The quality and velocity of maintenance and evolution tasks (e.g., bug fixes or a feature implementations) are in many ways a direct reflection of the individuals or teams who perform them. Determining the right solution providers for the task at hand is arguably as important as suggesting the right tool support for it, especially in the far too commonly found state of inadequate or obsolete documentation of large scale software systems.

Not surprisingly, a paramount effort has been devoted in software engineering research to effectively support the tasks related to humans involved in developing and evolving large-scale software systems. Among these key tasks are Developer Recommendation (DR), Software Change Impact Analysis (IA), Reviewer Recommendation (CR), and Patch Acceptance Predictor (PAR) . DR is the task of assigning the most appropriate developers to resolve an incoming change request [9, 53, 65, 94, 107]. IA is task of estimating the the potential impact on source code (and beyond) due to a proposed change [11, 12]. CR is the task of assigning the most appropriate reviewers to code review a source code change (e.g., a patch) [14]. PAR is a task of determining whether a submitted patch will be accepted or not.

Over the years, a number of approaches have been proposed in the literature to automate the IA and DR tasks [9, 11, 78, 29, 62, 75, 80]. Recently, there has been a few efforts to address the CR and PAR tasks [96, 98, 14, 17, 48]. Although, these investigations show ample progress, the resulting solutions are suboptimal, e.g., in terms of accuracy and coverage, whereby limiting or questioning their practical ubiquity in the software development workflow. We conjecture that the past solutions are severely hampered due to their principal reliance on a limited source of analysis. These solutions are based on analyzing a single snapshot of source code and/or its associated bug or change repositories. Although, these sources capture several important aspects, we posit that they are limited in scope and size. They only capture the *end points, i.e., macro events,* of software maintenance or evolution tasks (e.g., accepted code changes for bug fixes or feature requests), and do not necessarily capture *the means, i.e., micro events,* associated to achieve them.

### 1.1 Definitions

Software products are constantly growing in terms of size, complexity, and application domains. In this era of software governance, software products are seldom built from scratch, but are continuously evolved. In the life cycle of a software, maintenance is the final and perhaps the most expensive stage. All the tasks related to any post-delivery changes, such as change control and fixing bugs are handled in Maintenance phase. However, maintenance is considered to be the most expensive and long-lasting phase in the lifecycle of most software systems, where more than 50% of all maintenance costs arise from changing software [63]. Therefore, every development or analysis step which can be automated can save a lot of time and money. In this dissertation we investigate several maintenance tasks and we present automated solutions for these tasks. Below we explain each of these maintenance tasks in more detail.

**Software-change impact analysis (IA)** have been addressed for many years and date back to the 1970s. Bohner and Arnold [11, 12] investigated the foundations of impact analysis, and defined the term impact analysis as “Identifying the potential consequences of a change, or estimating what needs to be modified to accomplish a change [11]”. The automated tool for IA recommends the developer a set of relevant source code entities to the incoming change request.

**Developer Recommendation (DR)**, or bug triaging is a crucial activity in addressing change requests in an effective manner (e.g., within time, priority, and quality factors). It is not uncommon in large open source projects to receive several bug reports and new feature requests daily [9]. These change requests need to be effectively triaged. The task of automatically assigning issues or change requests to the developer(s) who are most likely to resolve them is called Developer Recommendation [9, 8].

**Code Reviewer Recommendation (CR)**: Code review is an important part of the software development process. Recently, many open source projects have begun practicing code review through "modern" tools such as GitHub pull-requests and Gerrit. These tools enable the owner of a source code change to request individuals to participate in the review, i.e., reviewers. However, this task comes with a challenge. Prior work has shown that the benefits of code review are dependent upon the expertise of the reviewers involved [13]. Thus, a common problem faced by authors of source code changes is that of identifying the best reviewers for their source code change. The task of automatically recommend reviewers who are best suited to participate in a given review is called Code Reviewer Recommendation [14].

**Patch Acceptance Predictor (PAR)**: Large-scale open source projects typically receive numerous patches to address change requests (e.g., bug fixes). It is not uncommon for a submitted patch to undergo a peer-review process before it can be accepted as a valid solution and merged to the code base. A peer-code review process has been shown to have impact on the software quality. Determining whether a submitted patch will be accepted or not could improve the efficiency of the peer-review process, e.g., help developers and reviewers prioritize and focus their efforts. Patch Acceptance Predictor is a predictive model that classifies whether a patch will be accepted or not as soon as it is submitted.

### 1.2 Conducted Research and Projected Contributions

In this work, we present automated solutions for DR, IA, and CR tasks that are centered on micro events. Similarly we present a predictive model that classifies whether a patch will be accepted or not as soon as it is submitted. Specifically, we forefront our analysis and root our so-

lutions in two types of micro-event sources: 1) Interaction archives which store the fine-granular events capturing the developer interactions within an Integrated Development Environment (IDE), e.g., with *Mylyn* and 2) Code review archives which capture the complete code change review life-cycle, including the associated discussion and discourse. The information in micro-event archives subsumes or complements the one in macro-event archives.

Our methodology consists of the restrained use of machine learning techniques, lightweight source code analysis, and mathematical quantification of different markers of developer and reviewer expertise. Based on this methodology, we instantiated specific approaches for the investigated tasks: 1) *InComIA* and *RevIA* for IA based on interaction and code review archives [112], 2) *iHDev* and *rDevX* for DR based on interaction and code-review archives respectively [111], 3) *cHRev* for CR based on code-review archives [110], and 4) Predictive model for PAR based on code-review archives. Our preliminary implementation and empirical evaluation show that our presented approaches outperform the existing ones in terms of a number of accuracy metrics on open source and commercial systems.

*The main contributions of our work include the first IA solutions to integrate the developer interaction and code review activity with the textual information from the source code. The first DR solutions based on interaction and code review archives and their empirical evaluation. The only empirical evaluation of CR solution in the commercial domain, and finally the predictive recommendation and assessment of whether a patch will be eventually accepted or not as soon as it is submitted for Modern Code Review were not investigated previously.*

### 1.3 Overview of The Presented Approaches

In this section we briefly explain an overview of each presented approach in our conducted research.

**Change Impact Analysis (*InComIA*):** This is an approach to perform impact analysis (IA) of an incoming change request on source code. The approach is based on a combination of interaction (e.g., *Mylyn*) and commit (e.g., CVS) histories. The source code entities (i.e., files and

methods) that were interacted or changed in the resolution of past change requests (e.g., bug fixes) were used. Information retrieval, machine learning, and lightweight source code analysis techniques were employed to form a corpus from these source code entities. Additionally, the corpus was augmented with the textual descriptions of the previously resolved change requests and their associated commit messages. Given a textual description of a change request, this corpus is queried to obtain a ranked list of relevant source code entities that are most likely change prone.

**Change Impact Analysis (*RevIA*):** Similar to previous approach, this is an approach to perform impact analysis (IA) of an incoming change request on source code. This approach uses the code review comments provided by developers in code review phase as an additional textual information to form the corpus. Information retrieval, machine learning, and lightweight source code analysis techniques were employed to form a corpus from the source code entities that are previously changed in the resolution of past change requests. Additionally, the corpus was augmented with the review comments written by developers during the review phase of previously submitted patches. These patches are the results of the implemented changed in the source code to resolve the change requests. Given a textual description of a change request, this corpus is queried to obtain a ranked list of relevant source code entities that are most likely change prone.

**Developer Recommendation (*iHDev*):** This is an approach to recommend developers who are most likely to implement incoming change requests. The basic premise of *iHDev* is that the developers who interacted with the source code relevant to a given change request are most likely to best assist with its resolution. A machine-learning technique is first used to locate source code entities relevant to the textual description of a given change request. *iHDev* then mines interaction trails (i.e., *Mylyn* sessions) associated with these source code entities to recommend a ranked list of developers.

**Developer Recommendation (*rDevX*):** This is an approach that uses the developer expertise markers from the previous code review activities of developers. We analyzed code reviews that are managed by ”modern” tools, such as *Gerrit*. We found eight markers of developer expertise associated with the source code changes and their acceptance, time line, and human roles

and feedback involved in the reviews. We formed a developer-expertise model from these markers and showed its application in bug triaging. Specifically, we derived a developer recommendation approach for an incoming change request (e.g., to fix a bug), named *rDevX*, from this expertise model.

**Code Review Recommendation (*cHRev*):** This is an approach to automatically recommend reviewers who are best suited to participate in a given review, based on their historical contributions as demonstrated in their prior reviews. It utilizes the past code changes and their reviewers to form a quantifiable model of the expertise of each reviewer in each source code file. This expertise model is then used to rank reviewers based on their expertise score. Finally, a ranked list of top n reviewers is recommended.

**Patch Acceptance Predictor:** This is a predictive model that classifies whether a patch will be accepted or not as soon as it is submitted. Logistic regression and support vector machine were employed to form the patch predictor. We examined bug, patch, developer, and reviewer features that characterize the patches that get accepted or not and used these features to build the classifier.

### 1.4 Dissertation Findings

Our findings show that:

1. Approach based on a combination of interaction (e.g., *Mylyn*) and commit (e.g., CVS) histories outperforms traditional approaches to perform impact analysis (IA) of an incoming change request on source code. (*InComIA*, Chapter 3)
2. Code review comments provide textual information related to both source code and bug itself. This textual information improves the performance of IA approaches (*RevIA*, Chapter 4)
3. Developers who interacted with the source code relevant to a given change request are most likely to best assist with its resolution. (*iHDev*, Chapter 5)
4. The historical code reviews capture many unique aspects that can be useful markers of developer expertise. (Chapter 6)

5. Developer expertise model based on previous code review activities of developers would typically recommend the correct developers at higher ranks than the competitor. (*rDevX*, Chapter 6)
6. Reviewers who reviewed the units of source code in the past are most likely to best assist with reviewing it in the future. (*cHRev*, Chapter 7)
7. Leveraging the specific information in previously completed reviews (i.e., quantification of review comments and their recency), could dramatically improve the performance of Reviewer Recommendation approach in comparison with the prior approaches, which (limitedly) operate on generic review information (i.e., reviewers of similar source code file and path names) or source code repository data. (*cHRev*, Chapter 7)
8. The most influential features that characterize the acceptance probability of a submitted patch by developers are, Patch Size, Number of Patch Revisions, Patch Owner's Reputation, Number of Assigned Reviewers, Number of Contributing Reviewers, and Contributing Reviewer's Reputation. (Chapter 8)

## 1.5 Dissertation Organization

The remainder of this dissertation is organized as follows. Chapter 2 provides background information and presents research related to our studies. Chapter 3 and 4 present *InComIA* and *RevIA* approaches for Change Impact Analysis with results from separated empirical studies. Chapter 5 and 6 present *iHDev* and *rDevX* approaches for Developer Recommendation with results from several empirical studies on open source and commercial software projects. Chapter 7 presents *cHRev*, our approach to select expert reviewers with results from empirical studies. Chapter 8 presents a classifier which predicts the outcome of code review. Finally, Chapter 9 draws conclusions, presents a summary of the main contributions of this thesis, and outlines the future work.

# CHAPTER 2

## Background and Related Researches

In this chapter, we first explain the Software maintenance background contextualizing the presented work and then provide the definition of micro and macro evolution repositories. We explain the interaction activity of developers and describe the Modern Code Review (MCR) process. Similarly, a survey of related researches on software maintenance tasks is provided in this chapter. More specifically, we describe how the related work motivates our presented approaches.

### 2.1 Software Maintenance Background Contextualizing the Presented Work

In the life cycle of a software, maintenance is the final and perhaps the most expensive stage. All the tasks related to any post-delivery changes, such as change control and fixing bugs are handled in Maintenance phase.

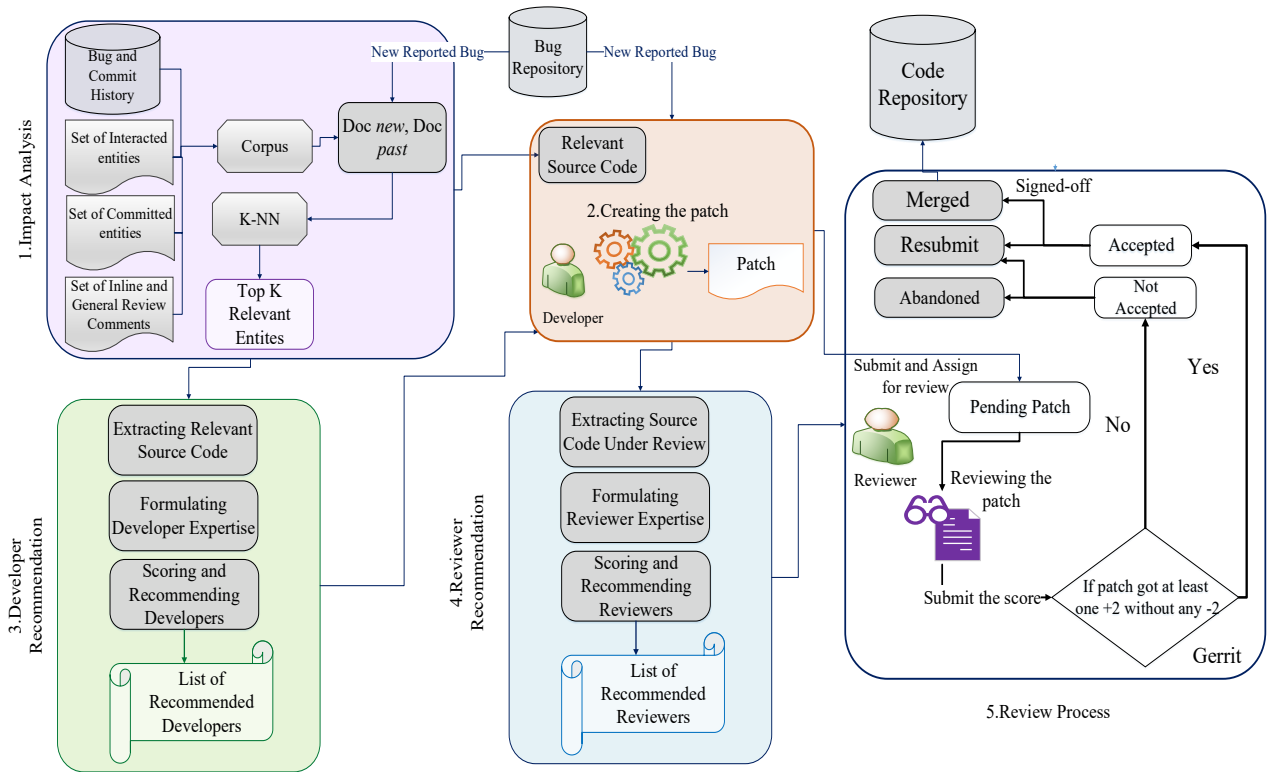


Figure 2.1: A Schematic diagram of presented approaches

The change requests and bugs reported in bug tracking tool (e.g., Bugzilla) are typically specified in natural language (e.g., English) and they mark the beginning of Maintenance phase. The bug reports include issues identified and submitted by programmers or end users during the post-delivery maintenance of a product and the change requests include the enhancements to the original product. In either case, developers need to identify the potential source code entities that are change prone due to a given change request. This task is called Impact Analysis (IA). An automated tool to help developers identify the related source code entities to simplify the task of issue fixing or implementing a new change request is needed. Previously investigated IA approaches use the textual similarity between the bug description and the textual information related to source code entities from macro-evolution repositories. In our conducted research we investigate the task of IA by using micro-evolution repositories. Figure 2.1 shows a schematic diagram of the presented approaches. Block number 1 is related to the task IA. Approaches *InComIA* in chapter 3 and *RevIA* in chapter 4 are our presented approaches for task IA.

IA recommends the developer a set of relevant source code entities to the incoming change request. Now developers need to know which of their team-mates have more expertise on those relevant source code entities and hence are most likely to best assist with its resolution. The task of automatically assigning issues or change requests to the developer(s) who are most likely to resolve them is called bug triaging or developer recommendation (DR). In this dissertation we use techniques to find expertise score of developers for each source code entities relevant to the incoming change requests. These techniques uses the information from micro-evolution repositories. Block number 3 in Figure 2.1 is related to the task DR. Two approaches *iHDev* and *rDevX* in chapters 5 and 6 are our presented automated solutions for task DR.

We provided developers with the part of source code that needs to be changed and who have the most expertise to implement the change. Once the identified developer addresses the incoming change request, she submits the patch (set of modified source code files to fix a bug or add a new feature) to the code review repository (blocks number 2 and 5 in Figure 2.1). Reviewers then inspect the change through the code review tool (e.g.,*Gerrit*) and provide feedback to the developers

in the form of review comments. The developer may modify the source code based on the review comments received and submit the updated patch for review. Once the submitted patch meets the desired code quality, a reviewer "signs-off" on the review to be checked into the code repository (block number 5 in Figure 2.1). It is not always easy to determine who has the most expertise given a particular change for review, especially for newcomers to a codebase or those changing parts of the code with shared ownership by many people. The task of automatically assigning reviewers to code change who are most likely to review them is called code reviewer recommendation (CR)(block number 4 in Figure 2.1). We present approach *cHRev* in chapter 7 which uses the previous review activity of developers from micro-evolution repositories to recommend the set of expert reviewers to review a code change.

We assist developers with providing the set of expert reviewers to review the submitted patch by the developers. Determining whether a submitted patch will be accepted or not by the expert reviewers could improve the efficiency of the peer-review process, e.g., help developers and reviewers prioritize and focus their efforts. In this dissertation we present a predictive model in chapter 8 which extracts the relevant features from code review data and classifies whether a patch will be accepted or not as soon as it is submitted.

## 2.2 Micro-Evolution Repositories

In this work, we present automated solutions for DR, IA, and CR tasks that are centered on events from micro-evolution repositories. The term micro-evolution repositories is used to refer to all the repositories that store the data generated from developer activities between the start and end points of a change request and its resolution. Broadly speaking, micro-evolution repositories store data about all the "in-between" activities. The prominent instances of micro-repositories are the ones that track and mange developer activities within IDEs (interaction activity) and code review tools (code review activity).

### 2.2.1 Interaction Histories from IDEs

Interaction is the activity of programmers in an IDE during a development session (e.g., editing a file or referencing an API documentation). Tools, such as *Mylyn*, have been developed to model programmers' actions in IDEs [74]. *Mylyn* [1] monitors programmers' activities inside the *Eclipse IDE* and uses the data to create an *Eclipse* user interface focused around a task. The *Mylyn* interaction consists of traces of interaction histories. Each historical record encapsulates a set of interaction events needed to complete a task (e.g., a bug fix). Once a task is defined and activated, the *Mylyn* monitor records all the interaction events (the smallest unit of interaction within an IDE) for the active task. A trace file contains the interaction history of a task. This file is typically attached to the project's issue tracking system (e.g., Bugzilla or JIRA). For each interaction, the monitor captures about eleven different types of data attributes. The most important of these is the structure handle attribute, which contains a unique identifier for the target element affected by the interaction. For example, the identifier of a Java class contains the names of the package, the file to which the class belongs to, and the class. Similarly, the identifier of a Java method contains the names of the package, the file and the class the method belongs to, the method name, and the parameter type(s) of the method. Figure 2.2 shows an example of the *Mylyn* interaction edit event. The interaction history captures in-between activities (e.g., editing a source code file) of developers during a development session.

### 2.2.2 Code Review Histories

. Code-review repositories archive all the data resulting from all the review activities throughout the lifetime of a software system. In our conducted research we used the code review data captured by Modern Code Review (MCR) tools. In recent years, many modern software organizations have adopted MCR, which is based on collaborative tools. MCR tools that tightly integrate with version control system (e.g., *git*) repositories provide a lightweight code review environment to manage reviewing processes that are similar to the formal software inspection, while allowing for asynchronous collaboration during code reviews. Google-based *Gerrit* is one example

[1] https://www.eclipse.org/mylyn/

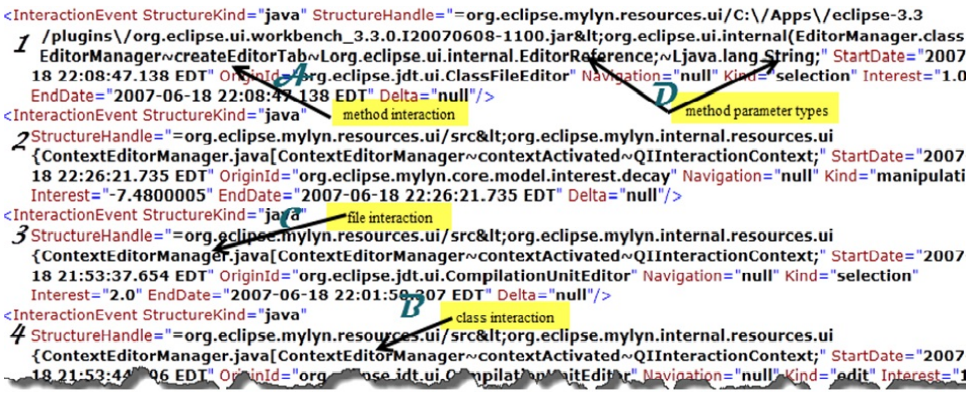


Figure 2.2: A snippet of an interaction event recorded by *Mylyn* for bug issue #330695 with trace ID #221752. File TaskListView.java is edited.

of tool supported peer review. A *Gerrit* review begins when the owner posts a patch to be reviewed. Reviewers are manually assigned, so that they can take part in the reviewing of the patch uploaded by the owner. Unassigned reviewers can also make comments. Reviewers can provide comments on individual lines that have changed or they can provide general comments. Reviewers can approve or reject the uploaded patch. A patch set encapsulates details regarding the author, committer, and the inline comments made by the reviewers [73]. Modern Code Review (MCR) [13], which is tool assisted, supports the quality-control process of reviewing a proposed code change or patch for a maintenance task. It includes human feedback, which drives whether a proposed code change needs to be revised, and eventually accepted or abandoned. Only accepted source code changes are merged to the code base. That is, it is one of the sources that captures *the means* adopted in reviewing and revising not only the patches that eventually get merged (i.e., committed), but also those abandoned. The code review history captures the in-between activities (e.g., providing the review feedback) of developers during a review session.

**Patch Lifecycle in *Gerrit***

*Gerrit* is a modern peer-review tool that facilitates a traceable review process for *git*-based software projects [1]. Developers make local changes in their private *git* repositories and then submit these changes as a patch for review [82]. The owner may indicate the intended reviewers, who are subsequently notified about the review invitation. Alternatively, the owner can broadcast

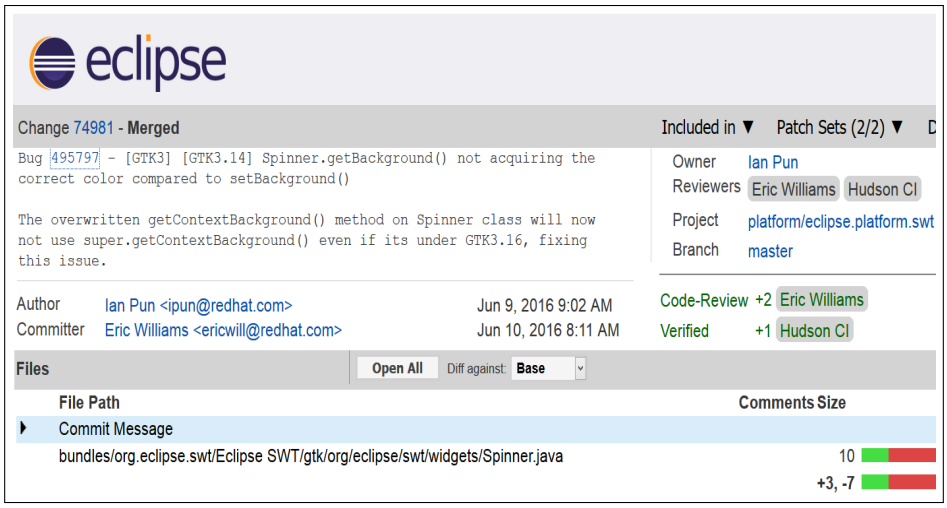


Figure 2.3: An example from *Gerrit* code review# 74981 related to *Eclipse Platform* project

a request for review to find reviewers for their patch. It should be noted that the invited reviewers do not necessarily accept the invitation and contribute to the review. The approval step requires human contributions (e.g., to ascertain the patch meets project guidelines and intent), whereas, the verification step is largely automatic (e.g., the build works and complies, and passes unit test cases). If it fails either of these steps, it is abandoned. Figure 2.3 shows an example from *Eclipse Platform Gerrit* for patch review # 74981. It was accepted and merged to the projects's *git* repository. This patch is for bug # 495797 which was assigned to *Ian Pun* for a fix and he patched it. The number of line changes is 10 and the number of changed files is 1. *Eric Williams* reviewed and accepted this patch.

Figure 2.4 shows the patch life cycle in *Gerrit*. Initially, a developer (the patch owner) makes changes to the source code in response to a bug report or feature request. They submit these code changes for review. The patch owner may indicate the intended reviewers (assigned reviewers), who are subsequently notified about the review invitation. It should be noted that the invited reviewers do not necessarily accept the invitation and contribute to the review. Contributing reviewers then inspect the change through the code review tool *Gerrit* and provide feedback in the form of review comments and scores to the owner. A submitted patch can have different statuses such as *Needs Code Review, Ready to Submit, Abandoned, and Merged*. For a patch to get accepted

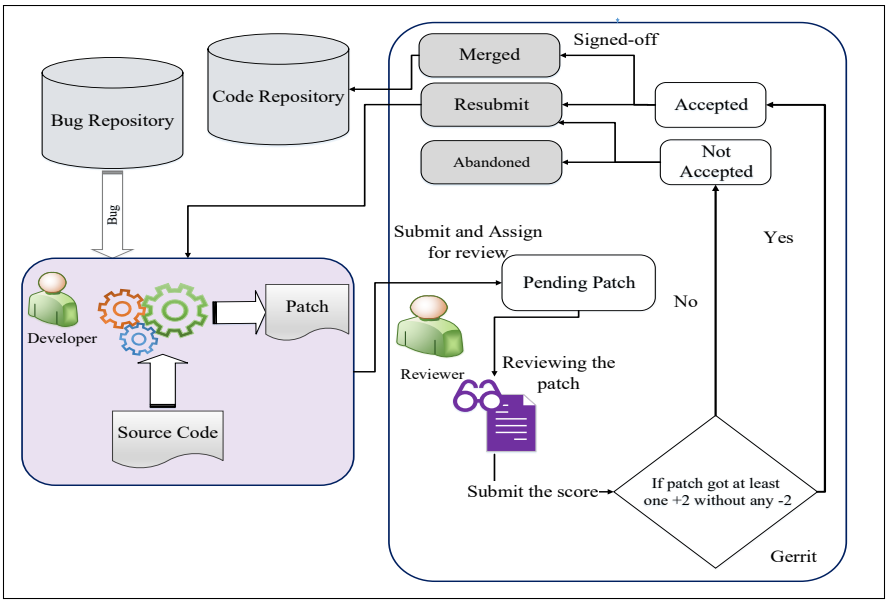


Figure 2.4: The patch life cycle in *Gerrit*

and finally merged to repository, it is necessary that it receive at least one +2 and not a single -2 score.

If a patch receives a score of -2, its status will be marked abandoned. If patch has not received a score -2, then the patch owner can rework the patch and resubmit the new version of patch. The review process for a particular submission may include multiple iterations between the patch owner and contributing reviewers. Eventually, a reviewer signs-off on the review after they perceive the code change to be of a sufficient quality and should be checked into the code repository. If a change never received sign-offs, it is abandoned. The number of sign-offs required to check in a code change is typically dependent on a particular team's policy. Automatic tools typically perform the verification part.

## 2.3 Macro-Evolution Repositories

Macro-evolution repositories include traditional sources such as the information stored in source code version-control systems (e.g., Subversion), requirements/bug-tracking systems (e.g., Bugzilla), and communication archives (e.g., email). Source-code control systems are used for storing and managing changes to source code artifacts. We term the repositories that store and

mange data about the end points as macro-evolution repositories. Typical examples of macro-evolution repositories are bug tracking systems (e.g., start of a change) and source-control systems (end of a change). The macro-evolution repositories typically mark the culmination of a major evolution task, e.g., a report of a bug or code commit.

## 2.4 Related Work

In this section, we survey the related research on interaction history, code review, and various software maintenance tasks such as IA, DR, CR, and PAR. More specifically, we describe how the related works motivate our presented approaches.

### 2.4.1 Interaction History

Researchers have been developing IDE plug-ins to capture programmers' interactions during programming activities [37, 93, 88]. NavTracks was used to mine Interaction coupling at the file-level granularity [93]. Team Track [37] was used to provide navigation support to programmers unfamiliar with the code base. In HeatMaps [88], the interestingness of a programming element is determined by computing a Degree-of-Interest (DOI) value based on the historical selection and modification of it. If an artifact is found interesting, it is decorated with colors to indicate its importance to the task.

Similarly, there are studies which present the applications of the captured interaction data for software maintenance tasks, for example, Zou et al. [115] used the interaction history to identify evolutionary information about a development process (such as restructuring is more costly than other maintenance activities). Rastkar et al. [79] report on an investigation which considered how bug reports considered similar based on change set information compare to bug reports considered similar based on interaction information. Robbes et al. [85] developed an incremental change based repository by retrieving the program information from an IDE (which includes more information about the evolution of a system than traditional SCM) to identify a refactoring event. Parnin and Gorg [76] identified relevant methods for the current task by using programmers' interactions with an IDE. Kobayashi et al. [59] presented a Change Guide Graph (CGC) based on

interaction information to guide programmers to the location of the next change. Each node in the graph presents a changed artifact and each edge presents a relation between consecutive changes. Following the CGC graph, the next target in the change sequence can be identified. Schneider et al. [90] presented a visual tool for mining local interaction histories to help address some of the awareness problems experienced in distributed software development projects. Both interaction history and static dependencies were used to provide a set of potentially interesting elements to a programming task.

These studies motivated us to investigate the usefulness of interaction data to provide the solutions for different software maintenance tasks such as IA and DR.

### 2.4.2 Code Review

There has been much investigation in the various aspects of modern code review. We briefly discuss a few representative efforts from this investigation.

Previous studies in code review area can be classified according to several empirical studies describing the different features of modern code review process, predicting the outcome of code review, influence of code review on the code quality, optimizing the effort of reviewers, and several tools which support the code review process. We focus our discussion of related code review research to work that considers code reviewers as a primary subject.

**Empirical studies on code review:** While a very rigid Fagan code inspection process may have been appropriate the mid-70s, a significant amount of time and effort is required to collate review material, and coordinate its distribution and review [38]. In contrast contemporary or modern code review encompasses a series of less rigid practices [81, 60, 32]. These lightweight practices allow peer review to be adopted to fit the needs of the development team. Peer code review, a manual inspection of source code by developers other than the author is recognized as a valuable tool for reducing software defects and improving the quality of software projects. Peer review is seen as an important quality assurance mechanism in both industrial development and open source software (OSS) community. Rigby et al. [83] examined two peer review techniques: review-then-commit and commit-then-review used by Apache server project. The frequency of reviews, the level of

participation in reviews, and the size of artifacts under review are a few factors that they have measured in their studies. Modern code review often leave out the team meeting and reduce the number of people involved in the review process to two. Wood et al. [104] found that the optimal number of reviewers should be two. Rigby et al. [82] compared three types of peer review methods: traditional inspection, OSS email-based peer review, and lightweight tool supported review. Despite differences among projects, many of the characteristics of the review process have independently converged to similar values. which indicates general principles of code review practice.

Software peer review has proven to be a successful technique in open source software development. In contrast to industry, where reviews are typically assigned to specific individuals, changes are broadcast to hundreds of potentially interested stakeholders. Rigby et al. [84] describe an empirical study to investigate the mechanisms and behaviors that developers use to find code changes they are competent to review. Bacchelli et al. [13] conducted an empirical study across diverse teams at Microsoft to empirically explore the motivations, challenges and outcome of tool-based code reviews. Baysal et al. [18] studied the patch lifecycle of the Mozilla Firefox project. Their study shows that patches from casual developers should receive extra care to ensure software quality and encourage future contributions. Porter et al. [77] studied effect of the variance among elements of the software inspection process, such as team size and the number and sequencing of session, on the inspection effectiveness.

**Influence of code review on the code quality:** There are several studies investigating the effect of code review on code quality. McIntosh et al. [69, 72] studied the effect of code reviews on code quality by mining the code review and change repositories of open source projects. They report that the percentage of reviewed changes a code component underwent correlates inversely to its chance of being involved in post-release fixes. Beller et al. [21] studied the types of defects fixed in modern code review repositories. Kemerer et al. [55] show that code review reduce the amount of defects in student projects. With the available data they were also able to study the impact of review rate on the inspection performance. They found high review rates (i.e., a high number of reviewed LOC/hour) to be associated with a decrease in inspections effectiveness.

**Tool support for code review:** There are studies presenting the developed tools that support the code review process. For example, Kim et al.[56] developed a tool called *LSdiff* to help reviewers inspect program differences. *LSdiff* covers the limitations of program difference tools by inferring systematic structural differences into logic rules. Zhang et al. [114] developed a tool called CRITICS, an Eclipse plug-in that assists developers in inspecting systematic changes. There are several researched tools to help support other aspects of modern code review. Modern code review is often supported by tools, preferably integrated into the development environment (IDE) [31]. One of these integrated IDE tools is ReviewClipse [22] and another is Mylyn Reviews [3]. A popular review tool is OSS Gerrit [1], offering web-based reviewing for projects using Git [70]. A number of other review tools: CodeFlow [13], Microsoft code review tool; Phabricator is Facebook's open-sourced tool [4]; Mondrian, a tool that Google uses for its closed-source projects [2]. With the advent of open source code review tools such as Gerrit along with projects that use them, code review data is now available for collection and analysis. Mukadam et al. [73] extracted Android peer review data from Gerrit and provide the data for future empirical software engineering questions. González et al. [42] presented an approach to retrieve and analyze the information produced by Gerrit based on a previously designed tool called Bicho.

None of the previous studies for Modern Code Review has investigated the application of code review for tasks IA and DR.

### 2.4.3 Software Change Impact Analysis

Information Retrieval (IR) methods were proposed and used successfully to address tasks of extracting and analyzing textual information in software artifacts, including change impact analysis in source code [30, 51, 78]. Existing approaches to IA using IR operate at two levels of abstraction: change request [30] and source code [51, 78]. In the first case, the technique relies on mining and indexing the history of change requests (e.g., bug reports). In particular, this IA method utilizes IR to link an incoming change request description to similar past change requests and the file revisions that were modified to address them [30, 101]. While this technique has been shown to be relatively robust in certain settings, it is entirely dependent on the history of prior change requests.

In cases where textually similar change requests cannot be identified (or simply do not exist), the technique may not be able to identify relevant impact sets. Also, these works show that a sizeable change request history must exist to make this approach operational in practice, which may limit the effectiveness. The other set of techniques to IA that use IR operates at the source code level and requires a starting point (e.g., a source code method that is likely to be modified in response to an incoming change request) [51, 78]. This approach is based on the hypothesis that modules (or classes) in software systems are related in multiple ways. The evident and most explored set of relationships is based on data and control dependencies; however, the classes can be also related conceptually (or textually), as they may contribute to the implementation of similar domain concepts. In our previous work [51, 41], given a textual change request (e.g., a bug report), a single snapshot (i.e., release) of source code, indexed using Latent semantic indexing, is used to estimate the impact set and to derive conceptual couplings from the source code. Machine learning techniques and information retrieval are used in different areas such as bug triaging, impact analysis, feature location, bug severity and classifying software changes [97, 58, 99]. Different machine learning techniques such as ML-KNN, SVM, LSI or decision tree are also used for recommending developers to resolve a reported bug request [105, 106, 66].

Although ample progress has been made, there remains quite a bit work left in improving the effectiveness of IA. The previous studies either used information from bug repository or source code repository. None of these approaches considered interaction activity of developers as source of information. Developers may interact (e.g., navigate, view, and modify) entities within an Integrated Development Environment (IDE) that may not be eventually committed to the code repository. These interactions could have contributed in locating and/or verifying the entities that were changed due to a change request, which potentially makes them candidates for future changes. Code review is another in-between activity of developers in software maintenance process. Developers provide feedback in the form of review comments. These review comments includes textual information related to both source code and bug itself. This textual information holds a wide range

of information that is instrumental in finding a solution for IA. None of the previous approaches considered code review comments provided by developers as source of information for task IA.

*We present a new approach, namely InComIA, for IA that is centered on the developer interaction and commit histories of source code entities that were involved in the resolution of previous change requests. Similarly we present another approach called RevIA which includes the code review comments written by developers during the review phase as an additional textual information. Several empirical studies are conducted to assess the effectiveness of the InComIA and RevIA approaches.*

### 2.4.4 Developer Recommendation

The task of automatically assigning issues or change requests (e.g., bug fixes or new feature) to the developer(s) who are most likely to resolve them has been studied under the umbrella of issue triaging. A number of approaches exist in the literature [9, 107, 47, 7, 35, 16, 8, 10, 111]. Anvik and Murphy [7] conducted an empirical evaluation of two techniques for identifying expert developers. Developers acquire expertise as they work on specific parts of a system. Expertise Browser (ExB) is a tool to identify developers with the desired expertise [71]. Tamrawi et al. [94] used fuzzy-sets to model bug-fixing expertise of developers based on the hypothesis that developers who recently fixed bugs are likely to fix them in the near future. An approach based on a machine learning technique is used to automatically assign a bug report to a developer [9]. Another approach to facilitate bug triaging uses a graph model based on Markov chains, which captures the bug reassignment history [47]. Matter et al. [68] used the similarity of textual terms between a given bug report of interest and source code changes (e.g., word frequencies of the *diff* given changes from source code repositories). Fritz et al. [39] introduced the degree-of-knowledge model that computes a real value for each source code element based on both authorship and interaction information. Their expertise model was applied to, and evaluated on, the developer recommendation problem. Bird et al. [25] analyzed the communication and co-ordination activities of the participants by mining email archives. Ma et al. [67] proposed a technique that uses implementation expertise (i.e., developers usage of API methods) to identify developers. Weissgerber et al. [102] depicts the relationship between the lifetime of the project and the number of files each author updates by analyzing and visualizing the check-in information for open source projects. German

[40] provided a visualization to show which developers tend to modify certain files by studying the modification records (MRs) of CVS logs. Fischer et al. Bortis et al. [27] introduced PorchLight, a tag-based interface and customized query expression, to offer triagers the ability to explore, work with, and assign bugs in groups. Shokripour et al. [92] proposed an approach for bug report assignment based on the predicted location (in the source code) of the bug. Begel et al. [20] conducted a survey of inter-team coordination needs and presented a flexible *Codebook* framework that can address most of those needs. Robbes et al. [87] used the interaction data from *Mylyn* to evaluate developer-expertise metrics based on time, which is the start time of a *Mylyn*-interaction session.

Approaches for developer recommendation typically operate on software repositories (e.g., models trained from past bugs/issues or source-code changes), the source-code authorship, or their combinations under the rationale that if a developer frequently commits changes to, or fixes bugs in, particular parts of a system, they have knowledge of that area and can competently make changes to it. We agree with the fundamental concept that historical records of developers' activity yield insight into their knowledge and ability, but we also posit that additional traces of developer activity such as interactions with source code (beyond those leading to commits) when resolving an issue can provide a more comprehensive picture of their expertise. Similarly, the historical code reviews capture many unique aspects that can be the useful markers of developer expertise. Among them is the reviewer role and contributions in terms of the code critique and formative experience in authoring code changes, which may or may not have been successful (i.e., eventually included in the project code base). Code reviews subsume the commits in source-code repositories (e.g., git or Subversion). We posit that these markers provide a unique opportunity to expand the scope of forming models of developer expertise in source code and their improved applications in software maintenance and evolution tasks. The records of developers' interactions with code in resolving an issue and code review activity of developers remain largely untapped in solving this problem.

*We present a new approach, namely iHDev, for assigning the incoming change requests to appropriate developers. iHDev is centered on the developers' interactions with source code entities that were involved in the resolution of previous change requests. Similarly, we present another approach, namely rDevX for assigning the incoming change requests to appropriate developers based on past code review activities of developers. Several empirical studies are conducted to assess the effectiveness of the presented approaches.*

### 2.4.5 Reviewer Recommendation

The reviewer recommendation task has not been examined much in the literature yet. There are only three approaches reported for reviewer recommendations. Balachandran [14] proposed a GIT blame like line oriented approach. Recently, Thongtanunam et al. [96] proposed an approach, namely *REVFINDER*, which is based on the past reviews of files with similar names and paths. They showed that *REVFINDER* outperformed Balachandran's approach on open source systems. *REVFINDER* finds past reviews with files whose paths and names are similar (based on string comparison) to the ones in the patch under review. It assigns all the reviewers from each such past review the same (string comparison) score. All the reviewers are ranked based on the sum of their scores. It does not look into other attributes of the past contributions of the reviews (e.g., how much and when) and is limited to whether a reviewer contributed or not. In summary, *REVFINDER*'s expertise model favors *breadth or generality* of review contributions. In parallel to our work, Xia et al. [98] proposed another approach for reviewer recommendation, namely TIE. The intuition of TIE is that the same reviewers are likely to review changes containing similar terms (words) and reviewers are likely to review changes to the same files or files in similar locations. TIE outperformed *REVFINDER* on open source systems. Similar to *REVFINDER*, TIE approach just look into the similarity of patch description and file path.

approaches for reviewer recommendation either need textual information from code changes or do not account for attributes such as the amount of comments and their recency. None of the previous approaches provides empirical evaluation included the closed-source domain.

*We present an approach, cHRev, to automatically recommend reviewers who are best suited to participate in a given review, based on their historical contributions as demonstrated in their prior reviews. We evaluate the effectiveness of cHRev on three open source systems as well as a commercial codebase at Microsoft.*

### 2.4.6 Predicting the outcome of code review

There are a few studies that evaluate the influence of different factors on the output of code review (e.g., code review response time and outcome). Baysal et al. [17] described an empirical study of code review process for WEbKit and the reported results provide that non-technical factors such as bug priority and patch writer experience can have a significant impact on the code review outcome. Weissgerber et al. [103] performed data mining on email archives of two open source projects to study the patch contributions. They found that smaller patches have a higher chance of being accepted than the larger ones. Jiang et al. [49] found that patch acceptance is affected by the developer experience, patch maturity, and prior subsystem churn, whereas, the reviewing time is impacted by the submission time, the number of affected subsystems, the number of suggested reviewers and the developer experience. Jeong et al. [48] examined the review process for two Mozilla projects along with the approaches to recommend reviewers. Then they presented predictors to predict the outcome of code review process done by the recommended reviewers.

There has been a few previous efforts in addressing the question, what makes patches get accepted or not? ; however, they have not been in the context of Modern Code Review [17], [50], [44], [19]. Also, the predictive recommendation and assessment of whether a patch will be eventually accepted or not as soon as it is submitted for MCR were not investigated previously.

*We first investigate the factors that influence the patch acceptance. We take into account several direct and indirect measures, which are derived from the structured and unstructured data available from bug tracking and code-review repositories. These measures are then used formulate a descriptive model. Secondly, we build a classifier to predict whether a patch will be accepted or not as soon as it is submitted for MCR process.*

# CHAPTER 3

## Change Impact Analysis (*InComIA*)

This chapter presents an approach to perform impact analysis (IA) of an incoming change request on source code [112]. The approach is based on a combination of interaction (e.g., *Mylyn*) and commit (e.g., CVS) histories. As we mentioned in Chapter 2, interaction activity of developer is an example of micro-events. The source code entities (i.e., files and methods) that were interacted or changed in the resolution of past change requests (e.g., bug fixes) were used. Information retrieval, machine learning, and lightweight source code analysis techniques were employed to form a corpus from these source code entities. Additionally, the corpus was augmented with the textual descriptions of the previously resolved change requests and their associated commit messages. Given a textual description of a change request, this corpus is queried to obtain a ranked list of relevant source code entities that are most likely change prone. Such an approach that combines information from interactions and commits for IA at the change request level was not previously investigated. Furthermore, the approach requires only the entities that were interacted and/or committed in the past, which differs from the previous solutions that require indexing of a complete snapshot (e.g., a release).

An empirical study on 3272 interactions and 5093 commits from *Mylyn*, an open source task management tool, was conducted. The results show that the combined approach outperforms an individual approach based on commits. Moreover, it also outperformed an approach based on indexing a single, complete snapshot of a software system.

### 3.1 Introduction

Software-change impact analysis, or simply impact analysis (IA), is an important activity during the software maintenance and evolution phase. IA aims at estimating the potentially impacted entities of a system due to a proposed change [11]. The applications of IA include cost

estimation, resource planning, testing, change propagation, managing ripple effects, and traceability [78, 29, 62, 75, 80].

Change requests are typically specified in natural language (e.g., English). They include bug reports submitted by programmers or end users during the post-delivery maintenance of a product and managed with issue tracking systems (e.g., Bugzilla). One variant of IA that is widely investigated in the literature is to identify the potential source code entities that are change prone on account of a given change request. A number of approaches to address this task have been presented in the literature for performing IA [41]. They range from traditional static and dynamic analysis techniques to modern methods based on Information Retrieval (IR) and Mining Software Repositories (MSR). For example, an IR technique is used to index all the source code entities in a single snapshot and then use the incoming change request to query for the top relevant source code entities. MSR techniques use the historic information from bug and source code repositories to predict the change prone source code. Although ample progress has been made, there remains quite a bit work left in improving the effectiveness of IA.

We present a new approach, namely *InComIA*, for IA that is centered on the developer interaction and commit histories of source code entities that were involved in the resolution of previous change requests. Developers may interact (e.g., navigate, view, and modify) entities within an Integrated Development Environment (IDE) that may not be eventually committed to the code repository. These interactions could have contributed in locating and/or verifying the entities that were changed due to a change request, which potentially makes them candidates for future changes. Tools such as *Mylyn* capture and store such interaction histories. Our approach combines the source code entities (i.e., files and methods) that were interacted or changed in the resolution of past change requests (e.g., bug fixes). Information retrieval, machine learning, and lightweight source code analysis techniques are then used to form a corpus from these source code entities. Additionally, the corpus is augmented with the textual descriptions of the previously resolved change requests and their associated commit messages. Given a textual description of a change request, this corpus is queried to obtain a ranked list of relevant source code entities that are most likely

change prone (i.e., the estimated impact set). In our previous work [15], we used combinations of interaction and commit histories for source code to source code IA. In this chapter, we investigate IA on the onset of an incoming change request to source code.

To evaluate the accuracy of our technique, we conducted an empirical study on the open source system *Mylyn*. Precision and recall metric values on a number of bug reports sampled from this system for IA are presented. That is, how effective our approach is at recommending the actual source code entities (files and methods) that are changed to fix these bugs. Additionally, we empirically compared our approach to those using commits and the entire source code from a single snapsot. The results show that the presented approach outperformed the baseline competitors: Lowest recall gains of 28% and 44%, highest recall gains of 225% and 350%, lowest precision gains of 28% and 33%, and highest precision gains of 250% and 350% were recorded.

Our work makes the following noteworthy contributions in the context of performing IA on source code due to an incoming change request:

1. To the best of our knowledge, our approach is the first to integrate the developer interaction and commit histories of source code.
2. We performed a comparative study with an approach that makes an exclusive use of source-code commits.
3. We performed a comparative study with an approach that uses the source code in a single snapshot.

The rest of the chapter is organized as follows: Our *InComIA* approach is discussed in Section 3.2. The empirical study on *Mylyn* open-source projects and the results are presented in Section 3.3. Threats to validity are listed and analyzed in Section 3.4.

## 3.2 The *InComIA* Approach

Our *InComIA* approach to IA of an incoming change request on source code consists of the following steps:

- The past change requests (e.g., bug reports) from software repositories are analyzed, and all the source code entities that were interacted and committed are extracted.
- The source code of each unique entity, from each revision (i.e., in subversion vocabulary) in which it was interacted or committed in the past, is parsed using a developer-defined granularity level (e.g., file and method). For each entity, comments, identifiers and expressions are then extracted from the parsed source code. For each entity all of its commit messages and bug descriptions are also obtained. A corpus is created such that each source code entity has a corresponding document (i.e., in IR vocabulary) in it.
- Given a change request description, a ranked list of relevant source code entities (e.g., files and methods) is recommended for IA based on the K-Nearest Neighbor algorithm and Cosine similarity.

Before, we present the details of these steps, we briefly discuss the two principal sources of information of *InComIA*.

### 3.2.1 Interactions and Commits for Change Request Resolution

Interaction is the activity of programmers in an IDE during a development session (e.g., editing a file, or referencing an API documentation). Different tools (such as *Mylyn*) have been developed to model programmers' actions in IDEs [74, 93, 90, 88]. *Mylyn*[1] monitors programmers' activities inside the Eclipse IDE and uses the data to create an Eclipse user interface focused around a task.

The *Mylyn* interaction consists of traces of interaction histories. Each historical record encapsulates a set of interaction events needed to complete a task. Once a task is defined and activated, the *Mylyn* monitor records all the interaction events (the smallest unit of interaction within an IDE) for the active task. For each interaction, the monitor captures about eight different types of data attributes. The structure handle attribute contains a unique identifier for the target element affected by the interaction. For example, the identifier of a Java class contains the names of the package, the

[1] https://www.eclipse.org/mylyn/

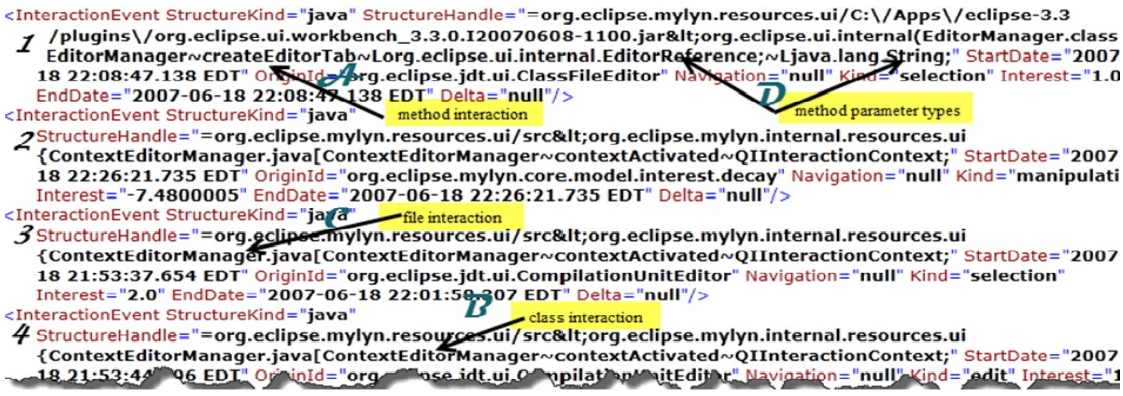


Figure 3.1: A snippet of 4 interaction events (labelled 1-4) recorded by *Mylyn* for bug issue #175229 with trace ID #71687.

file to which the class belongs to and the class. Similarly, the identifier of a Java method contains the names of the package, the file and the class the method belongs to, the method name and the parameter type(s) of the method. Figure 3.1 shows an example of 4 consecutive *Mylyn* interaction events. For each active task, *Mylyn* creates an XML trace file called *Mylyn*-context.zip. In the 1st interaction, the createEditorTab method is selected. In the 2nd, 3rd and 4th interactions, the contextActivated method is indirectly manipulated, then directly selected and finally edited. A) Method name: createEditorTab; B) Class name: ContextEditManager; C) File name: ContextEditorManager.java; D) Parameter types for createEditorTab method: Lorg.eclipse.ui.internal.EditorReference and Ljava.lang.String.

A trace file contains the interaction history of a task. This file is typically attached to the project's issue tracking system (e.g., Bugzilla or JIRA). The trace files for the *Mylyn* project are archived in the Eclipse bug tracking system as attachments to a bug report[2].

A common practice in the open source software development is for developers to include an explicit bug or issue ID in the commit message. The presence of this information establishes the traceability between an issue or bug reported in the bug tracking system and the specific commit(s) performed to address it. A regular-expression based method can be employed to process commits and extract this traceability information. Figure 3.2 shows the bug description for bug ID #175229,

[2]https://bugs.eclipse.org/bugs/query.cgi

<bug>
<bug_id>175229</bug_id>
<creation_ts>2007-02-23 03:34:00 </creation_ts>
<short_desc>
Should be able to open editor automatically when a task is activated
</short_desc>
</bug>
.....
<logentry revision="5113">
<author>mkersten </author>
<date>2007-06-19T02:27:07 </date>
<paths>
.....
</paths>
<msg>RESOLVED - bug 175229: Should be able to open editor automatically when a task is activated.
$https://bugs.eclipse.org/bugs/show_bug.cgi?id = 175229$
</msg>

Figure 3.2: Bug ID #175229 from Eclipse bug tracking and commit history.

which is extracted from the Eclipse bug tracking system, and the commit message related to this bug ID, which is for revision #5113. The files that were included in this commit are listed between the *paths* tags. Now, we describe the details of our *InComIA* approach.

### 3.2.2 Extracting Interacted Entities

We first need to identify bug reports that contain mylyn-context.zip attachment(s) because all bug issues may not contain interaction trace(s). To do so, we searched the Eclipse bug-tracking system for bugs containing at least one mylyn-context.zip attachment. Another factor to consider is that all of the interactions to a system may not result in committed changes to a source control system. If a bug issue is not fixed with a resolution, it is unlikely for a corresponding commit history to exist. Thus, we only searched for bug issues with a "Resolved" status and a "Fixed" resolution. We developed a tool to process the search result and performs the major following tasks:

`Downloading trace files`: The tool takes the search result from the Eclipse bug-tracking site as input and automatically downloads all the trace files to a user specified directory. The trace

files have the same name, mylyn-context.zip. The tool renames each file by using the bug ID and attachment ID (separated by an underscore) giving them a unique identifier in the directory they reside in. Internally, the tool identifies the trace file ID(s) for each bug issue. If options are specified to output this result, the tool can save the bug IDs with the corresponding trace IDs in a Java properties file format, the key is the bug id and the values is a comma separated list of trace IDs. It uses the URL pattern https://bugs.eclipse.org/bugs/attachment.cgi?id=X to download each trace file by replacing X with the trace ID.

`Processing trace files`: The tool takes the directory that contains the trace files as input and parses each trace file to identify the list of Java files and methods manipulated by each interaction history. We consider each trace file as an interaction transaction. For each transaction, the tool outputs the issue number together with a tab-separated list of Java files and methods. We need the issue number to create a link between interaction and commit transactions. The targeted files and methods are identified from the structure handle of the interaction event. Figure 3.1 (A and C) shows a method name and file name from event 1 and 3 respectively. Mylyn can create different types of interaction events (e.g., edit and selection) on the same target in a single interaction history. We consider only the first interaction to an element regardless of the type of interaction event. For further details on this step, please refer to our previous work [15]. We refer to the set of Interacted entities as set $I = \{(i, E_{im})\}$ where $i$ is the revision number and $E_{im}$ is the $m^{th}$ entity $E$ which was interacted in revision $i$.

### 3.2.3 Extracting Committed Entities

Our approach also requires commit data from version archives, such as SVN and CVS, because we need files that have been changed together in a single commit operation. SVN preserves atomicity of commit operations; however, older versions of CVS did not. Subversion assigns a new ”revision” number to the entire repository structure after each commit. For a project hosted in an ”older” CVS repository, we convert the CVS repository into an SVN repository using the CVS2SVN tool, which has been used in popular projects such as gcc3. Our tool mines file-level commit transactions from the SVN repository. For mining method-level transactions, we used a

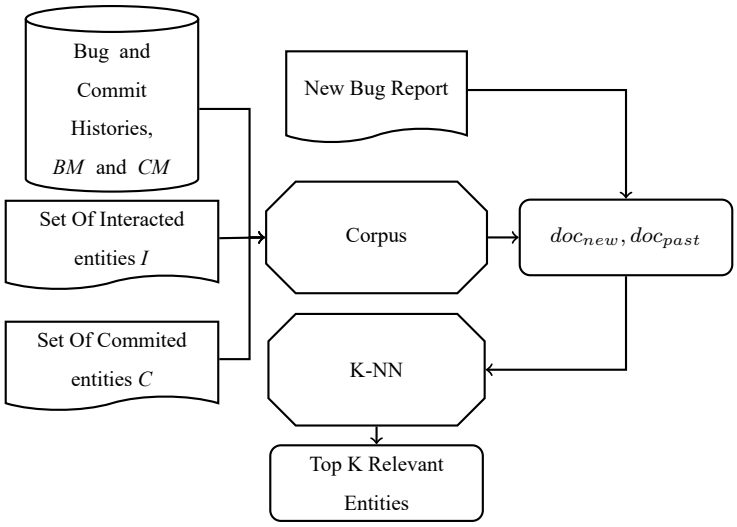


Figure 3.3: A Schematic diagram of *InComIA*.

previously developed tool with some modification to identify the issue number associated with each commit. We refer to the set of committed entities as set $C = \{(i, E_{in})\}$ where $i$ is the revision number and $E_{in}$ is the $n^{th}$ entity $E$ that was committed in the revision $i$. Figure 3.3 shows the overall structure of our *InComIA* approach. This section contributes to the left three blocks of Figure 3.3.

### 3.2.4 Obtaining Source Code of Entities from Interacted and Committed Revisions

Our *InComIA* approach needs the source code of all of the entities from all of the revisions in which they were interacted or committed previously in resolving change requests. The extraction step described above only gives the list of entity names and revisions in which they were involved, but not the source code. It is relatively straightforward to determine the source code revision of an entity in which it was committed; however, the precise revision number that was interacted is not available from the interaction history. Therefore, we consider that the source code in the $n - 1^{th}$ revision of an entity should have been interacted if the same entity was committed in the $n^{th}$ revision for an issue.

Source code from each of the revisions in which an entity was interacted is obtained. Similarly, source code from each of the revisions in which an entity was committed is obtained. At the

end of this step, we have all the needed source code of each interacted or committed source code entity. We use these source code files to form a corpus.

### 3.2.5 Creating a Corpus from Source Code and Textual Descriptions

**Extracting Comments, Identifiers and Expressions**

The source code files from each interacted or committed revision are converted to the srcML representation [3]. This conversion is done for the ease of extraction of identifiers, comments and expressions from the source code. All the comments are extracted from all of the srcML files with an XML query approach. We consider two types of comments in a source code file.

- Header comments that are generally the first comment in a source code file, or leading a source code class and/or source code method, and may contain useful keywords related to the specific role of the related class or method.
- Body comments that sometimes contain important description (e.g., related to a specific change that was done in a specific class or method).

For extracting identifier names, we use the CamelCase splitting technique for compound identifiers. For example, the identifier ”TaskHistoryTest” is split into ”Task”, ”History” and ”Test” and the identifier ”SSLCertificate” is split into ”SSL” and ”Certificate”. Similarly, we consider all of the expressions in a source code file, including all of the output messages, which may contain important keywords.

**Extracting Issue Descriptions**

We add the issue (i.e., bug) descriptions related to each revision that has files that have been interacted or committed to our corpus. Therefore, we need to have a list of all the issue IDs with their textual descriptions. We refer to the set of bug descriptions as set $BM = \{(i, B_{ik})\}$ where $i$ is the revision number and $B_{ik}$ is the $k^{th}$ bug description for revision $i$. It is possible that for one revision, more than one bug description exists.

[3] http://www.srcml.org/

```
public class TaskHistoryTest extends TestCase {
    .....
/**
 * Tests navigation to previous/next tasks that are chosen
 *  from a list rather than being sequentially navigated
 */
public void testArbitraryHistoryNavigation() {

    resetHistory();

    // Simulate activating the tasks by clicking rather
    //than navigating previous/next
    (new TaskActivateAction()).run(task1);
    history.addTask(task1);
```

```
<path
   action="M"
   kind="file">/trunk/.../tasklist/tests/TaskHistoryTest.java</path>
</paths>
<msg>Patch for Bug #110506: provide context for previous/next task
 actions and misc fixes
</msg>
```

Figure 3.4: A snippet of the file TaskHistoryTest.java

### Extracting Commit Messages

After converting the *Mylyn* CVS repository to an SVN repository, we simply parse the SVN log to find the commit message and bug ID related to each revision number. We refer to the set of commit messages as set $CM = \{(i, C_i)\}$ where $i$ is the revision number and $C_i$ is the commit message for revision $i$.

After this step, we have a document for each source code entity that was either interacted or committed in the corpus. The dimensions of these documents in the corpus are formed from all of the words extracted from processing the revisions for source code comments and identifiers, issue descriptions and commit messages.

Figure 3.4 shows that there is a strong text similarity between the header comment of file *TaskHistoryTest.java* "Test navigation to previous/next tasks.... ", the method body comment "....by clicking rather than navigating previous/next" and the commit message from the SVN log "provide context for previous/next task actions ....". After this step, all of the information needed from the left three blocks in Figure 3.3 is gathered and the initial corpus is formed.

### 3.2.6 Indexing and Querying the Corpus

In *InComIA*, we use techniques from Natural Language Processing in order to locate textually relevant files based on comments, identifiers, expressions and bug and commit descriptions. We have the two following documents: a new bug description for which we want to find all of the relevant files (which we will refer to as $doc_{new}$) and a corpus document made up of all of the information extracted from source code files plus bug descriptions and commit messages (which we will refer to as $doc_{past}$). Before transforming two documents to numeric vectors (document representation) we preprocess both of our documents.

**Preprocessing Step**

The preprocessing step includes two important parts:

`Removing Stop words`: Most common words in English (called stop words) are often removed from documents before any attempt to classify them is made.

`Stemming each word`: Stemming is the process of reducing inflected (or sometimes derived) words to their stem, base or root form. After removing the stop words, we stem each of the words in the bug description.

*nltk.stopwords* and *nltk.PorterStemmer* from the *Natural Language Toolkit* are used for these two steps[4].

**Document Presentation (Including Document Indexing and Term Weighting)**

To perform the process of document presentation we use *Gensim*[5] (topic modelling for humans) - a *Python* library.

`Document Indexing`: We produce a dictionary from all of the terms in our document and assign a unique integer ID to each term appearing in it.

`Term Weighting`: We use *tfidf* in our approach. The global importance of a term *i* is based on its inverse document frequency (*idf*). *idf* is the document frequency of term *i* in the whole document collection. $tf_{i,j}$ is the term frequency of a term *i* in a document *j*. Each document

[4]http://www.nltk.org
[5]http://radimrehurek.com/gensim/tut3.html

$d_j$ is represented as a vector $d_j = (w_{1,j}, ..., w_{i,j}, ..., w_{n,j})$ where $n$ is the total number of terms in our document collection and $w_{i,j}$ is the weight for term $i$ in document $j$.

**Dimensionality Reduction**

In *InComIA*, we use the singular value decomposition (SVD) of the feature document matrix representation of our dataset ($doc_{past}$) [91]. We transform our *tf-idf* corpus into a latent m-D space of a lower dimensionality via the *models.Lsimodel* function in the *Gensim* library. Thus, we have all the processing of corpus creation completed in the "corpus" block in Figure 3.3.

**K-Nearest-Neighbor**

The task of multi-label classification is to predict for each data instance, a set of labels that applies to it. Standard classification only assigns one label to each data instance. However, in many settings a data instance can be assigned by more than one label. For IA, each data instance (i.e., a bug report) can be assigned multiple labels (i.e., entities). ML-KNN is a state-of-the-art algorithm in the multi-label classification literature [113]. We employ the K-Nearest-Neighbor (KNN) search with a defined value of K to search the existing corpus ($doc_{past}$) based on similarities with the new bug description ($doc_{new}$) (i.e., the "K-NN" block in Figure 3.3). This search finds the top K similar files or methods. Cosine similarity is used to measure the similarity of the two document vectors (see the three corresponding blocks in Figure 3.3).

$$f_{sim}\left(doc_{new}, doc_{past}\right) = \frac{doc_{new}.doc_{past}}{|doc_{new}||doc_{past}|} \tag{3.1}$$

Once a new change request is received, by examining the created corpus, we can recommend the relevant source code entities that need to be fixed to resolve it. In summary, our *InComIA* approach (Formula (3.2)) is a model based on a union of elements in sets $C$ and $I$ plus the information from sets $CM$ and $BM$ which $Cor_{e_{in}}$ is a created corpora based on entity $e_{in}$.

$$\begin{gathered} \textit{InComIA} = Cor_{e_{in}} \uplus b_i \uplus c_i \\ (i, e_{i,n}) \in C \cup I, (i, b_i) \in BM, (i, C_i) \in CM \end{gathered} \tag{3.2}$$

We use *Mylyn* Interaction data and commits from the SCM for supporting IA. For access to the list of files that are interacted to fix previous bugs, we first extract *Mylyn* interaction files from

Table 3.1: Predicted files for bug# 201151 by three different approaches

| Approach | Predicted files |
|---|---|
| *SIA* | — |
| *ComIA* | AbstractRepositorySettingsPage.java |
| *InComIA* | AbstractRepositorySettingsPage.java<br>BugzillaRepositorySettingsPage.java |

the bug tracking system and then process them into transactions. Similarly for creating a list of files that are committed to fix previous bugs, we extract commits from the source code repository and then process them into transactions.

### 3.2.7 An Example from *Mylyn*

Here, we demonstrate our approach using an example from *Mylyn*. The change request of interest here is the bug# 201151 " *[patch]Bugzilla TaskRepositori Setting let Version Automatic being left when Version not supported*". This bug is fixed in revision# 5595. Our gold set GS, i.e., files changed, to fix this specific bug is *AbstractRepositorySettingsPage.java* and *IBugzillaConstants.java, BugzillaRepositorySettingsPage.java*.

Table 3.1 shows the results of search, i.e., files predicted by three different approaches. Approaches *SIA* and *ComIA* are discussed in Section 3.1. Only correctly predicted files out of 10 recommended ones are shown. The number of terms for *SIA* is 130607 and the number of documents is 1102. The number of terms for *ComIA* is 5294145 and the number of documents is 2398 and for *InComIA*, the number of terms is 15825476 and the number of documents is 2405. *InComIA* has the largest corpus. *InComIA* has the best performance in comparison with *ComIA* and *SIA*. *ComIA* and *InComIA* have larger corpora than *SIA* because these two approaches consider all the revisions in which files were interacted or committed plus commit messages and bug descriptions. Tables 3.2 and 3.3 show a few interesting commit messages, expressions, and comments for those revisions in which the file *AbstractRepositorySettingsPage.java* was committed and the file *BugzillaRepositorySettingsPage.java* was committed or interacted. This information does not exist in

*SIA*, because in *SIA* we only consider a single snapshot of software (in this case the revision 5594 is considered, which is the last). The expressions and comments extracted from source code are shown in tables 3.2 and 3.3 do not exist in our gold set source code files in revision 5594.

Table 3.2: Different comments and expressions extracted from file *AbstractRepositorySettingsPage.java* in different revision numbers and associated commit messages

| Revision | Comment, expression and commit messages |
|---|---|
| 2526 | ”Refactor TaskRepository to be extensible ” |
| 2872 | ”add Bugzilla repository templates” |
| 2827 | ”Trac connector: add repository templates for version support” |

Table 3.3: Different comments and expressions extracted from file *BugzillaRepositorySettingsPage.java* in different revision numbers and associated commit messages

| Revision | Comment, expression and commit messages |
|---|---|
| 5189 | ”Task List Repository preference page allows a negative number of days ” |
| 5268 | ”TaskRepository settings wizards don’t persist individual proxy settings” |

Our example suggests that comments and expressions from the different revisions of a file plus all of the commit messages and bug descriptions related to those revisions in the corpus could have important words related. If only one revision of software is considered in creating the corpus, we could lose important keywords from other revisions.

### 3.3 Empirical Evaluation

The main purpose of this case study was to investigate how well our combined approach *InComIA* for IA performed in predicting relevant entities for fixing a new incoming change request (e.g., bug). We compared *InComIA* with a previous approach for IA that indexes all the source code

in a single snapshot (denoted here as *SIA*). Moreover, we compared *InComIA* with an approach that uses past commits and bug descriptions (denoted here as *ComIA*). This comparison would permit the assessment of the impact of including interactions in *InComIA*.

### 3.3.1 Research Questions

We addressed the following research questions (RQs) in our case study:

- **RQ1.** How do *ComIA* (an IA approach trained from entities in only set *C*) and the *InComIA* (an IA approach trained by entities in combined set *C* and *I*) compare in predicting the top K relevant entities for incoming change requests?
- **RQ2.** How do IA approaches *ComIA* and *InComIA* compare with the previous model *SIA* in predicting the top K relevant entities for incoming change requests?

*ComIA* is a model based on committed entities in the set C plus the information from sets CM and BM. $Cor_{e_{in}}$ is a created corpora based on entity $e_{in}$ $(i, e_{i,n}) \in C, (i, b_i) \in BM, (i, C_i) \in CM$.

$$ComIA = Cor_{e_{in}} \uplus b_i \uplus c_i \tag{3.3}$$

*SIA* is a common model that is based on indexing the source code from a single snapshot (e.g., software release).

The purpose of **RQ1** was to assess if incorporating an additional set of interacted entities to our training set improved the accuracy of IA or not. The accuracy comparison between the approaches *InComIA* and *ComIA* would help us assess the level of the past interacted entities' contribution to the accuracy of *InComIA*. That is, *is it really worthwhile to put in the additional work of extracting entities interacted with in the resolution of past change requests, or would the committed entities would suffice?* The purpose of **RQ2** was to assess how do IA approaches based only source code entities that were interacted and/or committed in the past, compare with an approach that uses entire source code body in a single snapshot.

### 3.3.2 Experiment Setup

For *ComIA*, we considered all of the bugs that have the associated ID in a commit message. Subversion (SVN) repository commit logs were used to aid in this process. For example, keywords (such as the bug ID) in the commit messages/logs were used as starting points to determine if the commits were in fact associated with the mentioned change request in the issue tracking system. Similarly, we gained the associated revision numbers for each bug ID.

For *InComIA*, we considered all of the bugs that have at least one associated ID either in the commit messages or in the interacted trace files. A list of files (or methods) with the associated revision number was then extracted. The specific versions of the files (or methods) that were committed (in *ComIA*) and either committed or interacted (in *InComIA*) for each bug in the testing set were excluded from the training set. All of the necessary information (comments, identifiers and expressions) was extracted from each revision of the entity plus all of the commit messages and bug descriptions for each of the revisions in order to create the final training corpus. We used a dimensionality of 500, which is recommended for a large number of terms [28]. The experiment was run for two K values: K@10 and K@20.

### 3.3.3 Subject Software System

We focused our evaluation on the *Mylyn* project, which contains about 4 years of interaction data. It is an Eclipse Foundation project with the largest number of interaction history attachments. It is mandatory for *Mylyn* project committers to use the *Mylyn* plug-in. Commit history started 2 years prior to that of interaction, and commits to the *Mylyn* CVS repository terminated on July 01, 2011. To get both interaction and commit histories within the same period, we considered the history between June 18, 2007 (the first day of interaction history attachment) and July 01, 2011 (the last day of a commit to the *Mylyn* CVS repository).

### 3.3.4 Dataset

The *Mylyn* project consists of 2275 bug issues containing 3272 trace files. After preprocessing the traces and filtering out noises, 2357 file level and 2174 method level transactions were

identified. Table 3.4 provides information about the file and method levels of interaction transactions for the *Mylyn* project. There were more file level interaction transactions than method level interaction transactions. This difference may be due to the fact that *Mylyn* propagates lower-level interaction events into their parents. The *Mylyn* project contained 5093 revision histories. Out of 5093 change sets, 3727 revisions contained a change to at least one Java file and 2058 revisions contained a change to at least one Java method. About 3572 (96%) of file level changes and 1947 (95%) of method level changes were associated with issues.

Table 3.4: *Mylyn* project interaction and commit histories from June 18, 2007 to July 01,2011.

| System | Interaction | | Commit | |
|---|---|---|---|---|
| *Mylyn* | 3272 traces | | 5093 revisions | |
| | File | Method | File | Method |
| | 2357 | 2174 | 3727 | 2058 |

### 3.3.5 Training and Testing Sets

Both interaction and commit datasets were split into two groups: training and testing sets. The training sets were used to train our machine learning technique, and the testing sets were used to measure the effectiveness of three different models for impact analysis. We used the first 90% of the bugs for the training set and the next 10% of the bugs for the testing set. We had three different models and two levels of granularity. Therefore, we had a total of 6 training sets.

### 3.3.6 Performance Metrics

To evaluate the accuracy of the three models, we used two popular measures: precision and recall. Suppose that there are $m$ bug reports. For each bug report $b_i$ let the set of actual entities that were changed be $F_i$. We recommend the top K entities $E_i$ for $b_i$ with each of the approaches. Recall @K and precision @K for the $m$ bug reports are given by:

$$Recall@K = \frac{1}{m}\sum_{i=1}^{m}\frac{\mid E_i \cap F_i \mid}{\mid F_i \mid} \tag{3.4}$$

$$Precision@K = \frac{1}{m}\sum_{i=1}^{m}\frac{\mid E_i \cap F_i \mid}{\mid E_i \mid} \tag{3.5}$$

These metrics were computed for a recommendation list of entities with different sizes (i.e, K=10, K=20).

### 3.3.7 Hypotheses Testing

We derived testable hypotheses to evaluate our research questions. We only list the null hypotheses because one can easily derive the alternative hypotheses from them. SSD indicates statistically significant difference.

$H_{01}$: There is no SSD between the precision/recall values of *InComIA* and *ComIA* for the file-level.

$H_{02}$: There is no SSD between the precision/recall values of *InComIA* and *ComIA* for the method-level.

$H_{03}$: There is no SSD between the precision/recall values of *ComIA* and *SIA* for the file-level.

$H_{04}$: There is no SSD between the precision/recall values of *ComIA* and *SIA* for the method-level

$H_{05}$: There is no SSD between the precision/recall values of *InComIA* and *SIA* for the file-level.

$H_{06}$: There is no SSD between the precision/recall values of *InComIA* and *SIA* for the method-level.

We performed the analysis of variance (parametric ANOVA) test with $\alpha = 0.05$ to validate whether there is a statistically significant difference between the models. Table 3.5 is a heat-map summarizing the hypotheses test results across all three different models for two values of K. Significant difference: Cells colored black indicate that it exists for both method and file levels; for dark-gray it exists for only the file level; for light-gray it exists for only the method level; for white there is none for both levels.

Table 3.5: A heat-map summarizing hypotheses test results across all the three models for two different values of k.

| H | K@10 | | K@20 | |
|---|---|---|---|---|
| | Precision | Recall | Precision | Recall |
| $H_{01}$ | | | | |
| $H_{02}$ | | | | |
| $H_{03}$ | | | | |
| $H_{04}$ | | | | |
| $H_{05}$ | | | | |
| $H_{06}$ | | | | |

### 3.3.8 Case Study Results

In this section, we compare the results of three different models: *ComIA*, *InComIA* and *SIA*. For each change request in the benchmark, we parametrized three different models to predict the top ten and top twenty relevant entities. These recommendations were compared with the actual entities that were changed to fix the considered change request in order to compute the precision and recall values. Table 3.6 compares recall@10 and recall@20 at both the file and method levels of granularity. As expected, the recall value generally increases with the increase in the K value for each model, and the precision value decreases with the increase in the K value for each model. For example, for the *InComIA* model at the file level granularity, the recall@10 and recall@20 values are 0.26 and 0.32, and the precision@10 and precision@20 values are 0.15 and 0.12.

To answer the research question RQ1, we compare recall and precision values of the *InComIA* and *ComIA* models for different values of K. We computed the precision gain of *InComIA* over *ComIA*, which is computed using the formula:

$$GainP@k_{InComIA-ComIA} = \frac{prec@k_{InComIA} - prec@k_{ComIA}}{prec@k_{ComIA}} \times 100 \tag{3.6}$$

The two **GainP *InComIA-ComIA*** columns in Table 3.6 show the precision gain of *InComIA* over *ComIA* for different K values at both file and method levels. As can been seen, *InComIA* clearly outperforms *ComIA* at both the file and method levels for all K values. The precision gain at the

file level ranges from 33.33% to 36% and for method level ranges from 28.57% to 75%. Similarly, we computed the recall gain of *InComIA* over *ComIA*, which is computed using the formula:

$$GainR@k_{InComIA-ComIA} = \frac{rec@k_{InComIA} - rec@k_{ComIA}}{rec@k_{ComIA}} \times 100 \quad (3.7)$$

The two **GainR *InComIA-ComIA*** columns in Table 3.6 show the recall gain of *InComIA* over *ComIA* for the different K values at both file and method levels. As can been seen, *InComIA* clearly outperforms *ComIA* at both the file and method levels for all K values. The recall gain at the file level ranges from 28% to 44% and for the method level ranges from 58.82% to 80%. In summary, the overall results suggest that *InComIA* would generally perform better than *ComIA* in terms of recall and precision.

To answer the research question RQ2 we compare the recall and precision values of *InComIA* and *ComIA* with those of *SIA* for different values of K. We computed the precision gain of *InComIA* over *SIA* and the precision gain of *ComIA* over *SIA*, which is computed similarly with formula (3.6). The precision gain at the file level for *ComIA* over *SIA* ranges from 120% to 125% and for the method level it ranges from 100% to 250%. Therefore *ComIA* outperforms *SIA* at both the file and method levels for all different values of K. The precision gain at the file level for *InComIA* over *SIA* is 200% and for the method level it ranges from 250% to 350%. Therefore, *InComIA* outperforms *SIA* at both the file and method levels in terms of precision.

The recall gain of *InComIA* and *ComIA* over *SIA* is computed similarly with formula (3.7). From Table 3.6, the two columns labelled GainR *ComIA-SIA* show that at both the file and method levels *ComIA* outperforms *SIA* for all of the different values of K. The recall gain value for *ComIA* over *SIA* ranges from 108.33% to 125% at the file level and ranges from 150% to 183% at the method level. The recall gain value for *InComIA* over *SIA* ranges from 166.66% to 225% at the file level and is 350% at the method level. Therefore, *InComIA* outperforms *SIA* at both the file and method levels for recall. In summary, the overall results suggest that both *InComIA* and *ComIA* would generally perform better than *SIA* in terms of recall and precision.

To test our hypotheses, we applied the One Way ANOVA test on the recall and precision values of all three approaches with different values of K at both the file and method levels. From

Table 3.6: Recall@10 and 20 and Precision@10 and 20 of three models *InComIA*, *ComIA*, and *SIA*

| Recall@10 | | | | | | |
|---|---|---|---|---|---|---|
| Granularity | *InComIA* | *ComIA* | *SIA* | GainR *InComIA-ComIA* | GainR *ComIA-SIA* | GainR *InComIA- SIA* |
| File | 0.26 | 0.18 | 0.08 | 44.00% | 125.00% | 225.00% |
| Method | 0.18 | 0.1 | 0.04 | 80.00% | 150.00% | 350.00% |
| Precision@10 | | | | | | |
| Granularity | *InComIA* | *ComIA* | *SIA* | GainP *InComIA-ComIA* | GainP *ComIA-SIA* | GainP *InComIA- SIA* |
| File | 0.15 | 0.11 | 0.05 | 36.00% | 120.00% | 200.00% |
| Method | 0.09 | 0.07 | 0.02 | 28.57% | 250.00% | 350.00% |
| Recall@20 | | | | | | |
| Granularity | *InComIA* | *ComIA* | *SIA* | GainR *InComIA-ComIA* | GainR *ComIA-SIA* | GainR *InComIA-SIA* |
| File | 0.32 | 0.25 | 0.12 | 28.00% | 108.33% | 166.66 % |
| Method | 0.27 | 0.17 | 0.06 | 58.82% | 183.00% | 350.00 % |
| Precision@20 | | | | | | |
| Granularity | *InComIA* | *ComIA* | *SIA* | GainP *InComIA-ComIA* | GainP *ComIA-SIA* | GainP *InComIA-SIA* |
| File | 0.12 | 0.09 | 0.04 | 33.33% | 125.00% | 200.00% |
| Method | 0.07 | 0.04 | 0.02 | 75.00 % | 100.00 % | 250.00 % |

Table 5, we can see that there is a statistically significant difference between the precision and recall values of *InComIA* and *ComIA* at the file level for two different values of K, so we reject $H_{01}$. However, for the precision value of K@10 at the method level there is no statistically significant difference, so we accept $H_{02}$. For *ComIA* and *SIA* there is a statistically significant difference for the values of precision and recall at the file level. However, for the precision value of K@20 at the method level there is no statistically significant difference, Therefore we reject $H_{03}$ and we accept $H_{04}$. Results of the ANOVA test show a statistically significant difference between the precision and recall values of *InComIA* and *SIA* at both the file and method levels, so we reject both $H_{05}$ and $H_{06}$. In summary, the overall results from Table 5 suggest that *InComIA* would generally perform better than *ComIA* and *SIA* in terms of recall and precision.

## 3.4 Threats to Validity

We discuss internal, construct, and external threats to validity of the results of our empirical study.

*Incomplete or Missing Interaction History*: Although, a common period was considered for extracting the interaction and commit datasets in the Mylyn dataset, the number of commit transactions is significantly higher than the number of interaction transactions. This difference may not be the result of a single task getting defined for multiple commits because there are many cases in which committed files were never part of one of the corresponding interactions.

*CVS to SVN Conversion*: We do not know the error rate of CVS2SVN when grouping individual CVS files. It may erroneously split a commit into multiple, or group multiple commits into one. There are 1366 more commits than the number of interaction traces for the same period. This difference could be due to errors from CVS2SVN.

*Explicit Bug ID Linkage*: We considered interactions and commits to be related if there was an explicit bug id mentioned in them. Implicit relationships were not considered.

*Training and Testing Set Split*: We considered only a 90%:10% split between training and testing sets. It is possible that a different split point could produce different results.

*Single Period of History*: We considered only the history between June 18, 2007 and July 01, 2011. It is possible that this history is not reflective of the optimum results for all the models. A different history period might produce different results in terms of their relative performance.

*Accuracy and Practicality*: The accuracies of the two standalone techniques, however low in certain cases to raise a practicality concern, are comparable to other previous results (Zimmermann et al. 2005). Our work shows how to improve accuracy by forming effective combinations.

*Only One System Considered*: Due to the lack of adequate Mylyn interaction histories for open source projects, our validation study was performed only on a single system written in Java. It was the one with the largest available dataset within Eclipse Foundation. It had over 2600 fixed bug reports that contained at least one interaction trace attachment. The second and third largest

projects (the Eclipse Platform and Modeling) had about 700 and 450 such bugs. Nonetheless, this fact may limit the generality of our results.

# CHAPTER 4

## Change Impact Analysis (*RevIA*)

Similar to the previous chapter, this chapter presents an approach called *RevIA* to perform impact analysis (IA) of an incoming change request on source code. This approach uses the code review comments provided by developers in code review phase as an additional textual information to form the corpus. As we mentioned in Chapter 2, code review activity of developer is an example of micro-events. Information retrieval, machine learning, and lightweight source code analysis techniques were employed to form a corpus from the source code entities that are previously changed in the resolution of past change requests. Additionally, the corpus was augmented with the review comments written by developers during the review phase of the previously submitted patches. These patches are the results of the implemented changes in the source code to resolve the change requests. Given a textual description of a change request, this corpus is queried to obtain a ranked list of relevant source code entities that are most likely change prone. Such an approach that combines the code review comments with the textual information from the source code entities for IA at the change request level was not previously investigated. An empirical study on 1617 code reviews, 14691 review comments, and 4000 source code files from *Mylyn*, an open source task management tool, was conducted. The results show that the approach based on code review comments outperforms the approach based on only source code textual information.

### 4.1 Introduction

We introduced the Software-change impact analysis (IA) problem at previous chapter (Chapter 3). The solution we presented at Chapter 3 for task IA was a combined approach based on interaction and commit activities developers. In this chapter we present a new approach called *RevIA*. The novelty of *RevIA* is using code review comments from the code review process (micro-evolution repository) to create the corpus. All the review comments written for each source code file will be augmented to the corpus. As we described earlier about the review process, developers


Figure 4.1: A snippet of the code review comment in *Gerrit* which is written by *Sam Davis* for file *TaskReviewRelationshipListener.java* related to *Mylyn* project

make changes in their local *git* repositories and then submit these changes as a new patch for code review. Reviewers then inspect the code change through the code review tool (a web page in the case of Gerrit) and provide feedback in the form of review comments to the patch owner. Figure 4.1 shows an example of review comment written by *Sam Davis* for file *TaskReviewRelationshipListener.java* related to *Mylyn* project. Review comments are written for each source code entity within a patch i.e., each source code entity that is changed to fix the bug. Hence these review comments provide the textual information related to both source code and bug itself. The review comments are a mechanism that reviewers use to express their feedback and communicate with the owner and other peer reviewers of a code change. That is, these comments act as a medium for technical discussion about the code fix and the quality of it. They also play role in transferring knowledge between developers and it is possible to retrieve the latest information related to the source code from these review comments. Therefore, code review repository holds a wide range of information that are instrumental in finding a solution for various problems such as Impact Analysis (IA). *RevIA* combines these review comments with the textual information from the source code in order to form the corpus.

To evaluate the accuracy of our technique, we conducted an empirical study on the open

source system *Mylyn*. Precision, recall, and mean average precision metric values on a number of bug reports sampled from this system for IA are presented. That is, how effective our approach is at recommending the actual source code entities that are changed to fix these bugs. Additionally, we empirically compared our approach to another approach which is based on only textual information from the source code.

The results show that adding the review comments to the corpus increase the performance. Lowest recall, precision, and mean average precision gains of 67%, 100%, and 50% and highest recall, precision, and mean average precision gains of 126%, 166%, and 260% were recorded.

Our work makes the following noteworthy contributions in the context of performing IA on source code due to an incoming change request:

1. To the best of our knowledge, our approach is the first to integrate the code review activity of developers with the source-code textual information.
2. We performed a comparative study with an approach that makes an exclusive use of source-code textual information.

The rest of the chapter is organized as follows: Our *RevIA* approach is discussed in Section 4.2. The empirical study on *Mylyn* open-source projects and the results are presented in Section 4.3. Threats to validity are listed and analyzed in Section 4.4.

## 4.2 The *RevIA* Approach

Our *RevIA* approach to IA of an incoming change request on source code is almost similar to the presented approach *InComIA* in previous chapter and it consists of the following steps:

- The past change requests (e.g., bug reports) from software repositories are analyzed, and all the source code entities that were committed are extracted.
- The source code of each unique entity, from each revision (i.e., in subversion vocabulary) in which it was committed in the past, is parsed using a developer-defined granularity level. For each entity, comments, identifiers and expressions are then extracted from the parsed source code.

- For each source code entity the relevant code review comments are extracted from *Gerrit* repository and parsed. A corpus is created such that each source code entity has a corresponding document (i.e., in IR vocabulary) in it.

- Given a change request description, a ranked list of relevant source code entities (e.g., files and methods) is recommended for IA based on the K-Nearest Neighbor algorithm and Cosine similarity.

The first, second, and fourth steps are explained in detail in previous chapter. Below we explain the third step.

### 4.2.1 Extracting Code Review Comments and Creating a Corpus

In this section we briefly explain how we extract the code review comments from *Gerrit*

***Gerrit:*** To collect code review data for *Mylyn* we reverse engineered the *Grrit JSON API* and queried the *Gerrit* servers for data regarding the review for each project. *Gerrit* works by initially sending a web page skeleton and some Javascript to the browser. The Javascript then makes a number of web requests back to the *Gerrit* server and requests information about code review, which is returned in JSON format. *Gerrit* API provides an interface to JSON formatted review data [1]. This JSON data must still be parsed [73]. We also need to determine which fields in the displayed web page correspond to which fields within the JSON. The JSON response is fairly complex, deep, and redundant. After reading *Gerrit* Code Review tutorial documentation in section "*REST API/Query Changes*" we found that there is a GET command ('GET/changes') [2] which we can use for querying the changes submitted to the *Gerrit*. A query string must be provided by the $q$ parameter. The $n$ parameter can be used to limit the returned results. Each review in *Gerrit* has a unique *number* which can be used to query the information related to that review. Not all the data fields in the JSON response query meet our needs. We filter out the interesting fields which are used in our study.

[1] https://gerrit-review.googlesource.com/Documentation/rest-api.html
[2] https://gerrit-review.googlesource.com/Documentation/rest-api-changes.html#list-changes

**Creating a corpus from source code and cod review comments:** The corpus is created in a similar way that we explained in previous chapter section (3.2.5).

**Indexing and Querying the Corpus:** In *RevIA*, we use techniques from Natural Language Processing in order to locate the textually relevant files based on comments, identifiers, expressions and code review comments. This section is explained in detail in previous chapter (3.2.6).

### 4.2.2 Re-Ranking the Recommended Source Code Files with Issue Change Proneness

As we discussed in previous chapter, there is a one-to-many relationship between an IR query, i.e., description of a bug $b_i$, and source code files. Given a user provided the cutoff point of K, we get the K top ranked source code files $f_1, f_2, ....., f_k$ for the bug $b_i$. We use the change-proneness measure to re- rank these top K files. Issue Change Proneness (ICP) of a source code entity is a measure of its change affinity as determined from it's previous change (commit) history. A straightforward measure of the ICP of a source code entity is given by the number of commits in commit history that contain that source code entity[46]. The rationale behind this choice for re-ranking the set of the recommendation is based on the premise that the larger the number of changes related to past requests (e.g., bug fixes) in which the relevant source code file is involved, the higher the likelihood of the same file requiring changes due to a given (new) change request. For each bug $b_i$ in our benchmark we first extract the date which the bug was reported. Then we extract all the commits from code change history which were submitted within $m$ days before the the bug report date ($m$ is a configurable parameter). For each of the relevant source code files $f_1, f_2, ....., f_k$, we then calculate their ICPs from the extracted commits. The file with the highest ICP is ranked first, the one with the lowest ICP is ranked last, and so on. In our study we consider three different values for $m$.

- $m$=0 means considering the entire code change history.
- $m$=20 means considering all the commits from code change history which were submitted within the 20 days before the bug report date

- $m$=30 means considering all the commits from code change history which were submitted within the 30 days before the bug report date

We first run *RevIA* and *SIA* with out using any re-ranking mechanism, then we use these three different re-ranking mechanisms and we compare the results.

## 4.3 Empirical Evaluation

The main purpose of this case study was to investigate how well our approach *RevIA* for IA performed in predicting relevant entities for fixing a new incoming change request (e.g., bug). We compared *RevIA* with a previous approach for IA that indexes all the source code in a single snapshot, denoted here as *SIA*, explained in detail in previous chapter (Section 3.3). This comparison would permit the assessment of the impact of including code review comments in *RevIA*.

### 4.3.1 Research Questions

We addressed the following research questions (RQs) in our case study:

**RQ1.** How do IA approach *RevIA* compare with the previous model *SIA* in predicting the top K relevant entities for incoming change requests?

**RQ2.** How do the re-ranking mechanisms effect on the performance results?

### 4.3.2 Experiment Setup

For *RevIA*, we considered all of the bugs that have at least one associated ID either in the commit messages or code review repository. We used a dimensionality of 500, which is recommended for a large number of terms [28]. The experiment was run for two K values: K@10 and K@20. For *SIA*, we considered all of the bugs that have the associated ID in a commit message. Git repository commit logs were used to aid in this process. For example, keywords (such as the bug ID) in the commit messages/logs were used as starting points to determine if the commits were in fact associated with the mentioned change request in the issue tracking system. Similarly, we gained the associated revision numbers for each bug ID.

### 4.3.3 Subject Software System

We focused on *Mylyn* in this study. *Mylyn* contains about 3 years of code review data in *Gerrit* and is an Eclipse Foundation project which source code is written in java. Code review history in *Gerrit* started by Feb 2012. The *Mylyn* project consists of 1617 code reviews uploaded in *Gerrit* which includes at least one Java file [3]. 1549 Out of 1617 reviews have the status "MERGED" or "ABANDONED" which we just consider these reviews in our study. 78% of reviews in *Mylyn* includes patches which finally merged to repository and 22% of reviews are abandoned. Same as *Eclipse Platform* project noise were removed from review comments. After removing noise, totally 14691 review comments are written for *Mylyn* project.

### 4.3.4 Training and Testing Sets

The dataset is split into two groups: training and testing sets. The training sets were used to train our machine learning technique, and the testing sets were used to measure the effectiveness of two different models for impact analysis. We used the first 90% of the bugs for the training set and the next 10% of the bugs for the testing set. We considered 100 bugs in our benchmark.

### 4.3.5 Performance Metrics

To evaluate the accuracy of our approach, we used two popular measures: precision and recall. Suppose that there are $m$ bug reports. For each bug report $b_i$, let the set of actual entities that were changed be $F_i$. We recommend the top K entities $E_i$ for $b_i$ with each of the approaches. Recall @K and precision @K for the $m$ bug reports are given by:

$$Recall@K = \frac{1}{m}\sum_{i=1}^{m}\frac{\mid E_i \cap F_i \mid}{\mid F_i \mid} \tag{4.1}$$

$$Precision@K = \frac{1}{m}\sum_{i=1}^{m}\frac{\mid E_i \cap F_i \mid}{\mid E_i \mid} \tag{4.2}$$

$$MAP@K = \frac{1}{m}\sum_{i=1}^{m}\frac{1}{n_i}\sum_{z=1}^{n_i} Precision(R_{iz}) \tag{4.3}$$

$$n_i = \mid E_i \cap F_i \mid$$

[3] https://git.eclipse.org/r/#/q/mylyn,n,z

These metrics were computed for a recommendation list of entities with different sizes (i.e, K=10, K=20). We also calculated Mean Average Precision (MAP) by Equation 4.3. MAP is the mean of the average precision of each recommendation set for each bug report in our benchmark. For each bug report $b_i$ we have a set of recommended top K entities called $E_i = e_1, e_2, ...., e_z$ and $R_i z$ is the set of ranked retrieval results from the top result until we get to the correct entities.

### 4.3.6 Hypotheses Testing

We derived testable hypotheses to evaluate our research questions. We only list the null hypotheses because one can easily derive the alternative hypotheses from them. SSD indicates statistically significant difference.

$H_{01}$: There is no SSD between the precision, recall, and MAP values of *RevIA* and *SIA*.

$H_{02}$: There is no SSD between the precision, recall, and MAP values of results produced by running *RevIA* with and without using re-ranking mechanisms.

### 4.3.7 Case Study Results

In this section, we compare the results of *RevIA* and *SIA*. For each change request in the benchmark, we parameterized these two different models to predict the top ten and top twenty relevant entities. These recommendations were compared with the actual entities that were changed to fix the considered change request in order to compute the precision, recall, and MAP values.

Figure 4.2 shows the results for Recall, Precision, Mean Average Precision, and the calculated metric gains of two models *RevIA* and *SIA* with and without re-ranking mechanisms for K=10 and K=20. Based on what we explained in Section 4.2.2 we have four different type of ranking mechanisms. Hence Figure 4.2 presents four different groups of results that are explained as follows.

- No Re-ranking, presents the results of *RevIA* and *SIA* without performing any re-ranking mechanism.
- Re-ranking based on the entire history, presents the results of *RevIA* and *SIA* after re-ranking

the recommended source code files based on the calculated ICP from the entire code change history.

- Re-ranking based on the last 20 days before the bug report date, presents the results of *RevIA* and *SIA* after re-ranking the recommended source code files for each bug report in the benchmark. Re-ranking is done based on the calculated ICP from all the commits in code change history which were submitted within the 20 days before the bug report date.

- Re-ranking based on the last 30 days before the bug report date, presents the results of *RevIA* and *SIA* after re-ranking the recommended source code files for each bug report in the benchmark. Re-ranking is done based on the calculated ICP from all the commits in code change history which were submitted within the 30 days before the bug report date.

For each of these four groups we calculated recall, precision, and MAP for each test case in the benchmark for both *RevIA* and *SIA* and then we calculated the average of the results.

As expected, the recall value generally increases with the increase in the K value for each model, the precision value decreases with the increase in the K value for each model, and MAP is consistent with the increase in the K value for each model. The re-ranking mechanisms we explained in Section 4.2.2 are done in order to improve the precision and MAP results by changing the ranking of the recommendations.

To answer the research question RQ1, we compare recall, precision, and MAP values of the *RevIA* and *SIA* models for different values of K. We computed the metric gain of *RevIA* (i.e., X equals to precision, recall, and MAP) over *SIA* using the following formula:

$$GainX@K_{RevIA-SIA} = \frac{X@K_{RevIA} - X@K_{SIA}}{X@K_{SIA}} \times 100 \tag{4.4}$$

As can been seen, *RevIA* clearly outperforms *SIA* at all four different groups of results for both K=20 and K=10. The Gain% column at Figure 4.2 presents the calculated metric gain of *RevIA* over *SIA*. The recall gain ranges from 67.74% to 126.67% and precision gain ranges from 100% to 166.67%. The MAP gain ranges from 50% to 260%. Although the precision results for both *RevIA* and *SIA* are really low, but *RevIA* could improve the results by 100%.

| | No Re-ranking | | | | | |
|---|---|---|---|---|---|---|
| | K=20 | SSD | Gain % | K= 10 | SSD | Gain % |
| Recall *RevIA* | 0.39 | Yes | 77.27 | 0.34 | Yes | 126.67 |
| Recall *SIA* | 0.22 | | | 0.15 | | |
| Precision *RevIA* | 0.04 | Yes | 100.00 | 0.08 | Yes | 166.67 |
| Precision *SIA* | 0.02 | | | 0.03 | | |
| Mean Average Precision *RevIA* | 0.18 | Yes | 260.00 | 0.18 | Yes | 260.00 |
| Mean Average Precision *SIA* | 0.05 | | | 0.05 | | |

| | Re-ranking based on the entire history | | | | | |
|---|---|---|---|---|---|---|
| | K=20 | SSD | Gain % | K= 10 | SSD | Gain % |
| Recall *RevIA* | 0.53 | Yes | 70.97 | 0.41 | Yes | 95.24 |
| Recall *SIA* | 0.31 | | | 0.21 | | |
| Precision *RevIA* | 0.06 | Yes | 100.00 | 0.10 | Yes | 150.00 |
| Precision *SIA* | 0.03 | | | 0.04 | | |
| Mean Average Precision *RevIA* | 0.15 | No | 50.00 | 0.14 | No | 55.56 |
| Mean Average Precision *SIA* | 0.10 | | | 0.09 | | |

| | Re-ranking based on the last 20 days before the bug report date | | | | | |
|---|---|---|---|---|---|---|
| | K=20 | SSD | Gain % | K= 10 | SSD | Gain % |
| Recall *RevIA* | 0.51 | Yes | 70.00 | 0.47 | Yes | 67.86 |
| Recall *SIA* | 0.30 | | | 0.28 | | |
| Precision *RevIA* | 0.06 | Yes | 100.00 | 0.12 | Yes | 140.00 |
| Precision *SIA* | 0.03 | | | 0.05 | | |
| Mean Average Precision *RevIA* | 0.31 | Yes | 55.00 | 0.30 | Yes | 50.00 |
| Mean Average Precision *SIA* | 0.20 | | | 0.20 | | |

| | Re-ranking based on the last 30 days before the bug report date | | | | | |
|---|---|---|---|---|---|---|
| | K=20 | SSD | Gain % | K= 10 | SSD | Gain % |
| Recall *RevIA* | 0.52 | Yes | 67.74 | 0.50 | Yes | 72.41 |
| Recall *SIA* | 0.31 | | | 0.29 | | |
| Precision *RevIA* | 0.06 | Yes | 100.00 | 0.12 | Yes | 100.00 |
| Precision *SIA* | 0.03 | | | 0.06 | | |
| Mean Average Precision *RevIA* | 0.29 | Yes | 52.63 | 0.29 | Yes | 52.63 |
| Mean Average Precision *SIA* | 0.19 | | | 0.19 | | |

Figure 4.2: Recall, Precision, Mean Average Precision, and the calculated metric gains of two models *RevIA* and *SIA* with and without re-ranking mechanisms for K=10 and K=20.

To answer the research question RQ2, we compare recall, precision, and MAP values of the *RevIA* running with and without re-ranking mechanisms. The results from Figure 4.2 show that re-ranking the recommended source code files based on their calculate ICP improve the recall and precision in comparison with not using the re-ranking. Re-ranking mechanisms for *RevIA* could achieve the maximum recall gain of 47% , maximum precision gain of 50% , and maximum MAP gain of 66%. There is not much difference between the three different types of re-ranking mechanisms in terms of precision and recall.

In summary, the overall results suggest that *RevIA* generally performs better than*SIA*. Similarly, using re-ranking techniques improve the results.

To test our hypotheses, we applied the One Way ANOVA test on the recall, precision, and MAP values of *RevIA* and *SIA* approaches with different values of K. From Figure 4.2, we can see that there is a statistically significant difference between the recall, precision, and MAP values of *RevIA* and *SIA* for both K=20 and k=10 so we reject $H_{01}$. Similarly, there is a statistically significant difference between the recall, precision, and MAP values of of results produced by running *RevIA* with and without using re-ranking mechanisms so we reject $H_{02}$.

## 4.4 Threats to Validity

The threats to validity of the results of our empirical study is similar to what we explained in previous chapter (Section 3.4) and it only differs in one case.

*Code Review Comments' Noise*: During code review practice, there are several review comments that are generated and automatically submitted by the tools such as Hudson for *Mylyn* projects. Similarly, there are review comments submitted by reviewers which only include the review score but no other meaningful information. We considered these review comments as noise and we removed them form our dataset. There could be other sources of noise in review comments which are not considered as noise in our approach. This could effect the accuracy of results.

# CHAPTER 5

## Developer Recommendation (*iHDev*)

This chapter presents an approach, namely *iHDev*, to recommend developers who are most likely to implement incoming change requests [111]. The basic premise of *iHDev* is that the developers who interacted with the source code relevant to a given change request are most likely to best assist with its resolution. A machine-learning technique is first used to locate source code entities relevant to the textual description of a given change request. *iHDev* then mines interaction trails (i.e., *Mylyn* sessions) associated with these source code entities to recommend a ranked list of developers. *iHDev* integrates the interaction trails (micro-events) in a unique way to perform its task, which was not investigated previously.

An empirical study on open source systems *Mylyn* and *Eclipse Platform* was conducted to assess the effectiveness of *iHDev*. A number of change requests were used in the evaluated benchmark. Recall for top one to five recommended developers and Mean Reciprocal Rank (MRR) values are reported. Furthermore, a comparative study with two previous approaches that use commit histories and/or the source code authorship information for developer recommendation was performed. Results show that *iHDev* could provide a recall gain of up to 127.27% with equivalent or improved MRR values by up to 112.5%.

### 5.1 Introduction

Software change requests and their resolution are an integral part of software maintenance and evolution. It is not uncommon in open source projects to receive tens of change requests daily that need to be promptly resolved [9]. Issue triage is a crucial activity in addressing change requests in an effective manner (e.g., within time, priority, and quality factors). The task of automatically assigning issues or change requests to the developer(s) who are most likely to resolve them has been studied under the umbrella of bug or issue triaging. A number of approaches to address this task have been presented in the literature [9, 53, 65, 94, 107]. The fundamental idea underlying

most triage approaches is to identify the expertise and interests of developers, infer the concern or component that must be addressed for a task, and then match developers to tasks. Approaches typically operate on the information available from software repositories (e.g., models trained from past bugs/issues or source code changes), the source code authorship [65], or their combinations [46] under the rationale that if a developer frequently commits changes to, or fixes bugs in, particular parts of a system, they have knowledge of that area and can competently make changes to it.

We agree with the fundamental concept that historical records of developers' activity yield insight into their knowledge and ability, but we also posit that additional traces of developer activity such as interactions with source code (beyond those leading to commits) when resolving an issue can provide a more comprehensive picture of their expertise. The records of developers' interactions with code in resolving an issue remain largely untapped in solving this problem.

In this chapter, we show that exploiting these additional sources of such interactions leads to better task assignment than relying on commit and bug tracking histories. We present a new approach, namely *iHDev*, for assigning the incoming change requests to appropriate developers. *iHDev* is centered on the developers' interactions with source code entities that were involved in the resolution of previous change requests. Developers may interact with source code entities within an Integrated Development Environment (IDE) that may or may not be eventually committed to the code repository [15, 112]. These interactions (e.g., navigate, view, and modify) could have contributed in locating and/or verifying the entities that were changed due to a change request. Therefore, it suggests that the interacting developers are knowledgeable in those entities. On the face value, it could be conjectured that commits (i.e., changed entities) are a subset of interactions (i.e., viewed and changed entities). Our previous investigation found that a superset or subset relationship does not always hold [15]. Thus, the interaction trails have the potential to offer aspects that may not be embodied in commit histories. Tools such as *Mylyn* capture and store such interaction trails (histories) [74].

*iHDev* takes the textual description of an incoming change request (e.g., a short bug des-

cription) and locates relevant entities (e.g., files) from a source code snapshot. A machine learning technique, K-Nearest Neighbor (KNN) algorithm and cosine similarity, is used in this step. The interaction histories of these entities are mined to forge a ranked list of candidate developers to resolve the change request. *The basic premise of our approach is that the developers who interacted with the relevant source code to a given change request in the past are most likely to best assist with its resolution.* In a nutshell, our approach favors **i**nteraction **H**istories over other types of past information to recommend **Dev**elopers; hence, the name *iHDev*. It neither needs to mine for textually similar past change requests nor source code change (commit) histories. It only needs developer-interaction sessions from the issue repository of a system, which are typically attached to issue/bug reports (e.g., in the *Mylyn* prescribed XML format).

To evaluate the accuracy of our technique, we conducted an empirical study on two open source systems *Mylyn* and *Eclipse Platform*. Recall and Mean Reciprocal Rank (MRR) metric values of the developer recommendations on a number of bug reports sampled from this system are presented. That is, how effective our *iHDev* approach is at recommending the actual developer who ended up fixing these bugs. Additionally, our *iHDev* approach is empirically compared with two other approaches that use the commit and/or source code authorship information [65], [46], [53]. The results show that the presented *iHDev* approach outperformed these baseline competitors. Lowest recall gains of 6.17% and 9.72% were recorded against the two respective approaches. Highest recall gains of 125% and 127.27% were recorded against the two respective approaches. These gains came without incurring any decreased Mean Reciprocal Rank (MRR) values; rather *iHDev* recorded improvements in them. That is, *iHDev* would typically recommend the correct developers at higher ranks than the subjected competitors.

Our work makes the following noteworthy contributions in the context of recommending relevant developers to resolve incoming change requests:

1. To the best of our knowledge, our *iHDev* approach is the first to utilize developers' source code interaction histories involved with past change requests.

2. We performed a comparative study with two other approaches that use commit and/or source code authorship information.

The rest of the chapter is organized as follows: Our approach is discussed in Section 5.2. The empirical study on *Mylyn* and *Eclipse Platform*, and its results are presented in Section 5.3. Threats to validity are listed and analyzed in Section 5.4.

## 5.2 Approach

Our approach *iHDev* to assign an incoming change request to the appropriate developer(s) consists of the following steps:

1. **Locating Relevant Entities to Change Request**: We use the K-Nearest Neighbor (KNN) algorithm to locate relevant units of source code (e.g., files and classes) that match the given textual description of a change request or reported issue. The indexed source code release/snapshot is typically between the one in which an issue is reported and before the change request is resolved (e.g., a bug is fixed).
2. **Mining Interaction Histories to Recommend Developers**: The interaction histories of the units of source code from the above step are then analyzed to recommend a ranked list of developers that are the most experienced and/or have substantial contributions in dealing with those units (e.g., classes). The interaction histories are extracted from the issue-tracking system.

### 5.2.1 Key Terms and Definition

`Interaction:` Interaction is the activity of programmers in an IDE during a development session (e.g., editing a file or referencing an API documentation).

Tools, such as *Mylyn*, have been developed to model programmers' actions in IDEs [74]. *Mylyn* monitors programmers' activities inside the *Eclipse IDE* and uses the data to create an *Eclipse* user interface focused around a task. The *Mylyn* interaction consists of traces of interaction histories. Each historical record encapsulates a set of interaction events needed to complete a task

(e.g., a bug fix). Once a task is defined and activated, the *Mylyn* monitor records all the interaction events (the smallest unit of interaction within an IDE) for the active task. For each interaction, the monitor captures about eleven different types of data attributes. The most important of these is the structure handle attribute, which contains a unique identifier for the target element affected by the interaction. For example, the identifier of a Java class contains the names of the package, the file to which the class belongs to, and the class. Similarly, the identifier of a Java method contains the names of the package, the file and the class the method belongs to, the method name, and the parameter type(s) of the method.

`Trace file:` For each active task, *Mylyn* creates an XML trace file, typically named *Mylyn*-context.zip or its derivative. A trace file contains the interaction history of a task. This file is typically attached to the project's issue tracking system (e.g., *Bugzilla* or JIRA). The trace files for the *Mylyn* project are archived in the *Bugzilla* as attachments to a bug report. For example for issue #315184 there is one trace file named *Mylyn*-context.zip[1].

`Attacher:` Each issue/bug could have trace files. A developer submits these trace files to the project's issue tracking system and are attached to the associated issue report. We term this developer as the attacher. This term helps differentiate from the use of committers and developers in the context of source code repositories. There is no explicit information to distinguish between the attacher and the actual developer who performed the interaction session. We assume that the attacher is the developer who performed the attached interaction session. This issue is similar to the distinction between the developer who performed and the committer who committed the changes to a source code repository. For example, *Steffen Pingel* attached the file *Mylyn*-context.zip for issue #315184. Considering the *Mylyn* workflow for interactions, we did not expect nor found any instances where the attacher was not the developer who performed the interaction session.

[1]https://bugs.eclipse.org/bugs/show_bug.cgi?id=315184

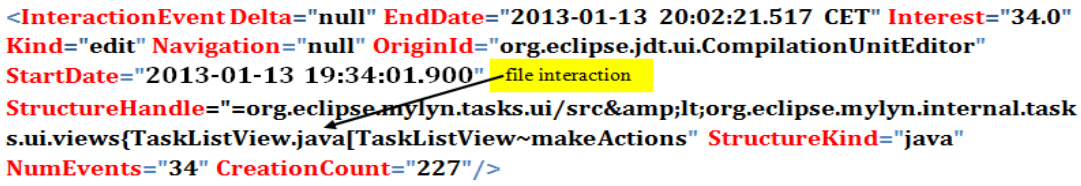


Figure 5.1: A snippet of an interaction event recorded by *Mylyn* for bug issue #330695 with trace ID #221752. File TaskListView.java is edited.

### 5.2.2 Locating Relevant Entities to Change Request

In our approach, we use techniques from natural language processing and machine learning to locate textually relevant source code files to a given change request for which we need to assign developers. The specific steps are given below:

**Creating a corpus from software**

The source code of a release, in or before which the change request is resolved, is parsed using a developer-defined granularity level (e.g., file) and documents are extracted. A corpus is created such that each file will have a corresponding document therein. Identifiers, comments, and expressions are extracted from the source code. Each document in the corpus is referred to as $doc_{past}$.

**Preprocessing Step**

Each document $doc_{past}$ is preprocessed in two parts: removing stop words and stemming.

**Document-Term Representation**

`Document Indexing`: We produce a dictionary from all of the terms in our document and assign a unique integer Id to each term appearing in it.

`Term Weighting`: We use *tfidf* in our approach. The global importance of a term *i* is based on its inverse document frequency (*idf*), calculated as the reciprocal of the number of documents that the term appears in. *idf* is the document frequency of term *i* in the whole document collection. $tf_{i,j}$ is the term frequency of the term *i* in the document *j*. Each document $d_j$ is represented as a

vector $d_j = (w_{1,j}, ..., w_{i,j}, ..., w_{n,j})$ where $n$ is the total number of terms in our document collection and $w_{i,j}$ is the weight of the term $i$ in document $j$.

**Using Change Request**

The textual description of a change request for which we want to find all of the relevant files, and eventually to be assigned developer(s), is referred to as $doc_{new}$. $doc_{new}$ also goes through the preprocessing step and is represented as a document.

**K-Nearest-Neighbor**

The task of multi-label classification is to predict for each data instance a set of labels that applies to it. Standard classification only assigns one label to each data instance. However, in many settings a data instance can be assigned by more than one label. In our context, each data instance (i.e., a bug report) can be assigned multiple labels (i.e., source code files). ML-KNN is a state-of-the-art algorithm in the multi-label classification literature [113]. We employ the ML-KNN search with a user-defined value of $K$ (e.g., the value of 10 as used in previous work [107]) to search the existing corpus ($doc_{past}$) based on similarities with the new bug description ($doc_{new}$). This search finds the top $K$ similar files. We consider these top $K$ files to be a set, i.e., they are eventually relevant to the change request regardless of their similarly levels or ranking. Cosine is then used to measure the similarity of the two document vectors.

$$f_{sim}\left(doc_{new}, doc_{past}\right) = \frac{doc_{new}.doc_{past}}{|doc_{new}||doc_{past}|} \tag{5.1}$$

At the conclusion of this step, we have identified the $K$ most relevant source code files to the given change request. Now, we need to mine the candidate developers from the interaction histories of these files.

### 5.2.3 Mining Interaction Histories to Recommend Developers

The basic premise of *iHDev* is that the attachers who substantially interacted with the specific source code in the past are most likely the best candidate to assist with the issues/bugs associated with it. Our approach uses an *interaction log*, which was assembled from source code interactions

```
<BugId Id="315184">
<AttachId Id="196936"/>
<Attacher name="Steffen Pingel"/>
<InteractionEvent EndDate="2011-05-31" Kind="edit"
StructureHandle="=...tasks.ui.wizards{AbstractTaskRepositoryPage.java;"/>
</BugId>
```

Figure 5.2: A snippet of the bug #315184 interaction log entry from its interaction log file.

submitted by attachers to the bug tracking system (e.g., *Bugzilla*). Interaction log entries include the dimensions: attacher, date, and path (e.g., files) involved in an interaction event. Interaction logs are not directly available from the issue/bug tracking systems. We engineered the interaction log to simplify the implementation of the mining component of *iHDev*. A notable side effect is that it provides an analogy to commit logs, which are a basis for several developer-recommendation approaches.

`Interaction Log:` An Interaction Log is a file that includes the interaction history and the attacher(s) who created the trace file for each bug in the issue tracking system. The specific steps for creating an interaction log are given below:

**Extracting Interactions**

We first need to identify the bug reports with the *mylyn-context.zip* attachment(s) because not all bug issues necessarily contain the interaction trace(s). We searched the *Bugzilla* issue tracking system and included bugs containing at least one *mylyn-context.zip* attachment.

**Determining the list of attachers**

All the trace files from the issue-tracking system are automatically downloaded to a user specified directory. A complete list of attachers for each trace file of every bug is then produced.

**Creating Interaction log**

The tool then takes the directory that contains the trace files as input and parses each trace file to create an interaction log. For each bug ID, this interaction log includes the attacher (from the list of attachers) and three of the eleven attributes from the trace file (see Figure 5.1): *End-*

*Date, Kind*, and *StructureHandle*. There are several different kinds of interaction events (e.g., edit, manipulation, selection, and propagation); however, we consider only the edit interaction events because these events refer to interactions that resulted in a change to the source code file (even if that change was never committed), an action that often requires some level of knowledge of the source code. Other events such as navigation or selection do not imply explicit interaction or expertise and can therefore potentially introduce noise rather than provide additional useful information. Also, it is possible for one bug to have multiple trace files, each with a different attacher. Thus, for each bug in the interaction log file, multiple log entries may exist. Figure 5.2 shows a log entry for bug #315184 in the interaction log, which has only one trace file (and one attacher). For this bug, only one file has an edit interaction event (i.e.,*"AbstractTaskRepositoryPage.java"*).

The interaction log includes data such as the bug ID, attacher, the interacted source code, and the date on which the source code was interacted to fix this issue (see Figure 5.2 for an example). This data explains who interacted with the source code and when they did it. An attacher may contribute multiple interactions with the same file or multiple attachers may interact with the same file in different trace files. Therefore, interactions give an opportunity to analyze both the exclusive and shared contributions of attachers to a source code file.

`Developer Expertise Measures`: One measure of an attacher's contribution is the total number of interactions on source code performed in the past [71]. An attacher who contributed a larger number of interactions on a specific part of the source code than another attacher can be considered as more knowledgeable on those parts. Another consideration for attacher contributions is the workdays (i.e., activity) involved in the interactions that are attached as a trace file. The activity of a specific attacher is the percentage of their workdays over the total workdays of the system. Here, an attacher's workday is considered as a day (calendar date) on which they interacted with at least one part of the source code, because an attacher can have multiple interactions on a given workday. A system's workday is considered a day on which at least one part of the source code is interacted. A day on which no interactions exist is not considered a workday. These two measures give us two different views on attachers' development styles. Some may perform smaller

interaction sessions and submit frequently in a workday (e.g., multiple attachments), while others may do it differently (e.g., single attachment). The third measure accounts for the recency of these interactions. We used these three measures, which were inspired by our previous work on commits [53], to determine the attachers that were more likely to be experts in a specific source code file, i.e., *attacher-interaction* map. The *attacher-interaction* map, $AI$ for the attacher *a* and file $f$ is given below:

$$AI_{(a,f)} = \langle I_f, A_f, R_f \rangle$$

- $I_f$ is the number of interactions that include file $f$ and are interacted by the attacher *a*.
- $A_f$ is the number of workdays in the activity of attacher *a* with interactions that include the file *f*.
- $R_f$ is the most recent workday in the activity of the attacher *a* with an interaction that includes the file *f*.

Similarly, the *file-interaction* map $FI$ represents the interaction contribution to the file *f*, and is shown below:

$$FI_{(f)} = \langle I'_f, A'_f, R'_f \rangle, \text{ where}$$

- $I'_f$ is the number of interactions that include file *f*.
- $A'_f$ is the total number of workdays in the activity of all attachers that include interactions with the file *f*.
- $R'_f$ is the most recent workday with an interaction that includes the file *f*.

The measures $I_f$, $A_f$, and $R_f$ are computed from the interaction log. More specifically, the dimensions attacher, date, and paths of the log entries are used in the computation. The dimension date is used to derive workdays or calendar days. The dimension attacher is used to derive the attacher information. The dimension path (StructureHandle) is used to derive the file information. The measures $I'_f$, $A'_f$, and $R'_f$ are similarly computed. The log entries are readily available in the

Table 5.1: The attachers extracted with *iHDev* from each of the top ten files relevant to Bug# 313712.

| **Files** | **Ranked Attacher based on *xFactor* value** |
|---|---|
| …/TaskEditorNewCommentPart.java | Jingwen 'Owen':1.23, Frank Becker:1.06, Steffen Pingel:0.54, Jacek Jaroczynski:0.19, David Shepherd:0.07, David Green:0.06 |
| …/CopyContextHandler.java | Shawn Minto:2.00, Steffen Pingel:0.61, Mik Kersten:0.42 |
| …/NewAttachmentWizardDialog.java | Frank Becker:1.40, David Green:0.90, Steffen Pingel:0.60, Mik Kersten:0.30 |
| …/Messages.java | .... |
| …/WebBrowserDialog.java | .... |
| …/BugzillaResponseDetailDialog.java | Frank Becker:3.00 |
| …/UpdateAttachmentJob.java | Frank Becker:1.94, Robert Elves:1.40 |
| …/TaskAttachmentPropertyTester.java | Philippe Marschall:1.63, Steffen Pingel:1.38 |
| …/TaskEditorRichTextPart.java | Frank Becker:1.11, Jingwen 'Owen' Ou:0.90, Steffen Pingel:0.66, Thomas Ehrnhoefer:0.26, Jacek Jaroczynski:0.12, David Green:0.11, David Shepherd:0.03 |
| …/BugzillaPlanningEditorPart.java | Steffen Pingel:1.85, Robert Elves:1.35 |

form of *XML* and straightforward *XPath* queries are formulated to compute the measures. The contribution or expertise factor, termed *xFactor*, for the attacher $a$ and the file $f$ is computed using the ratios of the *attacher–interaction* and *file–interaction* maps. The contribution factor, *xFactor*, is given below:

$$xFactor(a, f) = \frac{AI_{(a,f)}}{FI_{(f)}} \tag{5.2}$$

$$xFactor(a, f) = \begin{cases} \frac{I_f}{I'_f} + \frac{A_f}{A'_f} + \frac{1}{|R_f - R'_f|} & \text{if } |R_f - R'_f| \neq 0 \\ \frac{I_f}{I'_f} + \frac{A_f}{A'_f} + 1 & \text{if } |R_f - R'_f| = 0 \end{cases} \tag{5.3}$$

The *xFactor* score is computed for each of the relevant source code files to the given change request (see Section 5.2.2). According to Equation 5.3, the maximum value of *xFactor* can be three because we have used three measures, each of which can have a maximum contribution ratio of 1.

`Recommending developers based on xFactor scores`: We now describe how the ranked-list of developers is obtained from all of the scored attachers of each relevant source code file to a

given bug. From Section 5.2.3, there is a one-to-many relationship between the source code files and attachers. That is, each file $f_i$ may have multiple attachers; however, it is not necessary for all of the files to have the same number of attachers. For example, the file $f_1$ could have two attachers and the file $f_2$ could have three attachers. Each row in the matrix $D_a$ (see Equation 5.4) gives the list of unique attachers for each relevant file $f_i$. $D_{af_i}$ represents the set of attachers, with no duplication, for the file $f_i$, where $1 \leq i \leq n$ and $n$ is the number of relevant files. $a_{ij}$ is the $j^{th}$ attacher in the file $f_i$ with $l$ unique attachers.

$$D_a = \begin{pmatrix} f_1 & D_{af_1} \\ f_2 & D_{af_2} \\ \cdot\cdot & \cdot\cdot \\ \cdot\cdot & \cdot\cdot \\ f_n & D_{af_n} \end{pmatrix} D_{af_i} = \{\, a_{i1} \;\; a_{i2} \;\; \ldots \;\; a_{il} \,\} \tag{5.4}$$

Although, a single file does not have any duplicate attachers, two files may have common attachers. In Equation 5.5, $D_{au}$ is the union of all unique attachers from all relevant files.

$$D_{au} = \bigcup_{i=1}^{n} D_{af_i} \tag{5.5}$$

$$Score(a) = \sum_{i=1}^{n} xFactor_i(a, f_i) \tag{5.6}$$

Each attacher $a$ for a file $f$ has the $xFactor$ score. To obtain the likelihood of the attacher $a$, i.e., $Score(a)$, to resolve the given change request, we sum $xFactor$ scores of the relevant files in which it appears (see Equation 5.6). That is, their overall expertise in all the files is computed collectively. The $Score(a)$ value is calculated for each unique attacher $a$ in the set $D_{au}$.

In Equation 5.7, we have a set of candidate developers. The developers in this set are ranked based on their $Score(a)$ values. Once the developers are ranked in the descending order of their $Score(a)$ values, we have a ranked list of candidate developers. By using a cutoff value of $m$, we recommend the top $m$ candidate developers, i.e., with top $m$ $Score(a)$ values, from the ranked list obtained from the set $DF$.

$$DF = \{(a, Score(a)), \forall a \in D_{au}\} \tag{5.7}$$

This step concludes *iHDev* and we have the top $m$ candidate developers recommended for the given change request.

### 5.2.4 An Example from *Mylyn*

Here, we demonstrate *iHDev* using an example from *Mylyn*. The change request of interest here is the bug #313712 *"attachment dialog does not resize when Advanced section is expanded"*. We first collected a snapshot of *Mylyn*'s source code prior to this bug fixed and then parsed it using the file-level granularity (i.e., each document is a file). After indexing with a machine learning technique, we obtained a corpus consisting of 1,825 documents and 201,554 words. A search query was then formulated using the bug's textual description, the result of which (i.e., top 10 relevant files) is summarized in Table 5.1. These ($k = 10$) files are our starting point for *iHDev*. The correct developer who fixed this bug and committed the change is *Frank Becker*. In Table 5.1, the third column shows a set of all attachers with the *xFactor* values for each file $f_i$.

In *iHDev*, we first obtained the set $D_{au}$ from all of the attachers recommended for each relevant file $f_i$ to the bug #313712 in Table 5.1. The set $D_{au}$ consists of 11 unique attachers. Because a developer could use different identities for the attacher and committer roles, we normalized them to a single identity, which was their full name. For each of the 11 unique attachers, the $Score$ value is calculated according to Equation 5.6. Table 5.2 shows the top five $Score$ values and the corresponding attachers, i.e., $m = 5$. *Frank Becker* has the highest score in the set $DF$ (a value of 8.51), so he is the first recommended developer. For the remaining attachers, the value of the function $Score$ is less than *Frank Becker*'s score, so they all have a rank greater than 1.

Table 5.2: Top five attachers (developers) recommended to resolve bug #313712 by *iHDev*.

| **Attacher** | ***Score*** | **Rank** |
|---|---|---|
| *Frank Becker* | 8.51 | 1 |
| *Steffen Pingel* | 5.64 | 2 |
| *Robert Elves* | 2.75 | 3 |
| *Jingwen 'Owen' Ou* | 2.13 | 4 |
| *Shawn Minto* | 2.0 | 5 |

Table 5.3 shows the results for the approaches *iHDev*, *xFinder*, *xFinder′* and *iMacPro* (des-

cribed in sections 5.3.1) for $m = 5$. Clearly, the best result is for *iHDev*, as it recommends *Frank Becker* (the correct developer who fixed bug #313712) in the first position, whereas *xFinder*, *xFinder′*, and *iMacPro* recommend *Frank Becker* in the third, fourth, and third position respectively. *iHDev* outperforms the others with respect to recall@1 and recall@2 values. At recall@5, all the approaches would be equivalent; however, *iHDev* provides the best ranked position (i.e., the reciprocal ranked value).

Table 5.3: Top five recommended developers and their associated ranks for the compared approaches. *iHDev*, *xFinder*, *xFinder′* and *iMacPro*.

| **Approach** | **Top five recommended developers in ranked order** |
|---|---|
| *iHDev* | Frank Becker ①, Steffen Pingel ②, Robert Elves ③, Jingwen 'Owen' Ou ④, Shawn Minto ⑤ |
| *xFinder* | Steffen Pingel ①, Robert Elves ②, Mik Kersten ②, Frank Becker ③ |
| *xFinder′* | Steffen Pingel ①, Mik Kersten ②, Robert Elves ③, Frank Becker ④, Shawn Minto ⑤ |
| *iMacPro* | Steffen Pingel ①, Shawn Minto ②, Frank Becker ③ |

## 5.3 Case Study

The purpose of this study was to investigate how well our *iHDev* approach recommends expert developers to assist with incoming change requests. We also compared our approach with two previously published approaches. The first approach, *xFinder* [53], is based on the mining of commit logs. The second approach, *iMacPro* [46], uses the authorship information and maintainers of the relevant change prone source code to recommend developers. We used these two approaches for comparison because they require information from the commit repository. *iHDev* uses interaction history of source code. Therefore, this part of the study would allow us to compare interactions and commits with respect to the developer recommendation task. We addressed the fol-

lowing research question **RQ**: How do *iHDev* (trained from the interaction history) compare with *xFinder*, *xFinder′*, and *iMacPro* (trained from the commit history) in recommending developers?

### 5.3.1 Compared Approaches: *xFinder*, *xFinder′*, and *iMacPro*

The *xFinder* approach uses the source code commit history to recommend developers to assist with a given change request. The first step is finding the relevant source code files for the change request, similar to *iHDev* (see Section 5.2.2). The commit (and not interaction) histories of these files are analyzed to recommend a ranked list of developers. *xFinder* uses the frequency count of the developers in the relevant files for ranking purposes. For example, if a developer occurs in three relevant files, its score is assigned a value of 3, and is ranked above another developer that occurs in two relevant files (with a score of 2). If multiple developers have the same score, they are given the same rank (e.g., two developers are ranked second in Table 5.3). *iHDev* uses a different ranking mechanism. We replaced *xFinder*'s ranking algorithm with that of *iHDev*, which is based on the sum of $xFactor$ scores. This modified *xFinder* approach is termed *xFinder′*. *xFinder′* allows us to compare the core of two approaches (i.e., interactions and commits) by neutralizing the variability in their rankings.

The *iMacPro* approach uses the source code authorship and commit history to assign incoming change requests to the appropriate developers. The first step is finding the relevant source code files for the change request, similar to *iHDev* (see Section 5.2.2). These source code units are then ranked based on their change proneness. Change proneness of each source code entity is derived from its commit history. The developers who authored and maintained these source code files are discovered and combined. Finally, a final ranked list of developers for the given change request is recommended. Hossen et al. [46] showed that *iMacPro* outperformed the *iA* approach [65], which was shown to perform equivalent to or better than the approach of Anvik et al. [9].

### 5.3.2 Subject Software Systems

We focused on the *Mylyn* and *Eclipse Platform*.

***Mylyn***

*Mylyn* contains about 4 years of interaction data and is an Eclipse Foundation project with the largest number of interaction history attachments. It is mandatory for *Mylyn* project committers to use the *Mylyn* plug-in. Commit history started 2 years prior to that of interaction, and commits to the *Mylyn* CVS repository terminated on July 01, 2011. To get both interaction and commit histories within the same period, we considered the history between June 18, 2007 (the first day of an interaction history attachment) and July 01, 2011 (the last day of a commit to the *Mylyn* CVS repository). Doing so ascertained that none of the approaches had any particular advantage or disadvantage over the others due to the lack of history for training. The *Mylyn* project consists of 2272 bug issues containing 3282 trace files for a total of 276390 interaction logs for interaction events that lead to edit of the interacted source code. The *Mylyn* project consists of 5093 revisions, out of which 3727 revisions contained a change to at least one Java file and for 3536 revisions there exists at least one trace file in the interaction history.

***Eclipse Platform***

*Eclipse Platform* contains 6 different sub projects: *e4*, *Incubator*, *JDT*, *Orion*, *PDE* and *Platform*[2]. *Eclipse Platform* contains about 7 years of interacted data from July 2007 to May 2014. It is not mandatory for the *Eclipse Platform* committers to use the *Mylyn* plug-in. Therefore, it is not surprising that the number of issues that contain interactions is less than those in the *Mylyn* project. The commit history for *Eclipse Platform* started in April 2001, 6 years prior to that of interaction, and commits to the *Eclipse Platform* CVS repository terminated on November 05, 2012. To get both interaction and commit histories within the same period, we considered the history between July 25, 2007 (the first day of an interaction history attachment) and November 05, 2012 (the last day of commits/revisions to the *Eclipse Platform* CVS repository). During this history the *Eclipse Platform* consists of 700 bug issues containing 897 trace files for a total of 95834 interaction logs for interaction events that lead to edit of the interacted source code. It consists of 52126 revisions,

[2]http://www.eclipse.org/eclipse/

out of which 35290 revisions contained a change to at least one Java file and for 861 revisions there exists at least one trace file in the interaction history.

Both *xFinder* and *iMacPro* need the commit histories of the *Mylyn* and *Eclipse Platform* open source systems because they need files that have been changed together in a single commit operation. SVN preserves the atomicity of commit operations; however, older versions of CVS did not. Subversion assigns a new ”revision” number to the entire repository structure after each commit. The ”older” CVS repositories were converted to SVN repositories using the CVS2SVN tool, which has been used in popular projects such as *gcc3*. For the datasets considered in our study, *Mylyn* and *Eclipse Platform* had their commit histories in CVS depositories. Therefore, we converted the CVS repositories to SVN repositories.

### 5.3.3 Benchmarks: Training and Testing Datasets

For *Mylyn* and *Eclipse Platform*, we created a benchmark of bugs and the actual developers who fixed them to conduct our case study. The benchmark consists of a set of change requests that has the following information for each request: a natural language query (request summary) and a gold set of developers that addressed each change request. The benchmark was established by a manual inspection of the change requests, source code, their historical interactions in the bug tracking system, and their historical changes recorded in version-control repositories. Interaction trace files were used to find the interaction events related to each bug and the attacher who created the trace file was used as the attacher in our interaction log. For tracing each *Bug ID* in the Subversion (SVN) repository commit logs, keywords such as *Bug ID* in the commit messages/logs were used as starting points to examine if the commits were in fact associated with the change request in the issue tracking system that was indicated with these keywords. The author and commit messages in those commits, which can be readily obtained from SVN, were processed to identify the developers who contributed changes to the change requests (i.e., gold set). These developers were then used to form our actual developer set for evaluation. A vast majority of change requests are handled by a single developer (92% in *Mylyn* and 97% in *Eclipse Platform*). Table 5.4 shows the frequency distributions of developers resolving issues in the benchmarks for *Mylyn* and *Eclipse*

Table 5.4: The frequency distributions of developers resolving issues in the benchmarks for *Mylyn* and *Eclipse Platform*.

| System | Frequency distribution | | | Total Issues |
|---|---|---|---|---|
| | # 1 | # 2 | # 3 | |
| *Mylyn* | 277 | 21 | 3 | 301 |
| *Eclipse Platform* | 70 | 0 | 2 | 72 |

*Platform*. For example, 277 issues were resolved by a single developer in *Mylyn* and none of the issues were resolved by two developers in *Eclipse Platform*. Our technique operates at the change request level, so we also needed input queries to test. We considered all of the bugs that have at least one associated Id in the commit messages and traces files. We created the final training corpus from all of the source code files in the last revision before the bug issues in our benchmark were fixed. We split the benchmark into training and testing sets. We picked June 18, 2007 to March 16, 2010 as our training set and March 16, 2010 to July 01, 2011 as our testing set for *Mylyn*. The testing set period contains 600 different revisions and 301 different issues/bugs. Similarly We picked July 25, 2007 to March 01, 2010 as our training set and March 01, 2010 to November 05, 2012 as our testing set for *Eclipse Platform*. The testing set contains 140 different revisions and 72 different issues/bugs. Our benchmarks are available online[3]. The experiment was run for $m = 1$, $m = 2$, $m = 3$, and $m = 5$, where $m$ is the number of recommended developers. We considered the top *ten* relevant files from the machine learning step.

### 5.3.4 Metrics and Statistical Analyses

We evaluated the accuracy of each of the approaches for all of the bug issues in our testing set using the Recall and Mean Reciprocal Rank (MRR) metrics used in previous work [9, 65, 107]. For each bug *b*, in a set of bugs $B$ of size $n$, in the benchmark of a system and a $m$ number of recommended developers, the formula for the recall@m is given below:

$$recall@m = \frac{\mid RD(b) \cap AD(b) \mid}{\mid AD(b) \mid} \tag{5.8}$$

[3]http://serl.cs.wichita.edu/svn/projects/dev-rec-interactions/trunk/MSR2015/data/Benchmark

where *RD(b)* and *AD(b)* are the recommended developer by the approach and the actual developer who resolved the issue for the bug *b* respectively. This metric is computed for recommendation lists of developers with different sizes (e.g., $m = 1$, $m = 2$, $m = 3$, and $m = 5$ developers).

Increasing the value of $m$ could lead to a higher recall value (and typically does); however, it may come at the cost of an increased effort in examining the potential noise (false positives). Over 90% of the cases in our benchmarks have only a single correct developer. In such a scenario, each increment to $m$ in pursuit of a correct developer could add drastically to the proportion of false positives. Therefore, a traditional metric of the likes of precision is not a suitable fit for our context. For example, if a correct developer is found at $m = 5$, the possible precision value is in the range [0.2, 1.0] for the rank positions [5, 1], typically around the lower bound of 0.2. Note that the precision metric is also agnostic to the rank positions of recommendations. Therefore, for a cutoff value of $m$, it would produce the same value for two approaches presenting the same number of correct answers. For $m = 5$, two approaches presenting a single correct answer at the positions 1 and 5 respectively will have the same precision value of 0.2. Nonetheless, a complimentary measure to recall is also needed to assess the potential effort in addressing noise (false positives). We focused on evaluating the ranked position of the correct developer for each bug in each benchmark from a cumulative perspective regardless of the cutoff point $m$.

Mean Reciprocal Rank (MRR) is one such measure that can be used for evaluating any process that produces a list of possible responses to a sample of queries, ordered by probability of correctness. This metric has been used in previous work [100]. The reciprocal rank of a query response is the multiplicative inverse of the rank of the first correct answer. Intuitively, the lower the value (between 0 and 1), the farther down the list, examining incorrect responses, one would have to search to find a correct response.

$$MRR = \frac{1}{\mid n \mid} \sum_{i=1}^{|n|} \frac{1}{rank_i} \tag{5.9}$$

Here, the reciprocal rank for a query (bug) is the reciprocal of the position of the correct developer in the returned ranked list of developers ($rank_i$) and $n$ is the total number of bugs in our benchmark. When the correct developer for a bug is not recommended at all, we consider its inverse rank to be

a zero. When there are multiple correct developers (which are a very few cases in our benchmark), we consider the highest/first ranked position. The higher the value of MRR, the better it speaks of the potential effort spent in noise. For example, an MRR value of 0.5 suggests that the approach typically produces the correct answer at the 2nd rank. **Overall, our main research hypothesis is that *iHDev* will outperform the subjected competitors in our study in terms of recall without incurring additional costs in terms of MRR**.

We applied the One Way ANOVA test to validate whether there was a statistically significant difference with $\alpha = 0.05$ between the results of both recall and MRR values. For MRR, we considered the ranks of correct answers of the approaches for each bug (data point). We used this non-parametric test because we did not assume normality in the distributions of the recall results. The purpose of the test is to assess whether the distribution of one of the two samples is stochastically greater than the other. Therefore, we defined the following null hypotheses for our study (the alternative hypotheses can be easily derived from the respective null hypotheses):

- **H-1:** There is no statistically significant difference between the recall@m values of *iHDev* and each of *xFinder*, *xFinder′*, and *iMacPro*.
- **H-2:** There is no statistically significant difference between the MRR values of *iHDev* and each of *xFinder*, *xFinder′*, and *iMacPro*.

### 5.3.5 Results

The recall@1, recall@2, recall@3, and recall@5 values for each of the compared approaches for *Mylyn* and *Eclipse Platform* were calculated from the established benchmarks. Table 5.5 shows the $recall@m$ values of *Mylyn* and *Eclipse Platform* for all the compared approaches (see the Recall@m column). The MRR values for each of the compared approaches for *Mylyn* and *Eclipse Platform* were then calculated. Table 5.6 shows the $MRR$ values of *Mylyn* and *Eclipse Platform* for all approaches (see the MRR column).

We computed the recall gain of *iHDev* over another compared approach (i.e., $Y$) using the following formula:

Table 5.5: Average of recall @1, 2, 3 and 5 values of the approaches *iHDev*, *xFinder*, *xFinder′*, and *iMacPro* measured on the *Mylyn* and *Eclipse Platform* benchmarks.

| $m$ | Recall@m | | | | ***iHDev* vs *xFinder*** | | | ***iHDev* vs *xFinder′*** | | | ***iHDev* vs *iMacPro*** | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | *iHDev* | *xFinder* | *xFinder′* | *iMacPro* | **Gain%** | **Pvalue** | **Adv** | **Gain%** | **Pvalue** | **Adv** | **Gain%** | **Pvalue** | **Adv** |
| ***Mylyn*** | | | | | | | | | | | | | |
| 1 | 0.50 | 0.45 | 0.52 | 0.50 | 11.11 | 0.22 | *None* | -3.84 | 0.58 | *None* | 0 | 0.9 | *None* |
| 2 | 0.71 | 0.63 | 0.59 | 0.63 | 12.69 | 0.02 | *iHDev* | 20.33 | 0.001 | *iHDev* | 12.69 | 0.03 | *iHDev* |
| 3 | 0.79 | 0.68 | 0.69 | 0.72 | 16.17 | 0.001 | *iHDev* | 14.49 | 0.004 | *iHDev* | 9.72 | 0.05 | *iHDev* |
| 5 | 0.86 | 0.69 | 0.81 | 0.76 | 24.63 | ≤0.001 | *iHDev* | 6.17 | 0.08 | *None* | 13.15 | 0.0008 | *iHDev* |
| ***Eclipse Platform*** | | | | | | | | | | | | | |
| 1 | 0.25 | 0.12 | 0.12 | 0.11 | 108.33 | 0.03 | *iHDev* | 108.33 | 0.03 | *iHDev* | 127.27 | 0.02 | *iHDev* |
| 2 | 0.37 | 0.20 | 0.17 | 0.18 | 85.00 | 0.03 | *iHDev* | 117.64 | 0.007 | *iHDev* | 105.55 | 0.01 | *iHDev* |
| 3 | 0.41 | 0.20 | 0.22 | 0.22 | 105.00 | 0.005 | *iHDev* | 86.36 | 0.01 | *iHDev* | 86.36 | 0.01 | *iHDev* |
| 5 | 0.45 | 0.20 | 0.23 | 0.23 | 125.00 | 0.001 | *iHDev* | 95.65 | 0.004 | *iHDev* | 95.65 | 0.003 | *iHDev* |

$$GainR@m_{iHDev-Y} = \frac{recall@m_{iHDev} - recall@m_Y}{recall@m_Y} \times 100 \quad (5.10)$$

The $MRR$ column in Table 5.6 shows MRR values of the compared approaches.

We computed the MRR gain of *iHDev* over another compared approach (i.e., $Y$) using the following formula:

$$GainMRR_{iHDev-Y} = \frac{MRR_{iHDev} - MRR_Y}{MRR_Y} \times 100 \quad (5.11)$$

To answer the research question $RQ$, we compared the recall values of *iHDev*, *xFinder*, *xFinder′*, and *iMacPro* for $m = 1$, $m = 2$, $m = 3$, and $m = 5$. We computed the recall gain of *iHDev* over $Y$ in {*xFinder*, *xFinder′*, and *iMacPro*} using Equation 5.10. Similarly, we compared MRR values of *iHDev*, *xFinder*, *xFinder′*, and *iMacPro*. We computed the MRR gain of *iHDev* over $Y$ in {*xFinder*, *xFinder′*, and *iMacPro*} using Equation 5.11. Table 5.5 and Table 5.6 show the recall and MRR results respectively. The **Gain %** columns in Table 5.5 show the recall gains of *iHDev* over each compared approach for the different $m$ values. The **Gain %** columns in Table 5.6 show the MRR gain of *iHDev* over each compared approach. The **Pvalue** columns in Table

5.5 shows the p-values from applying the One Way ANOVA test on the recall values for $m = 1$, $m = 2$, $m = 3$, and $m = 5$. The **Pvalue** columns in Table 5.6 shows the p-values from applying the One Way ANOVA test on the reciprocal rank values. The **Advantage** columns show the approach that had a statistically significant gain over the other. In cases neither did, **None** is shown.

For each pair of competing approaches, there were eight comparison points in terms of recall values (four each for *Mylyn* and *Eclipse Platform*). Overall, out of eight comparison cases between *iHDev* and *xFinder* for recall values, *iHDev* was advantageous over *xFinder* in seven of them; the remaining one being a statistical tie. Out of eight comparison cases between *iHDev* and *xFinder′* for recall values, *iHDev* was advantageous in six of them; the remaining two being a statistical tie. Out of eight comparison cases between *iHDev* and *iMacPro* for recall values, *iHDev* was advantageous in seven of them; the remaining one being a statistical tie. Despite a few observations of negative gains, there was not even a single case in which other approaches was statistically advantageous over *iHDev* in terms of recall. Therefore, we reject the hypothesis $H_1$.

For each pair of competing approaches, there were two comparison points in terms of MRR values (one each for *Mylyn* and *Eclipse Platform*). Overall, *iHDev* was advantageous in both cases of comparison between *iHDev* and *xFinder*. *iHDev* was advantageous in one case each for comparisons between *iHDev* and *xFinder′*, and between *iHDev* and *iMacPro*. The remaining two cases were a statistical tie. There were no cases in which the other approaches were statistically advantageous over *iHDev* in terms of MRR. Therefore, we reject the hypothesis $H_2$.

In summary, the overall results suggest that *iHDev* generally performs better than *xFinder*, *xFinder′*, and *iMacPro* in terms of both recall and MRR. Using the interaction history typically leads to improvements in accuracy. Not only does it identify the correct developers more often (as evident by the significant recall gains or no loss), but also at a high enough position in the list of recommended candidates (as evident by the significant MRR gains or no loss). For example, *iHDev* recorded recall gains over *xFinder* in the range [85% , 125%] for *Eclipse Platform*. Also, on average, the correct developer would appear at the 3rd position (MRR=0.34) for *iHDev* recommendations, whereas, in *xFinder* this value would be at the 6th position (MRR=0.16). Thus, the

Table 5.6: Mean Reciprocal Rank of the approaches *iHDev*, *xFinder*, *xFinder′*, and *iMacPro* measured on the *Mylyn* and *Eclipse Platform* benchmarks.

| System | MRR | | | | ***iHDev* vs *xFinder*** | | | ***iHDev* vs *xFinder′*** | | | ***iHDev* vs *iMacPro*** | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | *iHDev* | *xFinder* | *xFinder′* | *iMacPro* | **Gain%** | **Pvalue** | **Adv** | **Gain%** | **Pvalue** | **Adv** | **Gain%** | **Pvalue** | **Adv** |
| *Mylyn* | 0.66 | 0.56 | 0.62 | 0.64 | 17.85 | 0.003 | *iHDev* | 6.45 | 0.22 | *None* | 3.12 | 0.4 | *None* |
| Eclipse | 0.34 | 0.16 | 0.16 | 0.17 | 112.5 | 0.005 | *iHDev* | 112.5 | 0.007 | *iHDev* | 100 | 0.01 | *iHDev* |

MRR gain of over 112%. ***iHDev* takes us a step forward toward achieving the ideal goal of a recommender that not only always identifies the correct developers, but also puts them in the top positions.**

### 5.3.6 Discussion

We discuss a few qualitative points that would help understand the rationale behind the improved performance with using interactions in *iHDev*.

**Multiple Attempts at Resolution**. Given the nature of OSS, there are often multiple attempts at resolving a given change request. For example, multiple (a few incomplete or incorrect) fixes are attempted by perhaps multiple developers. In the end, only a complete and correct resolution is accepted and/or merged into the source code repository (i.e., the main development trunk or branch). The interaction history records these attempts; however, the commit history only records the final outcome (i.e., only the things committed), if any. Gousios et al. [43] observed that in GitHub some issues receive multiple pull-requests with solutions (all representing interaction and competence), but not all are accepted and merged. Our results show that past experiences (including failures) are important ingredients in shaping developer expertise. Interactions offer a valuable insight into these micro-level developers' activities in building their expertise, whereas, the commit repositories would largely miss out on them. For example, the file *UpdateAttachmentJob.java* has 458 edit events, but it has only 6 commits. Additionally, the developer *Frank Becker* contributed 50 edit events, but had only one commit. The interaction history shows that he attempted resolutions for 6 bugs, but, the commit shows that he contributed only one bug fix. This example suggests there is quite a disparity between the micro and macro level perspectives of contributions.

**Resolutions and Peer Review**. The practice of peer review of proposed resolutions/patches exist in both *Mylyn* and *Eclipse Platform*. We observed cases in which resolutions (even those that were correct in the sense of a technical fix for the problem at hand) were not merged to the source code repository (e.g., potential conflicts with other code elements or patches did not make it in time for the review process to go through or were out of luck as they arrived after something else was already accepted) or were revised and then merged (e.g., a few changes to a few files revised or not needed). The interaction history captures the resolutions while the commit history misses out on these alternative solutions.

**Interactions come first, Commits later**. In the typical workflow, interactions come first and commits later. Interactions are available much earlier than commits to mine and integrate the results. Our datasets show that for a given time period, there are more interactions than commits. Therefore, using interaction data may reduce the latency in building and deploying actionable models on a project, as the training data would become available sooner.

**Self-Serving Benefits**. We believe that the potential benefits shown to developers by converting the interactions into actionable support for their routine tasks could serve as a motivating factor in using *Mylyn* type activity loggers. That is, developers would see the value in logging their activities now, so that they could benefit for concrete tasks in the future.

**Task Applicability**. Our work shows the potential value of interactions in improving the developer recommendation tasks; however, they could be use for other software maintenance and evolution tasks that have typically relied on commit histories. Previously, we used interactions for impact analysis [15, 112].

In summary, our findings highlight many aspects that the developer recommendation methodology based on the commit history may not capture, but interactions do.

### 5.4 Threats to Validity

We discuss internal, construct, and external threats to validity of the results of our empirical study.

*Accuracy measures and correctness of developer recommendations*: We used the widely

used metric recall in our study. We also calculated mean reciprocal rank. We considered a gold-set to be developers who contributed source code changes to address change requests. Of course, it is possible that other team members are also equally qualified to handle these change requests; however, such a gold-set would be very difficult to ascertain in practice. Nonetheless, our benchmark can be viewed as conservative bounds.

Accuracy@k [108, 23, 94]generally considers only one correct answer to claim 100%. In our study, over 90% of cases had only one correct answer. Therefore, the differences in recall@k and accuracy@k values are negligibly small. Although, (as discussed in Section 5.3.4) we deemed the precision metric to be unsuitable for our technique due to the ordered nature of responses, we summarize the results. In case of *Mylyn* and *xFinder* precision gain ranges between 12.76% and 136.36% and out of eight comparisons with ANOVA, *iHDev* was advantageous in four of them. In case of *Mylyn* and *xFinder'* precision gain ranges between 12% and 137.50 % and out of eight comparisons with ANOVA, *iHDev* was advantageous in six of them. In case of *Mylyn* and *iMacPro* precision gain ranges between 6% and 136.36 % and out of eight comparison with ANOVA, *iHDev* was advantageous in seven of them. Furthermore, should there be a larger number of cases with multiple correct answers, a metric such as Mean Average Precision (MAP) could be employed.

*Machine Learning-based matching of change requests to relevant files*: KNN sometimes returned classes (i.e., files) that were not found in the commits related to the bug fixes or change request implementations. However, based on our prior work we observed that the files that were recommended as textually similar were either relevant (but not involved in the change that resolved the issue) or conceptually related (i.e., developers were also knowledgeable in these parts).

*Narrow Ground Truth:* Our ground truth included only the developers who eventually resolved the change request. It is likely that other developers are equally capable of resolving the same requests. Identifying them is not obvious from repositories and would require a quantitative study.

*Developer identity mismatch*: Although we carefully examined all of the available sources

of information in order to match the different identities of the same developer, it is possible that we missed or mismatched a few cases.

*Incomplete or Missing Interaction History*: Although, a common period was considered for extracting the interaction and commit datasets, the number of commits is significantly higher than the number of interaction transactions. This difference is not necessarily the result of a single task defined for multiple commits, because there are many cases in which committed files were never part of one of the interactions.

*Explicit Bug ID Linkage*: We considered interactions and commits to be related if there was an explicit bug ID mentioned in them. Implicit relationships were not considered.

*Two Systems Considered*: Due to the limited availability of Mylyn interaction histories, our study was performed on only two systems written in Java. *Mylyn* and *Eclipse Platform* were the largest available datasets within the Eclipse Foundation. Nonetheless, this fact may limit the generality of our results.

# CHAPTER 6

## Developer Recommendation (*rDevX*)

The practice of code review of patches submitted for software maintenance is common in open source and commercial domains. There is empirical evidence supporting many of its benefits, including in improving software quality and knowledge transfer. In this chapter, we investigated the research question ”what can historical records of code reviews tell us about developer expertise?”. We analyzed code reviews that are managed by ”modern” tools, such as *Gerrit*. We found eight markers of developer expertise associated with the source code changes and their acceptance, time line, and human roles and feedback involved in the reviews. We formed a developer-expertise model from these markers and showed its application in bug triaging. Specifically, we derived a developer recommendation approach for an incoming change request (e.g., to fix a bug), named *rDevX*, from this expertise model. An empirical study on open source systems *Eclipse Platform*, *Mylyn*, and *OpenStack Nova* was conducted to assess the effectiveness of *rDevX*. A number of change requests were used in the evaluated benchmark. Furthermore, a comparative study on another previous approach that use commit history for developer recommendation was performed. The metrics recall and MRR (Mean Reciprocal Rank) were used to measure their quantitative effectiveness. Results show that *rDevX* outperforms the subjected competitor with statistical significance. We also performed a qualitative analysis to ratiocinate the gains of *rDevX*.

### 6.1 Introduction

Software engineering is very much human driven and prone. The quality and velocity of maintenance and evolution tasks (e.g., bug fixes or feature implementations) are in many ways a direct reflection of the individuals or teams who perform them. Determining the right solution providers for the task at hand is arguably as important as suggesting the right tool support for it, especially in the far too commonly found state of inadequate or obsolete documentation of large scale software systems. The expressiveness and effectiveness of the artifacts could be directly at-

tributed to the expertise of its authors. Questions about a specific source code unit from newcomers to a project may be best answered by that code's developers or owners [13]. The developers who have the needed expertise to a change request should be assigned to resolve it [8]. Unfortunately, the knowledge or expertise of developers is oftentimes not readily documented or easily available in large, distributed projects such as open source software development.

There has been previous efforts to build models of developer expertise in source code [9, 54, 65, 94, 107, 111, 46]. Most of these models use historical change records of source code captured in software repositories (bug reports and/or commits). Recently, there have been efforts to build knowledge models from developer interactions that go beyond a change event (e.g., navigation) captured in integrated development environments. Although these models capture aspects related to developer activity, they cannot and do not include other important markers of expertise.

Modern Code Review (MCR) [13], which is tool driven and assisted, has been found to be useful in not only improving software quality, but also as a means of knowledge transfer and collaboration [13]. The historical code reviews capture many unique aspects that can be useful markers of developer expertise. Among them is the reviewer role and contributions in terms of the code critique and formative experience in authoring code changes, which may or may not have been successful (i.e., eventually included in the project code base). Code reviews subsume the commits in source-code repositories (e.g., *git* or *Subversion*). We posit that these markers provide a unique opportunity to expand the scope of forming models of developer expertise in source code and their improved applications in software maintenance and evolution tasks.

**Our Contribution:** In this chapter, we investigated the research question *"how to transform the historical records of code reviews into a developer expertise model, which can be used in the software-change-request-resolution phase?"*. Approaches for recommending reviewers from analyzing code reviews have been proposed before [110, 96, 98, 14]; however, developer expertise models and automatically recommending developers for change-request, e.g., bug, triaging using code reviews have not been investigated yet. Our work makes the following noteworthy contributions:

*Developer Expertise Model*: We identify eight markers of developer expertise in source code from analyzing past code reviews – *expert-markers*[1]. The markers are associated with the source code changes and their acceptance, time line, and human roles and feedback involved in the reviews. We then unify these markers to form a developer expertise model in source code.

*rDevX– Developer Recommendation Technique*: We show the application of our developer-expertise model for the task of developer recommendation, i.e., automatically assigning issues or change requests to the developer(s) who are most likely to resolve them. *rDevX* takes the textual description of a change request (e.g., a short bug description) and recommends a ranked list of developers to resolve it.

*Empirical Evaluation*: To evaluate the accuracy of *rDevX*, we conducted an empirical study on three open source systems *Eclipse Platform*, *Mylyn*, and *OpenStack Nova*. Recall and Mean Reciprocal Rank (MRR) metric values of the developer recommendations on a number of bug reports sampled from these systems are presented. Our *rDevX* approach is empirically compared with an approach based on the commit history [54]. The results show that the presented *rDevX* approach outperformed its competitor. *rDevX* registers gains as much as 75% over the other approach. These gains came without incurring any decreased Mean Reciprocal Rank (MRR) values; rather *rDevX* recorded improvements in them. That is, *rDevX* would typically recommend the correct developers at higher ranks than the competitor.

The rest of this chapter is organized as follows: Our presented developer expertise model is discussed in Section 6.2 and its application in developer recommendation, i.e., *rDevX*, is presented in Section 6.3. The empirical study is presented in Section 6.4. Threats to validity are discussed in Section 6.5.

## 6.2 The Developer Expertise Model

We first provide our rationale for going beyond developer expertise models formed from macro-repositories such as commit archives and then discuss our presented model based on historical code reviews.

[1] aspired to be in spirit of biomarkers

### 6.2.1 Why Code Reviews to Build a New Model for Developer Expertise?

Numerous approaches have been proposed to formulate the expertise of developers in source code from software repositories [9, 107, 47, 7, 46, 16, 8, 10, 111]. Source-code-version archives, i.e., commits, are the most commonly analyzed repositories for this purpose. The underlying premise of these approaches is typically that developers who contributed changes to specific source code entities are knowledgeable or experts of those entities. Their expertise is predominately determined from their commit activity and their roles as code owners or authors or maintainers [46]. Although, developer expertise modeled in such a way provides several important aspects, we posit that it is limited in scope and size. Are there other markers that demonstrate developers gaining expertise in particular source code entities beyond their record of the code owner role and accepted changes that are eventually merged to the code base, i.e., commits? Commit repositories only capture the *end points, i.e., macro events,* of software maintenance or evolution tasks (e.g., accepted code changes for bug fixes or feature requests), and do not necessarily capture *the means, i.e., micro events,* associated to achieve them. The question is where to find the records of these *means* and how could they help in forming an enhanced and more effective developer expertise model than those formed from the archives of *ends*?

Several open source projects and commercial companies practice modern code review. *Gerrit* is one popular tool that facilities MCR. Previous empirical studies show that the code review process helps developers spread the knowledge across the development team [81, 82, 13]. A senior developer at Microsoft made the following remark:

> *one of the things that should be happening with code reviews over time is a distribution of knowledge* [13].

Bacchelli et al. [13] similarly state that *Team Awareness* and *Shared Code Ownership* are other aspects of the code review process at Microsoft. These findings suggest that the benefits of MCR extend beyond finding and reducing defects.

We resort to the code-review archives to detect markers of developer expertise in source code. Modern Code Review (MCR), which is tool assisted, supports the quality-control process of

reviewing a proposed code change or patch for a maintenance task. It includes human feedback, which drives whether a proposed code change needs to be revised, and eventually accepted or abandoned. Only accepted source code changes are merged to the code base. That is, it is one of the sources that captures *the means* adopted in reviewing and revising not only the patches that eventually get merged (i.e., committed), but also those abandoned. We explore code reviews, especially the proposed changes (regardless of them committed or not), human roles and feedback. These aspects are not captured in commit repositories. Next, we discuss the exclusive markers (measures) that can be obtained from the code-review archives compared to those from commit archives.

*Owner and Reviewer Roles*: In MCR, humans have two main roles: 1) The *owner* is the one who submits the proposed code change or patch for review and 2) A *reviewer* is the one who reviews the proposed change, provide feedback on it, and/or accept it. Typically, there is one code owner and multiple reviewers for a proposed change. At least one reviewer must accept the change for it to be merged/committed to the project code base or else it is abandoned. Is there a difference in the sets of owners and reviewers, and their code expertise? If there is none, the corresponding commit and code-review archives should produce the identical set of human contributors. That is, both sources provide equivalent information, and perhaps the commit archives are a sufficient source for expertise.

We investigated three open source systems, *Eclipse Platform*, *Mylyn*, and *OpenStack Nova* to investigate this issue. For each system, we obtained the set of authors from the commit, and the set of owners and reviewers from the code-review archives. We calculated the Jaccard similarity between these sets. The Jaccard coefficient measures the similarity between sample sets from their intersection size divided by their union size. The Jaccard values for *Eclipse Platform*, *Mylyn*, *OpenStack Nova* are 0.45, 0.67,and 0.48 respectively. The low jaccard similarity between two sets of reviewers and owners show the difference between the sets. In other words, how dissimilar both the sets are. This observation shows that a role change could occur. For example, a reviewer of code today may turn into its author tomorrow. We also investigated the historical records of

files in both commit and code review repositories for these systems. For each system, we obtained the set of unique files from the commit and the set of unique files from the code-review archives. We calculated the Jaccard similarity between these sets. The Jaccard values for *Eclipse Platform*, *Mylyn*, *OpenStack Nova* are 0.55, 0.55, and 0.60 respectively. These values show that these two sets are quite dissimilar. This observation shows that there is potential for broader scope of source code files available in code review than commit archives.

*Reviewer to Author Role Change*: The role of a reviewer is not always confined to providing feedback or suggesting the improvements to or critiquing the patch. It is possible that during the code review process, a reviewer ends up taking the mantle of ownership. For example, consider the review # 62379 [2] in *Eclipse Platform*. The owner of the patch under this review is *Xavier Coulon*[3]. *Jeff Johnston* and *Roland Grunberg* provided review comments for this patch. *Roland Grunberg* offered suggestions on the file *DockerContainersView.java*. *Jeff Johnston* implemented those suggested changes and submitted the second, revised patch, which was accepted. This case suggests that others who may have only the review knowledge about the source code can start contributing as authors. Again, this aspect of role changes is not documented in commit archives. The knowledge garnered in reviewing code can later be used in its authorship, as early as in the same patch.

*Review Before Commit*: Studies show that the code review process help in knowledge transfer [81], [82], [13]. New contributors are initiated to the software development via the code review process. They first participate as reviewers and observe how other more experienced owners and reviewers conduct their activities. After they gain enough knowledge in this incubation period, they also become code contributors. For example the developer *"Niraj Modi"* started his first review experience in the source code file *"Control.java"* from the package *swt* of *Eclipse Platform* in the review # 28462[4]. He actually made an in-line comment for the reviewed changes in the file *"Control.java"* on June 13, 2014. Several months later on January 23, 2015, he submitted his fix

[2]https://git.eclipse.org/r/#/c/62379/
[3]We use the names from publicly available data
[4]https://git.eclipse.org/r/#/c/28462/

for bug # 435384 which included a change to the file *"Control.java"*. This case suggests that the code-review archive may capture the developer expertise on specific source code elements much earlier than the commit archives.

*Abandoned Patches*: The code changes that are accepted in code review are eventually committed to the code base. Thus, they are captured in the source-code version archives. Not all the submitted patches are eventually accepted (see Table 6.5). The abandoned patches are not recorded in the commit repositories. The question is what could the abandoned patches tell us about the developer expertise? It should be noted that not all patches are abandoned due to the inferior expertise of developers or their suboptimal solutions, i.e., they are perceived to be of low quality (defect prone or coding style noncompliance). In some instances, they are abandoned due to timing or feature-priority issues. Nevertheless, even when patches are abandoned due to their lack of quality, they maybe indicators of developers gaining expertise for future success, i.e., their patches in the future are accepted. Thus, abandoned patches may embed indicators of equivalent expertise of other developers or breeding grounds for developers.

We manually examined several abandoned patches and observed interesting characteristics. Two developers submitted two separate patches in a close time proximity. One got accepted and the other was left abandoned after a successful build. The patch submitted in review #60129[5], which is related to bug #475941, in *Eclipse Platform* was abandoned. Whereas another patch submitted in review #60243[6] was merged.

In another instance, two developers submitted two different patches, and one was abandoned not necessarily because of its poor quality in terms of verification and validation, but due to the preferred design of the other[7] (which perhaps indicates a better internal quality). We also found a case in which one of developers submitted a patch for an enhancement request (bug#398110[8])

[5]https://git.eclipse.org/r/#/c/60129
[6]https://git.eclipse.org/r/#/c/60243/
[7]https://git.eclipse.org/r/#/c/35282/
[8]https://bugs.eclipse.org/bugs/show_bug.cgi?id=398110

and then it was determined that the enhancement was not needed[9]. These observations suggest that abandoned patches are a valuable source for inferring developer expertise.

There have been attempts to model developer expertise from records of interactions captured in IDEs [111, 39]. It should be noted that interactions capture only the code owner's activities involved in resolving and implementing of an issue, and not those of reviewers and owner-reviewer discussion and discourse. Similarly, bug repositories [8, 107] typically record the front stages of a change request, i.e., from reporting to resolving it. For example, a bug report may include comments on clarifying or additional information on a defect. This information may help design and implement the needed changes to address it, but usually not to review them.

### 6.2.2 Markers and Expertise Model

The discussion in the preceding section suggests that code review archives provide a valuable source for inferring developer expertise in source code. We considered markers (measures) of developer expertise associated with the source code changes and their acceptance, and human roles and feedback involved in the reviews. These markers are lightweight, yet capture important dimensions of expertise, which could make maintenance tasks effective. We considered two types of measures: 1) frequency and 2) recency. Given a unit of source code, e.g., the file $f$, determine the expertise of the contributor $d$ in it. Our model unifies the markers and provides the relative expertise of a particular contributor in a specific source code unit. We next describe the markers and the model.

*Patches Submitted*, $N_f$: We calculate the contributor $d$'s total number of *submitted* patches that include the file $f$. That is, the contributor $d$ is in the owner's role. This metric is frequency based.

Previous studies show that the recency of contributions is an important and influential factor [111, 54]. It helps provide a more favorable weight to relevant contributions than irrelevant (much older) ones. Therefore, we also consider the date of the last submitted patch.

[9]https://git.eclipse.org/r/#/c/9891/

*Last Submitted Patch Date*, $D_f$, is the date of the contributor *d*'s last *submitted* patch that include the file *f*.

*Patches Accepted*, $P_f$: We calculate the contributor $d$'s total number of *accepted* patches that include the file $f$.

*Last Accepted Patch Date*, $A_f$, is the date of the contributor *d*'s last *accepted* patch that include the file *f*.

*Review Comments on Patches Submitted*, $T_f$: We calculate the contributor $d$'s total number of comments written for the *submitted* patches that include the file $f$. That is, the contributor $d$ is in the reviewer's role. This metric is frequency based.

*Last Comment Date on Submitted Patch*, $L_f$, is the last comment date of the contributor *d* on the *submitted* patches that include the file *f*.

*Review Comments on Patches Accepted*, $R_f$: We calculate the contributor $d$'s total number of comments written for the *accepted* patches that include the file $f$.

*Last Comment Date on Accepted Patch*, $W_f$, is the last comment date of the contributor *d* on the *accepted* patches that include the file *f*.

We used the above eight measures to determine the developer's expertise in a specific source code file, i.e., *developer-review* map. The *developer-review* map, *DR* for the developer *d* and file *f* is formulated into a vector, which is given below:

$$DR_{(d,f)} = \langle P_f, A_f, N_f, D_f, R_f, W_f, T_f, L_f \rangle$$

Note that there is one to many relationship between a given source code entity and developers, i.e., multiple developers maybe knowledgeable in it. To account for this issue, we also compute the equivalent eight measures for a specific source code entity and represent them into a vector form. The *file-review* map *FR* represents the review contributions to the file *f*, and is shown below:

$FR_{(f)} = \langle P'_f, A'_f, N'_f, D'_f, R'_f, W'_f, T'_f, L'_f \rangle$, where

- $P'_f$ is the total number of patches that got accepted, that includes the file *f*.
- $A'_f$ is the date of last accepted patch that include the file *f*.

- $N'_f$ is the total number of patches that got submitted, that includes the file *f*.
- $D'_f$ is the date of last submitted patch that include the file *f*.
- $R'_f$ is the total number of review comments written on accepted patches that include the file *f*.
- $W'_f$ is the date of last review comments written on accepted patches that include the file *f*.
- $T'_f$ is the total number of review comments written on submitted patches that include the file *f*.
- $L'_f$ is the date of last review comments written on submitted patches that include the file *f*.

The contribution or summaries of expertise, termed *SoE*, for the developer *d* and the file *f* is computed using the sum of the ratios of the individual features from the *developer-review* and its respective feature from the *file-review* maps. That is, the *file-review* map serves as a normalizing factor. The equation to calculate the total score of a developer for every file, in other words, the *SoE*, is calculated in two steps: the total of recency factors and the total of frequency factors for every developer with respect to the source code file. The total of recency factor is the sum of the scores of all the date related features and the total of frequency factor is the sum of all the frequency related features. The score for each date feature is calculated as follows:

*Y(d,f)* is a generic meta-representation of how the score for all the date related features are calculated.

$$Y(d,f) = \begin{cases} \frac{1}{|y_f - y'_f|} & \text{if } |y_f - y'_f| \neq 0 \\ 1 & \text{if } |y_f - y'_f| = 0 \end{cases} \tag{6.1}$$

The term $y_f$ and $y'_f$ are numerical representation of date values and hence the term $y_f - y'_f$ would also be numerical. Here, the term $y$ can take the value of $A_f$, $D_f$, $W_f$, or $L_f$. The term $y'$ can take the corresponding value of $A'_f$, $D'_f$, $W'_f$ or $L'_f$.

*TD(d,f)* is the total of recency factors, i.e., the score corresponding to all date related features:

$$TD(d, f) = \sum_{i=1}^{4} Y_i(d, f) \tag{6.2}$$

*Y(d,f)* is calculated as in equation (1), for all the date related features i.e. $A_f$,$A'_f$, $D_f$, $D'_f$, $W_f$, $W'_f$ and $L_f$, $L'_f$. Here, the value of *i* ranges from one to four as there are four groups of date related features.

*PD(d,f)* is the total of frequency factors, i.e., the score corresponding to all patch and contributor related features:

$$PD(d, f) = \left\{ \frac{P_f}{P'_f} + \frac{N_f}{N'_f} + \frac{R_f}{R'_f} + \frac{T_f}{T'_f} \right. \tag{6.3}$$

The final equation of *SoE(o,f)* is given as the sum of the total of frequency and recency factors:

$$SoE(d, f) = TD(d, f) + PD(d, f) \tag{6.4}$$

According to Equation 6.4, the maximum value of *SoE* can be eight because we have used eight measures. It should also be noted that these measures can be computed for any given historical window, and not necessarily on the entire historical data (e.g., to form an expertise model at a given point of time from prior contributions – see Section 6.3 ). Although, we described our model at the file level, it can be extrapolated to higher or lower levels of granularity. For high-level entities (e.g., the package containing a particular file), we compute the measures for all its files and aggregate them to determine the *SoE* value. For lower-level granulates (e.g., classes and methods), we need to further process the file and line changes, i.e., it requires additional cost. The measures and *SoE* value are then computed for each entity at that level.

### 6.3 Application for Recommending Appropriate Developers in Change Request Triaging

We show the application of our developer expertise model from code reviews for the developer recommendation task in bug triaging. Developer recommendation is the assignment of

an incoming change request (e.g., a bug report or feature request) to the developers who are most likely to address it. Our presented approach, named *rDevX*, consists of the following steps:

**Locating Relevant Entities to Change Request**: We use the K-Nearest Neighbor (KNN) algorithm to locate relevant units of source code (e.g., files and classes) that match the given textual description of a change request or reported issue. The indexed source code release/snapshot is typically the one in which an issue is reported for the first time (i.e. the issue is not resolved in that revision). We impose this criterion because we cannot use any forward looking information in training our model.

**Recommending developers based on the SoE Model**: For each of the relevant files from the above step, we compute its *SoE* score using our developer expertise model from the reviews contributed prior to the snapshot time in the above step. Again, we impose this criterion because we cannot use forward looking reviews to train our model. We use a ranking mechanism from these scores to arrive at a ranked list of recommended developers.

### 6.3.1 Locating Relevant Entities to Change Request

In our approach, we use techniques from natural language processing and machine learning to locate textually relevant source code files to a given change request. The specific steps are given below:

#### Creating a corpus from software

The source code of a release, in or before which the change request is resolved, is parsed using a developer-defined granularity level (e.g., file) and documents are extracted. A corpus is created such that each file will have a corresponding document therein. Identifiers, comments, and expressions are extracted from the source code. Each document in the corpus is referred to as $doc_{past}$.

#### Preprocessing Step

Each document $doc_{past}$ is preprocessed in two parts: removing stop words and stemming.

**Document-Term Representation**

We produce a dictionary from all of the terms in our document and assign a unique integer Id to each term appearing in it.

`Term Weighting`: We use *tfidf* in our approach. The global importance of a term *i* is based on its inverse document frequency (*idf*), calculated as the reciprocal of the number of documents that the term appears in. *idf* is the document frequency of term *i* in the whole document collection. $tf_{i,j}$ is the term frequency of the term *i* in the document *j*. Each document $d_j$ is represented as a vector $d_j = (w_{1,j}, ..., w_{i,j}, ..., w_{n,j})$ where $n$ is the total number of terms in our document collection and $w_{i,j}$ is the weight of the term $i$ in document $j$.

**Using Change Request**

The textual description of a change request for which we want to find all of the relevant files, and eventually to be assigned developer(s), is referred to as $doc_{new}$. $doc_{new}$ also goes through the preprocessing step and is represented as a document.

**K-Nearest-Neighbor**

The task of multi-label classification is to predict for each data instance a set of labels that applies to it. Standard classification only assigns one label to each data instance. However, in many settings a data instance can be assigned by more than one label. In our context, each data instance (i.e., a bug report) can be assigned multiple labels (i.e., source code files). ML-KNN is a state-of-the-art algorithm in the multi-label classification literature [113]. We employ the ML-KNN search with a user-defined value of $K$ (e.g., the value of 10 as used in previous work [107]) to search the existing corpus ($doc_{past}$) based on similarities with the new bug description ($doc_{new}$). This search finds the top $K$ similar files. We consider these top $K$ files to be a set, i.e., they are eventually relevant to the change request regardless of their similarly levels or ranking. Cosine is then used to measure the similarity of the two document vectors.

$$f_{sim}\left(doc_{new}, doc_{past}\right) = \frac{doc_{new}.doc_{past}}{|doc_{new}||doc_{past}|} \tag{6.5}$$

The conclusion of this step identifies the $K$ most relevant source code files to the given change request. Now, we need to find the developer expertise using the *SoE* scores.

### 6.3.2 Recommending developers based on SoE scores

We now describe how the ranked-list of developers is obtained from all of the scored contributors of each relevant source code file to a given bug. Each contributor can be an author or a reviewer. Each file $f_i$ may have multiple contributors; however, it is not necessary for all of the files to have the same number of contributors. For example, the file $f_1$ could have two contributors and the file $f_2$ could have three contributors. Each row in the matrix $D_c$ (see Equation 6.6) gives the list of unique contributors for each relevant file $f_i$. $D_{cf_i}$ represents the set of contributors, with no duplication, for the file $f_i$, where $1 \leq i \leq n$ and $n$ is the number of relevant files. $c_{ij}$ is the $j^{th}$ contributor in the file $f_i$ with $l$ unique contributors.

$$D_c = \begin{pmatrix} f_1 & D_{cf_1} \\ f_2 & D_{cf_2} \\ .. & .. \\ .. & .. \\ f_n & D_{cf_n} \end{pmatrix} \quad D_{cf_i} = \{\, c_{i1}\ c_{i2}\ \ldots\ c_{il} \,\} \tag{6.6}$$

Although, a single file does not have any duplicate contributors, two files may have common contributors. In Equation 6.7, $D_{cu}$ is the union of all unique contributors from all relevant files.

$$D_{cu} = \bigcup_{i=1}^{n} D_{cf_i} \tag{6.7}$$

$$Score(c) = \sum_{i=1}^{n} SoE_i(c, f_i) \tag{6.8}$$

Each contributor $c$ for a file $f$ has the $SoE$ score. To obtain the likelihood of the contributor $c$, i.e., $Score(c)$, to resolve the given change request, we sum $SoE$ scores of the relevant files in which contributor $c$ appears (see Equation 6.8). That is, their overall expertise in all the files is computed collectively. The $Score(c)$ value is calculated for each unique contributor $c$ in the set $D_{cu}$. In Equation 6.9, we have a set of candidate developers. The developers in this set are ranked based on their $Score(c)$ values. Once the developers are ranked in the descending order of their $Score(c)$ values, we have a ranked list of candidate developers. By using a cutoff value of $m$, we

recommend the top $m$ candidate developers, i.e., with top $m$ $Score(c)$ values, from the ranked list obtained from the set $DF$.

$$DF = \{(c, Score(c)), \forall c \in D_{cu}\} \tag{6.9}$$

This step concludes *rDevX* and we have the top $m$ candidate developers recommended for the given change request. If we do not get any recommendations, we resort to the package level or higher.

### 6.3.3 An Example from *Mylyn*

We demonstrate *rDevX* using an example from *Mylyn*. The change request of interest here is the bug #428544[10] *"Review Dashboard selection for a Gerrit server"*. We first collected a snapshot of *Mylyn*'s source code prior to this bug fixed and then parsed it using the file-level granularity (i.e., each document is a file). After indexing with a machine learning technique, we obtained a corpus consisting of 1,825 documents and 201,554 words. A search query was then formulated using the bug's textual description, the result of which (i.e., top 10 relevant files) is summarized in Table 6.1. These ($k = 10$) files are fed to the second step, *rDevX*. The correct developer who fixed this bug and committed the change is *Jacques Bouthillier*. In Table 6.1, the second column shows a set of all developers with the *SoE* values for each file $f_i$.

In *rDevX*, we first obtained the set $D_{cu}$ from all of the developers recommended for each relevant file $f_i$ to the bug #428544 in Table 6.1. The set $D_{cu}$ consists of 9 unique developers. Because a developer could use different identities for the owner and reviewer roles, we normalized them to a single identity, which was their full name. For each of the 9 unique developers, the $Score$ value is calculated according to Equation 6.8. Table 6.2 shows the top five $Score$ values and the corresponding developers, i.e., $m = 5$. *Jacques Bouthillier* has the highest score in the set $DF$ (a value of 30.48), so he is ranked first. For the remaining developers, the value of the function $Score$ is less than *Jacques Bouthillier*'s score, so they all have a rank greater than 1.

Table 6.3 shows the results for the approaches *rDevX*, *xFinder* and *DevCom* (described in sections 6.4.1) for $m = 5$. Clearly, the best result is for *rDevX* and *DevCom*, as they recommend

[10]https://bugs.eclipse.org/bugs/show_bug.cgi?id=428544

Table 6.1: The developers extracted with *rDevX* from each of the top ten files relevant to Bug# 428544.

| **Files** | **Ranked developer based on *SoE* value** |
|---|---|
| …/SelectReviewSiteHandler.java | Francois Chouinard:0.43, Steffen Pingel:2.00, Jacques Bouthillier:6.08, Marc-Andre Laperle:0.86 |
| …/AddGerritSiteHandler.java | Francois Chouinard:0.36, Steffen Pingel:2.00, Jacques Bouthillier:6.01, Guy Perron:0.44, Marc-Andre Laperle:0.86 |
| …/GerritOperationFactory.java | Steffen Pingel:4.51, Miles Parker:3.01, Tomasz Zarna:0.16 |
| …/AdjustMyStarredHandler.java | Francois Chouinard:1.23, Steffen Pingel:2.00, Jacques Bouthillier:5.36, Marc-Andre Laperle:0.86 |
| …/BuildServerValidator.java | .... |
| …/R4EGerritQueryUtils.java | Francois Chouinard:6.09, Jacques Bouthillier:1.04 |
| …/AllOpenReviewsHandler.java | Francois Chouinard:0.24, Steffen Pingel:2.00, Jacques Bouthillier:6.25, Marc-Andre Laperle:0.86 |
| …/RefreshConfigRequest.java | Steffen Pingel:1.00, Miles Parker:3.00, Tomasz Zarna:0.16 |
| …/R4EGerritServerUtility.java | Francois Chouinard:0.70, Jacques Bouthillier:5.74 |
| …/GerritUiPlugin.java | Sebastien Dubois:0.05, Sam Davis:1.05, Steffen Pingel:1.31, Miles Parker:5.93, Tomasz Zarna:1.17 |

*Jacques Bouthillier* (the correct developer who fixed bug #428544) in the first position, whereas *xFinder* recommends *Steffen Pingel*. The original developer, *Jacques Bouthillier* who fixed the bug was not recommended by *xFinder*. Clearly, *rDevX* outperforms its competitor with respect to recall@1, recall@2, recall@3, and recall@5 values.

Table 6.2: Top five developers recommended to resolve bug #428544 by *rDevX*.

| **Owner** | ***Score*** | **Rank** |
|---|---|---|
| *Jacques Bouthillier* | 30.48 | 1 |
| *Steffen Pingel* | 14.82 | 2 |
| *Miles Parker* | 11.94 | 3 |
| *Francois Chouinard* | 9.05 | 4 |
| *Marc-Andre Laperle* | 3.44 | 5 |

Table 6.3: Top five recommended developers and their associated ranks for the compared approaches.

| textbfApproach | Top five recommended developers in ranked order |
|---|---|
| *rDevX* | Jacques Bouthillier ①, Steffen Pingel ②, Miles Parker ③, Francois Chouinard ④, Marc-Andre Laperle ⑤ |
| *xFinder* | Steffen Pingel ①, Miles Parker ② |
| *DevCom* | Jacques Bouthillier ①, Steffen Pingel ② ,Miles Parker ③, Francois Chouinard ④, Marc-Andre Laperle ⑤ |

## 6.4 Case Study

We assess our developer-expertise model derived from code-review archives in terms of its effectiveness in developer recommendation, i.e., we evaluate *rDevX*. Thus, the purpose of this study was to investigate how well our *rDevX* approach recommends expert developers to assist with incoming change requests, especially when compared to a commit-based approach (see Section 6.2.1 for the commit and code review comparison). We addressed the following research question:

**RQ**: How do the accuracies of *rDevX* (trained from the code review history), *xFinder* (trained from the commit history [54]), and *DevCom* (trained from a combination of the code review and commit histories) compare in recommending code reviewers?

Because *xFinder* is an approach that used developer expertise features only from commit history and its expertise markers used bear a close resemblance to the ones we used from code reviews in *cHRev*, we chose to compare our approach with *xFinder*. Additionally, we had access to its tool support. We also consider another approach DevCom which is a combination of developer expertise features from both commit and code review history. The idea of comparing our approach with *xFinder* and *DevCom* was to investigate how expertise built from the code review history compare with the commit history. Additionally, does their combination provide for a more effective developer-expertise model than considering them individually? Next, we elaborate on *xFinder* and *DevCom*.

### 6.4.1 *xFinder*

*xFinder* builds the developer expertise based on the number of commits, and their number of workdays and recency. Similar to *rDevX* it adds the scores for each developer. It considers the actual developers or authors of changed code in commits (and not their committers). In a *git* repository, there are separate fields for authors and committers. These fields are populated automatically from *Gerrit* after the patch gets merged to the *git* repository. For example, in review # 279312[11] for *OpenStack Nova*, the author in the *git* repository is copied from the owner of the patch in *Gerrit*, who in this case was *Dan Smith*. The committer was *Jay Pipes*. Therefore, this patch/commit is attributed to *Dan Smith* and not *Jay Pipes* in *xFinder*. *xFinder* was shown to be competitive with other developer recommendation approaches, including those that use bug repositories [54].

*xFinder* uses three measures, the number of commits, their number of workdays and recency, to determine the developers who are more likely to be experts in a specific source code file, i.e., *developer-commit* map. The *developer-commit* map, *DC* for the developer *d* and file *f* is given below:

$$DC_{(d,f)} = \langle I_f, C_f, M_f \rangle$$

- $I_f$ is the number of commits that include file *f* and are done by the developer *d*.
- $C_f$ is the number of workdays in the commit activity of developer *d* that include the file *f*.
- $M_f$ is the most recent workday in the commit activity of the developer *d* that includes the file *f*.

Similarly, the *file-commit* map *FC* represents the commit contribution to the file *f*, and is shown below:

$$FC_{(f)} = \langle I'_f, C'_f, M'_f \rangle, \text{ where}$$

- $I'_f$ is the number of commits that include file *f*.

[11] https://review.openstack.org/#/c/279312/

- $C'_f$ is the total number of workdays in the commit activity of all developers that include file *f*.

- $M'_f$ is the most recent workday with a commit that includes the file *f*.

The measures $I_f$, $C_f$, and $M_f$ are computed from the commit history. The contribution or expertise factor, termed *xFactor*, for the developer *d* and the file *f* is computed using the ratios of the *developer–commit* and *file–commit* maps. The contribution factor, *xFactor*, is given below:

$$xFactor(d, f) = \frac{DC_{(d,f)}}{FC_{(f)}} \tag{6.10}$$

$$xFactor(d, f) = \begin{cases} \frac{I_f}{I'_f} + \frac{C_f}{C'_f} + \frac{1}{|M_f - M'_f|} & \text{if } |M_f - M'_f| \neq 0 \\ \frac{I_f}{I'_f} + \frac{C_f}{C'_f} + 1 & \text{if } |M_f - M'_f| = 0 \end{cases} \tag{6.11}$$

We first need to locate relevant source code files to the change request. This step is similar to the first step of *rDevX*, which is explained in Section 6.3.1. The *xFactor* score then is computed for each of these relevant source code files. According to Equation 6.11, the maximum value of *xFactor* can be three because we have used three measures, each of which can have a maximum contribution ratio of 1. We used the same approach of *cHRev* (See Section 6.3.2) to obtain the ranked-list of developers from all of the scored contributors of each relevant source code file to a given bug. The only difference is that we replaced the *SoE* score with the *xFactor* score in Equation 6.8.

### 6.4.2 *DevCom*

The presence of orthogonality between different sources have been leveraged in several other software engineering tasks previously [41], which served as an inspiration to emulate the combination for the developer recommendation task. To assess the potential orthogonally between the commits and reviews, we devised a combined approach, namely *DevCom*, which is based on the factors of *rDevX* and *xFinder*. Similar to *rDevX* and *xFinder*, we first locate relevant source code files to the change request. This step is similar to the first step of *rDevX* (see Section 6.3.1). For each of these source code files, *DevCom* considers eight metrics from reviews and another

three from commits. That is, for each source code file we calculate the sum of its relevant *SoE* score (Equation 6.4) from the code review history and its relevant *xFactor* score (Equation 6.11) from the commit history. The maximum value of the cumulative sum can be 11 because we have used totally 11 measures, each of which can have a maximum contribution ratio of 1. We used the same approach from Section 6.3.2 to obtain the ranked-list of developers from all of the scored contributors of each relevant source code file to a given bug. The only difference is that we replaced *SoE* score with the calculated cumulative score in Equation 6.8. The commits and code reviews are managed in two different systems (git and *Gerrit*). As such, there might be two different identities of the same developer. We did string matching with their credentials (e.g., names) between the two systems in case of an id mismatch to ascertain if they were different or same developers. This automatic step was augmented with a manual examination.

Developer interactions captured in IDEs is another source that has been used for developer expertise models [111, 39] and recommendations [112, 86, 64]. We could not include this approach in our comparison because of the lack of availability of such interaction repositories in the open source domain. *Mylyn* and *Eclipse Platform* projects which did collect developer interactions extensively no longer seem to do so. Only 120 bug reports from 2013-04-08 to 2016-04-08 have an attachment of *Mylyn* interaction traces. The interactions seem to be replaced with code reviews in these projects. The transition seems to perhaps indicate the better value from code reviews than interactions.

### 6.4.3 Benchmarks: Bugs, Commits, and Reviews

We collected all the *Bug ID*s for our benchmark from *Bugzilla*[12] for *Mylyn*, *Eclipse Platform*, and *LaunchPad*[13] for *OpenStack Nova*. The benchmark consists of a set of change requests. Each change request has a natural language query (short, textual summary) and a gold set of developers that addressed it. The benchmark was established by a manual inspection of the change requests, source code, their code review history in the Code review, *Gerrit*, and their historical changes

[12]https://bugs.eclipse.org/bugs
[13]https://bugs.launchpad.net/nova

recorded in version-control repositories, *Git*. In order to create our benchmark, we considered the following bug selection criteria: 1) were fixed and resolved, 2) had both commits and code reviews available prior to their reporting dates, 3) contained changes to the source code, 4) fix commits traceable to the bug tracking system, and 5) traceable to the release of the software because we need the latest snapshot of a system on or before a particular bug was reported.

For tracing each *Bug ID* in the commit repository, keywords such as *Bug ID* in the commit messages/logs were used as starting points to examine if the commits were in fact associated with the change request in the issue tracking system that was indicated with these keywords. The author and commit messages, which can be readily obtained, were processed to identify the developers who contributed changes to the change requests (i.e., gold sets). It should be noted that committers are not necessarily the actual authors or developers who resolved the bug and contributed fixes. The actual authors are typically mentioned in the commit messages or in a special author field (which is different from the committer field, e.g., in git). We considered the actual authors or developers in our gold set.

A vast majority of change requests are handled by a single developer (90% in *Mylyn*, 89% in *Eclipse Platform* and 95% in *OpenStack Nova*). Table 6.4 shows the frequency distributions of developers resolving issues in the benchmarks for *Mylyn*, *Eclipse Platform* and *OpenStack Nova*. For example, 114 issues were resolved by a single developer in *Mylyn*, 11 were resolved by two developers in *Eclipse Platform* and none of the issues were resolved by three developers in *OpenStack Nova*. The Inter Quartile Range (IQR) column shows the second and third quartile values for number of bugs which are resolved by unique developers in the benchmarks. For example, *Mylyn* has 23 unique developers of which 50% (second quartile) resolved at most one bug and 75% (third quartile) resolved at most 6 bugs out of the total 126 bugs in benchmark. *OpenStack Nova* has the most number of unique developers and hence the second and third quartile values are less than other projects. Our technique operates at the change request level, so we also needed input queries to test. We considered all of the bugs that have at least one associated Id in the commit messages and traces files. We created the final training corpus from all of the source code files in the last revision

Table 6.4: The frequency distributions of developers and summary statistics (IQR, unique developers) resolving issues in the benchmarks for *Mylyn*, *Eclipse Platform* and *OpenStack Nova*.

| System | # of Issues Resolved by | | | Total Issues | IQR | | #Unique Developers |
|---|---|---|---|---|---|---|---|
| | 1 Developer | 2 Developers | 3 Developers | | $Q_2$ | $Q_3$ | |
| *Mylyn* | 114 | 10 | 2 | 126 | 1 | 6.50 | 23 |
| *Eclipse Platform* | 116 | 11 | 3 | 130 | 1 | 2.25 | 40 |
| *OpenStack Nova* | 114 | 6 | 0 | 120 | 1 | 2.00 | 64 |

Table 6.5: Descriptive statistics of the review and commit history of the three open source subject systems

| | Start Date-Code Review | End Date-Code Review | #Submitted Patches | #Merged Patches | #Abandoned Patches | #Review Comments | Start Date-Commit | End Date-Commit | #Commits |
|---|---|---|---|---|---|---|---|---|---|
| *Eclipse Platform* | 02/28/2012 | 12/22/2014 | 1902 | 1419 | 331 | 10975 | 02/28/2012 | 12/22/2014 | 3100 |
| *Mylyn* | 02/29/2012 | 12/26/2014 | 1617 | 1212 | 337 | 10392 | 02/29/2012 | 12/26/2014 | 1867 |
| *OpenStack Nova* | 09/22/2011 | 03/10/2016 | 24776 | 17467 | 6384 | 401181 | 09/22/2011 | 03/10/2016 | 25607 |

before the bug issues in our benchmark were fixed. We considered the commits and reviews prior to that revision for training the respective approaches.

Table 6.5 shows the descriptive statistics of review and commit history related to the three studied systems. The number of commits is larger than the number of reviews because in the first couple of months of starting the review practice not all the developers did it. Also, we could not retrieve a few reviews from *Gerrit*. The experiment was run for $m = 1$, $m = 2$, $m = 3$, and $m = 5$, where $m$ is the number of recommended developers. We considered the top *ten* relevant files from the machine learning step.

### 6.4.4 Metrics and Hypotheses

We evaluated the accuracy of each of the approaches for all of the bug issues in our test set using the Recall and Mean Reciprocal Rank (MRR) metrics used in previous work [9, 65, 107]. For each bug *b*, in a set of bugs $B$ of size $n$, in the benchmark of a system and a $m$ number of

recommended developers, recall@m is given below:

$$recall@m = \frac{\mid RD(b) \cap AD(b) \mid}{\mid AD(b) \mid} \tag{6.12}$$

where *RD(b)* and *AD(b)* are the recommended developer by the approach and the actual developer who resolved the issue for the bug *b* respectively. This metric is computed for recommendation lists of developers with different sizes (e.g., $m = 1$, $m = 2$, $m = 3$, and $m = 5$ developers).

Increasing the value of $m$ could lead to a higher recall value (and typically does); however, it may come at the cost of an increased effort in examining the potential noise (false positives). Over 90% of the cases in our benchmarks have only a single correct developer. In such a scenario, each increment to $m$ in pursuit of the correct developer could add drastically to the proportion of false positives. Therefore, a traditional metric of the likes of precision is not suitable for our context. For example, if a correct developer is found at $m = 5$, the possible precision value is in the range [0.2, 1.0] for the rank positions [5, 1], typically around the lower bound of 0.2. Note that the precision metric is also agnostic to the rank positions of recommendations. Therefore, for a cutoff value of $m$, it would produce the same value for two approaches presenting the same number of correct answers. For $m = 5$, two approaches presenting a single correct answer at the positions 1 and 5 respectively will have the same precision value of 0.2. Therefore, we focused on evaluating the ranked position of the correct developer for each bug in each benchmark from a cumulative perspective regardless of the cutoff point $m$.

Mean Reciprocal Rank (MRR) is one such measure that can be used for evaluating any process that produces a list of possible responses to a sample of queries, ordered by probability of correctness. This metric has been used in previous work [100]. The reciprocal rank of a query response is the multiplicative inverse of the rank of the first correct answer. Intuitively, the lower the value (between 0 and 1), the farther down the list, examining incorrect responses, one would have to search to find a correct response.

$$MRR = \frac{1}{\mid n \mid} \sum_{i=1}^{|n|} \frac{1}{rank_i} \tag{6.13}$$

The reciprocal rank for a query (bug) is the reciprocal of the position of the correct developer in the returned ranked list of developers ($rank_i$) and $n$ is the total number of bugs in our benchmark. When the correct developer for a bug is not recommended at all, we consider its inverse rank to be a zero. When there are multiple correct developers (which are a very few cases in our benchmark), we consider the highest/first ranked position. The higher the value of MRR, the better it speaks of the potential effort spent in noise. For example, an MRR value of 0.5 suggests that the approach typically produces the correct answer at the 2nd rank.

We defined the following null hypotheses for our study (the alternative hypotheses can be easily derived from the respective null hypotheses):

- **H-1:** There is no statistically significant difference among the recall@m values of *rDevX*, *xFinder*, and *DevCom*.
- **H-2:** There is no statistically significant difference among the MRR values of *rDevX*, *xFinder*, and *DevCom*.

### 6.4.5 Results

The recall@1, recall@2, recall@3, and recall@5 values using *rDevX*, *xFinder*, and *DevCom* for *Eclipse Platform*, *Mylyn*, and *OpenStack Nova* were calculated from the established benchmarks. Table 6.6 shows the $recall@m$ values of all three systems using all three approaches. (see the Recall@m column). Similarly the MRR values for each of the compared approaches for *Eclipse Platform*, *Mylyn*, and *OpenStack Nova* were then calculated. Table 6.7 shows the $MRR$ values of all three system using compared approaches (see the MRR column).

To investigate the research question *RQ*, we computed the metric gain of *rDevX* (i.e., X equals to recall, or MRR) over another compared approach (i.e., Y equals to *xFinder*, or *DevCom*) using the following formula:

$$GainX@m_{rDevX-Y} = \frac{X@m_{rDevX} - X@m_Y}{X@m_Y} \times 100 \tag{6.14}$$

We computed the recall gain of *rDevX* over *xFinder* and *DevCom* using Equation 6.14. Similarly, we computed the MRR gain of *rDevX* over *xFinder* and *DevCom* using Equation 6.14. The **Gain %** columns in Table 6.6 show the recall gains of *rDevX* over each compared approach for the different $m$ values. The **Gain %** columns in Table 6.7 show the MRR gain of *rDevX* over each compared approach. The **p-valuech6** columns in Table 6.6 shows the p-values from applying the One Way ANOVA test on the recall values for $m = 1$, $m = 2$, $m = 3$, and $m = 5$. The **p-value** columns in Table 6.8 shows the p-values from applying the One Way ANOVA test on the reciprocal rank values.

Clearly, *rDevX* outperforms *xFinder* across recall, and MRR values in all the three systems. All the recall gains for different values of $m$ (with the exception of *Mylyn* and *OpenStack Nova* recall at $m$=1), and MRR gains are statistically significant (i.e., p-values$<$0.05). Nonetheless, *rDevX* is no worse than *xFinder* in these exceptional cases. We manually investigated the results for *OpenStack Nova* and noticed that both *rDevX* and *xFinder* recommend four core developers as top 1 for 50% of bugs in the benchmark. These core developers have contributed in majority of source code files in *OpenStack Nova* and that is the reason both approaches recommend them as top 1 and caused the similar results with no significant difference.

In case of the comparison between *rDevX* and *DevCom*, all the gains (with the exception of *Mylyn* recall at $m$=3, *OpenStack Nova* recall at $m$=1 and $m$=2 and *OpenStack Nova* MRR ) are positive or zero. The statistical testing showed that the gains are not significant (p-values $>$0.05). This analysis clearly suggests that combining both the commit history data and code review data does not have a significant impact on the developer expertise model. Infact, *rDevX* approach is better than *DevCom* as we get better results with relatively lesser amount of data i.e with code review data alone.

In case of the comparison between *DevCom* and *xFinder*, *DevCom* outperforms *xFinder* across recall, and MRR values in all the three systems. All the recall gains for different values of $m$ (with the exception of *Mylyn* at $m$=1 and $m$=2, *Eclipse Platform* at $m$=3 and *OpenStack Nova*

Table 6.6: Average, gains and p-values of recall @1, 2, 3 and 5 values of the approaches.

| System | | Recall@m | | | Gain*rDevX-* | | Gain*DevCom-* | p-value*rDevX-* | | p-value*DevCom-* |
|---|---|---|---|---|---|---|---|---|---|---|
| | $m$ | *rDevX* | *xFinder* | *DevCom* | *xFinder%* | *DevCom%* | *xFinder%* | *xFinder* | *DevCom* | *xFinder* |
| ***Mylyn*** | 1 | 0.31 | 0.27 | 0.31 | 14.90 | 0.00 | 14.90 | $< 0.50$ | $< 0.10$ | $\leq 0.40$ |
| | 2 | 0.42 | 0.30 | 0.40 | 40.00 | 5.00 | 33.33 | $\leq 0.04$ | $< 0.80$ | $\leq 0.10$ |
| | 3 | 0.57 | 0.38 | 0.59 | 50.00 | -3.34 | 55.26 | $\equiv 0.00$ | $< 0.90$ | $\equiv 0.00$ |
| | 5 | 0.70 | 0.40 | 0.70 | 75.00 | 0.00 | 75.00 | $\equiv 0.00$ | $< 0.10$ | $\equiv 0.00$ |
| ***Eclipse Platform*** | 1 | 0.31 | 0.20 | 0.29 | 55.00 | 6.90 | 45.00 | $\leq 0.03$ | $\leq 0.80$ | $\leq 0.04$ |
| | 2 | 0.45 | 0.30 | 0.43 | 50.00 | 4.65 | 43.33 | $\equiv 0.00$ | $\leq 0.80$ | $\leq 0.02$ |
| | 3 | 0.52 | 0.40 | 0.50 | 30.00 | 4.00 | 25.00 | $\leq 0.04$ | $\leq 0.80$ | $\leq 0.08$ |
| | 5 | 0.60 | 0.49 | 0.59 | 22.44 | 1.70 | 20.40 | $\leq 0.05$ | $\leq 0.90$ | $\leq 0.05$ |
| ***OpenStack Nova*** | 1 | 0.20 | 0.14 | 0.21 | 42.85 | -4.76 | 50.00 | $< 0.20$ | $< 0.90$ | $\leq 0.10$ |
| | 2 | 0.31 | 0.19 | 0.34 | 63.15 | -8.82 | 78.95 | $\leq 0.02$ | $< 0.60$ | $\equiv 0.00$ |
| | 3 | 0.43 | 0.29 | 0.43 | 48.27 | 0.00 | 48.27 | $\leq 0.01$ | $< 0.10$ | $\leq 0.01$ |
| | 5 | 0.50 | 0.37 | 0.49 | 35.13 | 2.04 | 32.43 | $\leq 0.03$ | $< 0.90$ | $\leq 0.04$ |

at $m$=1), and MRR gains are statistically significant (i.e., p-values<0.05). Nonetheless, *DevCom* is no worse than *xFinder* in these exceptional cases.

In summary, the overall results suggest that *rDevX* generally performs better than *xFinder* in terms of both recall and MRR. Using the code review history typically leads to improvements in accuracy. Not only does it identify the correct developers more often (as evident by the significant recall gains or no loss), but also at a high enough position in the list of recommended candidates (as evident by the significant MRR gains or no loss). For example, *rDevX* recorded recall gains over *xFinder* in the range [14.90% , 75%] for *Eclipse Platform*, *Mylyn*, and *OpenStack Nova* respectively. Also, on average, the correct developer would appear at the 2nd or 3rd positions (MRR=[0.33 , 0.48]) for *rDevX* recommendations, whereas, in *xFinder* this value would be at the 3rd and 4th positions (MRR=[0.24 , 0.34]) which caused the MRR gain of 41%. Based on the results, we find support to reject Hypothesis **H-1** and **H-2** in favor of *rDevX*.

***RQ:*** *rDevX performs much better than xFinder which relies on source code repository data, and rDevX is statistically equivalent to DevCom which requires both past reviews and commits.*

Table 6.7: Mean Reciprocal Rank of the approaches *rDevX*, *xFinder*, and *DevCom* measured on the benchmarks.

| System | MRR | | | Gain*rDevX*- | | Gain*DevCom*- |
|---|---|---|---|---|---|---|
| | *rDevX* | *xFinder* | *DevCom* | *xFinder%* | *DevCom%* | *xFinder%* |
| ***Mylyn*** | 0.48 | 0.34 | 0.47 | 41.17 | 2.12 | 38.24 |
| **Eclipse** | 0.46 | 0.34 | 0.45 | 35.29 | 2.22 | 32.35 |
| ***OpenStack*** | 0.33 | 0.24 | 0.34 | 37.50 | -2.94 | 41.67 |

Table 6.8: p-values from applying one way ANOVA on MRR values for each subject system.

| System | MRR p-value | | |
|---|---|---|---|
| | *rDevX*- | | *DevCom*- |
| | *xFinder* | *DevCom* | *xFinder* |
| ***Mylyn*** | $< 0.01$ | $< 0.9$ | $\leq 0.01$ |
| ***Eclipse*** | $< 0.02$ | $< 0.9$ | $\leq 0.02$ |
| ***OpenStack*** | $\leq 0.04$ | $< 0.9$ | $\leq 0.02$ |

### 6.4.6 Qualitative Analysis

We discuss a few points that would help understand the rationale behind the improved performance from *rDevX*:

*Different Contributions*: There are indeed different levels of contributions measured from commit and review repositories. For example, *David Green* submitted eight patches to the package *"org.eclipse.mylyn.wikitext.markdown.core"*, which were captured in the review archives. Only seven were accepted, which were recorded in the commit archives. The one abandoned patch was not due to poor quality, but it was coauthored with another developer and moved to a different review record. Thus, *David Green* is more knowledgable in this package than what the commit history would suggest.

*Multiple Roles*: We did find evidence that the same individuals contribute in multiple roles. Again, *David Green* reviewed 13 patches authored by other developers in the package *"org.eclipse.mylyn.wikitext.markdown.core"* but not necessarily the code he authored. Manual ex-

amination of his textual comments suggests that they were very detailed and demonstrated subject knowledge. Again, commit archives completely miss this dimension of knowledge acquisition, but code review archives do capture it.

*Makeup for Inexact Relevant Files*: *David Green* resolved the issue #440935 "*create a document builder for Markdown*" for *Mylyn*, which appears to be a feature request. The KNN method to feature location did not list any of the source code files he changed, i.e., the accepted patch, to resolve this issue. Instead, KNN listed top files (e.g., *MarkdownReferenceValidationRule.java*) which were again in the same package, *"org.eclipse.mylyn.wikitext.markdown.core"*. *David Green* did not have a single commit related to these files; however, he did contribute reviews to them. Thus, he had a previous record of code review, which was used in *rDevX*.

*Dissimilar Recommendations*: We calculated the Jaccard similarity between the recommended lists of developers from *xFinder* and *rDevX* for our benchmarks. The average Jaccard values for *Eclipse Platform*, *Mylyn*, and *OpenStack Nova* were 0.57, 0.40, and 0.70 respectively. These values suggest their recommendations are quite dissimilar.

### 6.5 Threats to Validity

*Accuracy measures and correctness of developer recommendations*: We used the widely used metric recall in our study. We also calculated mean reciprocal rank. We considered a gold-set to be developers who contributed source code changes to address change requests. Of course, it is possible that other team members are also equally qualified to handle these change requests; however, such a gold-set is very difficult to obtain. Nonetheless, our benchmark can be viewed as conservative bounds.

*Machine Learning-based matching of change requests to relevant files*: KNN sometimes returned classes (i.e., files) that were not found in the commits related to the bug fixes or change request implementations. However, based on our prior work [112], we observed that the files that were recommended as textually similar were either relevant (but not involved in the change that resolved the issue) or conceptually related (i.e., developers were also knowledgeable in these parts).

*Normalizing Developer Expertise*: Our approach calculates eight markers for each develo-

per's contribution to specific source code entity (file $f$). Each of these markers was normalized to the corresponding marker of the total activity to that entity (file $f$), i.e., the *SoE* score. We could have used other distance measures such as the cosine similarity between their vector representations. This issue may have introduced a construct validity to the results.

*Individual Effect of Expertise Markers*: Although, we empirically compared commit and review set of markers, we did not study the statistical effect of individual markers in each category. For example, we did not empirically compare the effects of the number of patches submitted vs accepted, which were markers derived from code reviews.

*Narrow Ground Truth:* Our ground truth included only the developers who eventually resolved the change request. It is likely that other developers are equally capable of resolving the same requests. Identifying them is not obvious from repositories and would require a quantitative study.

*Reviewer Identity Mismatch*: Although we carefully examined the available sources of information to match the different identities of the same reviewer, it is possible that we missed or mismatched a few cases. There were several cases of different IDs of the individuals in *git* and *Gerrit* repositories, which we manually mapped with string/pattern matching.

*Availability and Workload of Developers*: Our approach only considers the expertise of developers and not other factors such as their availability, workload, and/or willingness. If the highest ranked developers cannot be assigned due to such non-technical factors, others down the list should be considered. Therefore, our approach recommends a user-defined number of developers.

*Different Review Processes and Tools*: We only considered *Gerrit*'s review process and model. We did not consider other methods and tools of reviews such as emails, and pull requests in *git* or *GitHub*.

*Generality*: Although we investigated three open source systems, which actively use the code review process, we do not claim that our results would generalize to every software system in these domains.

# CHAPTER 7

## Code Review Recommendation (*cHRev*)

Code review is an important part of the software development process. Recently, many open source projects have begun practicing code review through ”modern” tools such as GitHub pull-requests and Gerrit. Many commercial software companies use similar tools for code review internally. These tools enable the owner of a source code change to request individuals to participate in the review, i.e., reviewers. However, this task comes with a challenge. Prior work has shown that the benefits of code review are dependent upon the expertise of the reviewers involved. Thus, a common problem faced by authors of source code changes is that of identifying the best reviewers for their source code change. To address this problem, we present an approach, namely *cHRev*, to automatically recommend reviewers who are best suited to participate in a given review, based on their historical contributions as demonstrated in their prior reviews [110]. We evaluate the effectiveness of *cHRev* on three open source systems as well as a commercial codebase at Microsoft and compare it to the state of the art in reviewer recommendation. We show that by leveraging the specific information in previously completed reviews (i.e., quantification of review comments and their recency), we are able to improve dramatically on the performance of prior approaches, which (limitedly) operate on generic review information (i.e., reviewers of similar source code file and path names) or source code repository data. We also present the insights into why our approach *cHRev* outperforms the existing approaches.

### 7.1 Introduction

software peer review, which is a manual inspection of source code by other stakeholders besides its author, has been in practice for several years [5, 6]. Recently, a number of empirical studies about various facets of the modern code review process have been reported in the literature [69, 82, 17, 103, 49, 18, 81, 83, 84]. Deeply inspired by these efforts, we focus our work on the critical topic of finding the human reviewers who are most likely to contribute in peer reviewing

source code changes. Bacchelli et al. [13] studied modern code review at Microsoft and found that if reviewers have a prior knowledge of the context and code under review, they complete the reviews more quickly and provide more valuable feedback to the author. Thus, expertise and knowledge have a direct effect on code review quality. Rigby et al. [84] studied the broadcast based peer review on OSS. They discovered that sometimes an author of a patch, based on their confidence that they have a good working knowledge of the code involved in the patch, prefers to use an explicit review request or send an email message directly to potential reviewers.

These studies demonstrate that a developer's expertise on a certain part of the source code is an important factor for considering them as a potential reviewer. However, it is not always easy to determine who has the most expertise given a particular change for review, especially for newcomers to a codebase or those changing parts of the code with shared ownership by many people. Thus, authors of a change and/or reviewer assigners are often confronted with the question *"Who should review this change?"* In the area of code review, requests for help selecting the right reviewers are one of the most common asks from developers at Microsoft (requests for a system providing help occur weekly on review mailing lists). One developer recently shared his frustration:

*"I made a one line change to Exchange in a part of the code that I don't typically work on and so of course I had to have it reviewed. I added the dev who most recently changed the file and he reviewed it, but then told me to be sure to add the owner and told me who it was. So I added him and he told me to make sure and have his lead review it as well. In the end, it took two weeks to get my one line change in!."*

Finding the right reviewers does not often take two weeks; however, this experience is emblematic of the need to find appropriate reviewers in a timely manner for a code change. Clearly, there is strong anecdotal and empirical evidence from both OSS and commercial domains on the importance of finding the most appropriate reviewers to sustain the code review process effectively and efficiently [13, 82]. The value of choosing the right reviewers to examine code is not new. Selecting and assigning reviewers to a review process was one of the managers' responsibilities in traditional inspection, which was done manually [38]. Unfortunately, there has been little effort in

building automatic approaches to recommend the most suitable reviewers in modern (tool-based) code review process, which includes the work of Xia et al. [98], Thongtanunam et al. [96], and Balachandran [14]. Balachandran termed the task of identifying the most appropriate reviewers for a change as *Reviewer Recommendation*.

This chapter presents an approach, namely *cHRev*, to solve this problem automatically based on historical code review information. In a nutshell, it favors **c**ode review **H**istories over other types of past information to recommend **Rev**iewers; hence, the name *cHRev*. *cHRev* rests on two key insights. The first is that reviewers are not necessarily confined to developers who may have committed changes to source code previously that is the subject of review again for another change, e.g., for a bug fix or a feature implementation. For example, there may be team members who own other related features and/or source code modules or who do not work on code directly that have the expertise to provide quality code review feedback. The second is that expertise changes over time and thus both the *frequency* and *recency* must be accounted for to find the most appropriate reviewers.

In an effort to demonstrate the effectiveness of our approach, we compare *cHRev* with *REVFINDER* [96], *xFinder* [52], and *RevCom*. We show that *cHRev* outperforms all these three approaches. *REVFINDER* is a recently proposed technique that uses code review history to identify reviewers. *REVFINDER* assigns an expertise score to reviewers based on their number of past reviews on similar file names and paths. Unlike *cHRev*, it does not consider the amount of contributions (feedback comments and days) in each past review and their temporal recency. *xFinder* is a developer recommendation approach for source code, which is used here for reviewer recommendation. To assess the potential orthogonally between the code commits and reviews, we devised a combined approach, namely *RevCom*, which is based on the factors of *cHRev* and *xFinder*.

Our work makes the following noteworthy contributions in recommending relevant reviewers for a given change:

1. We present *cHRev* that utilizes code review histories for recommending reviewers for a code change.

2. We perform a comparative study of *cHRev*, *REVFINDER*, *xFinder*, and *RevCom*.

3. We demonstrate the effectiveness of *cHRev* through an empirical evaluation on one industrial (*MS Office*) and three open source (*Android Platform*, *Eclipse Platform*, and *Mylyn*) systems.

The rest of this chapter is organized as follows: Section 7.2 presents background of modern code review and associated terminology. Our approach is discussed in Section 7.3. The empirical study on *Android Platform*, *Eclipse Platform*, *Mylyn*, and *MS Office*, and its results are presented in Section 7.4. Threats to validity are encountered in Section 7.5. Related work is discussed in Section 7.6.

## 7.2 Background on Modern Code Review

In this section, we define the key concepts involved in the modern code review, which is driven by supporting infrastructure and tools, e.g., *Gerrit* and *CodeFlow*.

*Code Change*: A code change is a set of modified source code files submitted to fix a bug or add a new feature.

*Review*: A code review is a record of the interactions between the owner of a change and reviewers of the change including comments on the code and signoffs from reviewers.

*Owner*: An owner is the developer who makes the change in the source code and submits it for review.

*Reviewer*: A reviewer on a particular review is a developer who is assigned to and/or contributes to that review.

*Review Comment*: A review comment is textual feedback written by a reviewer about the code change during the review process. A review comment may be about the change in general or may be explicitly tied to a particular part of the change.

The lifecycle of a review is as follows: Initially a developer (the owner) makes changes to the source code in response to a bug report or feature request. Once complete, they submit the code change for review. The owner may indicate the intended reviewers, who are subsequently notified about the review invitation. It should be noted that the invited reviewers do not necessarily accept

the invitation and contribute to the review. Reviewers then inspect the change through the code review tool (a web page in the case of *Gerrit* or a windows application in the case of *CodeFlow*) and provide feedback in the form of review comments to the owner. The code change is typically depicted by showing the difference of the code before and after the change. The owner may update the change and submit the update to the review as a result of such feedback. Eventually, a reviewer "signs-off" on the review, once they believe the code change is of sufficient quality to be checked into the code repository. If a change never received sign-offs, it is abandoned. The number of sign-offs required to check in a code change is typically dependent on the team policy. *Gerrit* is a modern peer-review tool that facilitates a traceable review process for *git*-based software projects [69]. Developers make local changes in their private *git* repositories and then submit these changes as a patch for review [82]. Most Microsoft developers practice code review using *CodeFlow*, an internal tool for reviewing code, which is under active development and regularly used by more than 50,000 developers. *CodeFlow* is a collaborative code review tool similar to other popular review tools such as *Gerrit*.

Code review is a quality assurance mechanism and is required for checkin. Therefore, it is critical that it is both effective (actually improves code changes and blocks poor code from being checked into the repository) and timely (does not act as a bottle-neck to by slowing down changes). Prior research [13] has found that higher expertise of reviewers leads to both.

### 7.3 The *cHRev* Approach

The basic premise of our approach is that the reviewers who reviewed the units of source code in the past are most likely to best assist with reviewing it in the future. Our approach, *cHRev*, takes a code change submitted for review and mines the archives of reviews, i.e., review history, from the code review system (e.g., *Gerrit*) to recommend a ranked list of candidates for reviewing the given code change. It utilizes the past code changes and their reviewers to form a quantifiable model of the expertise of each reviewer in each source code file. In a code change, the cardinality of source code files is typically greater than 1. Therefore, the overall expertise of each candidate reviewer for the given code change is derived from a cumulative scoring function for all source

code files in it. Finally, a ranked list of top $n$ (a tunable user parameter) reviewers is recommended. To be specific, *cHRev* consists of the following steps:

**Step 1: Extract Source Code Under Review:** Given a code change under review for which reviewers are desired, it extracts each source code file.

**Step 2: Formulate Reviewer Expertise:** For each source code file in Step 1, it forms a reviewer expertise model based on how many, who performed, and when reviews were performed on it in the past. That is, we need to know the contribution of each past reviewer over the total number of reviews on it from the code-review history.

**Step 3: Score and Recommend Reviewers:** Finally, the cumulative contributions of the reviewer in Step 2 for all the source code files in Step 1 are scored to arrive at a ranked list of candidate reviewers. A user defined parameter $m$ is used to recommend the top $m$ candidates from this list. The choice of $m$ can be guided by the organizational or project practices or historical information on the typical number of reviewers.

### 7.3.1 Formulating Reviewer Expertise Model

The review comments are a mechanism that reviewers use to express their feedback and communicate with the owner and other peer reviewers of a code change. That is, these comments are a primary means for discussion and discourse in modern peer code review. They can be considered a manifestation of their expertise. Now, the question is how these valuable source can be used to quantify the expertise of reviewers. We use three metrics to quantify reviewers' expertise from their contributed review comments.

One measure of a reviewer's contribution is the total number of review comments they contributed to previous code changes. A particular reviewer who contributed a larger number of review comments than another peer to specific units of source code (i.e., files) can be considered more knowledgeable on those parts. Although, this count measure may capture valuable expertise information, it may not be the only reflection of expertise. Depending on the complexity and nature of each code change, different levels of effort may be needed. We consider time as a proxy measure of effort, which is typically used in other domains [87]. We consider the smallest unit of

work, i.e., effort, devoted by a reviewer to be a workday. A reviewer's workday is considered as a day (calendar date) on which they contributed at least one review comment (to at least one file) in a code change, because a reviewer can have multiple review comments on a given workday. A day on which no such review comments exist is not considered a workday. Two reviewers are considered to have made the same overall effort in reviewing changes to the same source code file if they wrote all their review comments (regardless of the variation in their counts) in the same number of calendar (work) days. Accounting for the frequency (review count) and effort (workday) may not suffice, if they are not relevant to the submitted code change under review. The third measure accounts for the recency of the review comments. Recent review comments are given a higher weight than the distant ones, i.e., it is an inverse measure. Each of these three measures is normalized with respect to the total contributions on each source code file.

Previously, these three measures were used and validated in the context of commit history and developer recommendation, i.e., finding the developers who are most likely experts in particular source code units and/or fixing a bug [52]. Therefore, using this foundation, we contextualized and redefined them for the reviewer recommendation task, i.e., to determine the reviewers who are more likely to be experts in reviewing a specific source code file than others, i.e., *reviewer-expertise* map. The *reviewer-expertise* map, *RE*, for the reviewer *r* and file $f$ is given by

$RE_{(r,f)} = \langle C_f, W_f, T_f \rangle$ , where $C_f$ is the number of review comments contributed by the reviewer *r* for the file *f*. $W_f$ is the number of workdays of the reviewer *r* on which they contributed review comments for the file *f*. $T_f$ is the most recent workday of the reviewer *r* with the file *f*. Similarly, the *file–review* map, *FR*, represents the review contribution to the file $f$ and is given by

$FR_{(f)} = \langle C'_f, W'_f, T'_f \rangle$ , where $C'_f$ is the number of review comments that are written for the file *f*. $W'_f$ is the total number of workdays on which review comments were contributed for the file *f*. $T'_f$ is the most recent workday on which a review comment was contributed for the file *f*.

The contribution or expertise factor, termed *xFactor*, for the reviewer *r* and the file $f$ is computed using the ratios of the *reviewer-expertise* and *file–review* maps. The contribution factor, *xFactor*, is given below:

$$xFactor(r, f) = \frac{RE_{(r,f)}}{FR_{(f)}} \tag{7.1}$$

$$xFactor(r, f) = \begin{cases} \frac{C_f}{C'_f} + \frac{W_f}{W'_f} + \frac{1}{|T_f - T'_f|} & \text{if } |T_f - T'_f| \neq 0 \\ \frac{C_f}{C'_f} + \frac{W_f}{W'_f} + 1 & \text{if } |T_f - T'_f| = 0 \end{cases} \tag{7.2}$$

The *xFactor* score is computed for each of the source-code files that exist in the code change. According to Equation 7.2, the maximum value of *xFactor* can be three because we have used three measures, each of which can have the maximum contribution ratio of 1.

### 7.3.2 Scoring and Recommending reviewers

We now describe how the ranked-list of reviewers is obtained from all of the scored reviewers of each source code file in the code change. There is a one-to-many relationship between the source code files and reviewers. That is, each file $f_i$ may have multiple reviewers; however, it is not necessary for all of the files to have the same number of reviewers. For example, the file $f_1$ could have two reviewers and the file $f_2$ could have three reviewers. The matrix $D_r$ (see Equation 7.3) gives the list of unique reviewers for each file $f_i$. $D_{rf_i}$ represents the set of reviewers, with no duplication, for the file $f_i$, where $1 \leq i \leq n$ and $n$ is the number of unique files in patch. $r_{ij}$ is the $j^{th}$ reviewer in the file $f_i$ with $l$ unique reviewers.

$$D_r = \begin{pmatrix} D_{rf_1} \\ D_{rf_2} \\ \vdots \\ D_{rf_n} \end{pmatrix} D_{rf_i} = \{ r_{i1}\ r_{i2}\ \cdots\ r_{il} \} \tag{7.3}$$

Although, a single file does not have any duplicate reviewers, two files may have common reviewers. In Equation 7.4, $D_{ru}$ is the union of all unique reviewers from all files.

$$D_{ru} = \bigcup_{i=1}^{n} D_{rf_i} \tag{7.4}$$

$$Score(r) = \sum_{i=1}^{n} xFactor_i(r, f_i) \tag{7.5}$$

Each reviewer $r$ for a file $f$ has an $xFactor$ score. To obtain the likelihood of the reviewer $r$, i.e., $Score(r)$, to review the code change, we sum $xFactor$ scores of the unique files in which it

appears (see Equation 7.5). The $Score(r)$ value is calculated for each unique reviewer $r$ in the set $D_{ru}$.

In Equation 7.6, we have a set of candidate reviewers. If the owner of the review occurs in this list, we remove them, as recommending the owner as a reviewer makes little sense because we want to recommend other peers. The reviewers in this set are ranked based on their $Score(r)$ values. Once the reviewers are ranked in descending order of their $Score(r)$ values, we have a ranked list of candidate reviewers. By using a cutoff value of $m$, we recommend the top $m$ candidate reviewers, i.e., with top $m$ $Score(r)$ values, from the ranked list obtained from the set $RF$.

$$RF = \{(r, Score(r)), \forall r \in D_{ru}\} \tag{7.6}$$

This step concludes *cHRev* and we have the top $m$ candidate reviewers recommended for the given code change.

We considered a cumulative view of all the files in a patch to recommend reviewers. Therefore, we lower the probability of an empty recommendation because a code change typically has multiple files; however, some files may have not been reviewed in a very long time or added for the very first time to the review process. As a result, there will not be any recommendations at the file level. To overcome this problem, we look for reviewers with review contributions to a package that contains the file, and recommend them instead. If no package-level reviewers can be identified, we turn to the system-level reviewers as the final option. Package here means the immediate directory that contains the file, i.e., we consider the physical organization of source code. The system means a collection of packages. It can be a subsystem or a module (i.e., the top level directory). In this way, we move from the lowest, most specific expertise level (file) to the higher, broader levels of expertise (package then system). According to this approach, we guarantee that our tool always gives a recommendation, unless this is the first ever file added to the system.

Table 7.1: The reviewers extracted with *cHRev* from each of the files related to code change in the review # 33689.

| Files | Reviewers and their *xFactor* |
|---|---|
| …/TaskListFilteredTree.java | Sam Davis: 3.00 |
| …/CustomTaskListDecorationDrawer.java | Frank Becker: 1.83, Sam Davis: 1.62,Tomasz Zarna: 1.54 |

### 7.3.3 Implementation of *cHRev*

To extract the code review data from *Android Platform*, *Eclipse Platform*, and *Mylyn*, we used the *Gerrit* JSON API and queried their *Gerrit* servers[1]. We also engineered a *review log*, which is akin to a *version log* from source code repositories. Unfortunately, a review log is not readily available like the version log. It was assembled from the code review history available in *Gerrit*. Review log entries include the dimensions: reviewer, date and path (e.g., files), involved in review process. After assembling the review log from the available code review history in *Gerrit*, the review log entries are readily available in the form of *XML* and straightforward *XPath* queries are formulated to compute the measures. The measures $C_f$, $W_f$, and $T_f$ are computed from the review log. More specifically, the dimensions reviewer's name, date, and paths of the log entries are used in the computation. The dimension date is used to derive workdays or calendar days. The dimension reviewer's name is used to derive the reviewer information. The dimension path is used to derive the file information. The measures $C'_f$, $W'_f$, and $T'_f$ are similarly computed. The expertise model and scoring functions are implemented in Java.

### 7.3.4 A Motivating Example from *Mylyn*

Here, we demonstrate our approach *cHRev* using an example from *Mylyn*. The goal is to show the inner workings of the *cHRev* mechanisms and compare its results with three other approaches. The first approach *REVFINDER* is based on reviews. The second approach *xFinder* is based on commits. The third approach *RevCom* is based a combination of commits and reviews (see Section 7.4.2 for details). The review of interest here is the review #33689: *"clean up workspace*

[1]https://gerrit-review.googlesource.com/Documentation/rest-api.html

*warnings in tasks.ui"*. *Steffen Pingel* is the owner and the code change includes two files. *Sam Davis* and *Tomasz Zarna* are the actual reviewers of review #33689 (highlighted in red color with their names postfixed and an asterisk in Table 7.2).

In *cHRev*, we first obtained the set $D_{ru}$ from all of the reviewers recommended for each file $f_i$ that exist in the review #33689 (see Table 7.1). The set $D_{ru}$ consists of 3 unique reviewers. A review log is created and all the reviews before this example review have been considered in calculating the expertise metrics and forming the model. Table 7.1 shows the two related files to the review #33689, for each file $f_i$ there is a set of recommended reviewers with their associated *xFactor* values calculated by *cHRev*.

For each of the 3 unique reviewers, the $Score$ value is calculated according to Equation 7.5. Table 7.2 shows the top four $Score$ values and the corresponding reviewers, i.e., $m = 4$ for four approaches: *cHRev*, *REVFINDER*, *xFinder*, and *RevCom*. $Score$ values for *REVFINDER* have been calculated in different way (see section 7.4.2). *Sam Davis* has the highest score in the set $RF$ (a value of 4.62 in the first column) for *cHRev*, so he is the first recommended reviewer. For the remaining reviewers, the value of the function $Score$ is less than *Sam Davis*'s score, so they all have a rank greater than 1. *REVFINDER* recommended one of the reviewers at rank 4 and two of the recommended reviewers by *REVFINDER* do not exist in the recommendation list by *cHRev*. *xFinder* did not recommend any of correct reviewers in the golden set. One of the recommended reviewers by *xFinder* does not exist in the recommendation list by *cHRev*. *RevCom* recommended the reviewers but with different (worse) ranking. Clearly, the best result belongs to *cHRev* because it recommended *Sam Davis* and *Tomasz Zarna* with ranks 1 and 3. Based on the degree of file name and path similarity which is determined by string comparison techniques for *REVFINDER*, *Sebastien Dubois* has the highest score related to the string similarity score. After investigating the review history and commit history, we ascertained that *Sam Davis* and *Tomasz Zarna* did not have any commits on those two files before the creation date of review #33689. Hence, *xFinder* could not recommend them as candidate reviewers. The most contribution for those two files according to the commit history belongs to *Steffen Pingel* (owner) and *Frank Becker*. One can ask the question

if the most contribution belongs to *Frank Becker* then probably he is the best candidate to review the code based on the findings of previous work [52]. Even considering this point, Table 7.2 shows that *cHRev* recommends *Frank Becker* at rank 2 and *REVFINDER* recommends him at rank 3. *Sam Davis* and *Tomasz Zarna* had acted as reviewers in *Mylyn*, hence *cHRev* picked them.

Table 7.2: Top four reviewers recommended to review the review #33689 with their associated ranks and score by *cHRev*, *REVFINDER*, *xFinder*, and *RevCom*.

| | ***cHRev*** | | ***REVFINDER*** | | ***xFinder*** | | ***RevCom*** | |
|---|---|---|---|---|---|---|---|---|
| **Reviewer** | **Score** | **Rank** | **Score** | **Rank** | **Score** | **Rank** | **Score** | **Rank** |
| *Sam Davis** | 4.62 | 1 | 33.31 | 4 | - | - | 4.62 | 2 |
| *Frank Becker* | 1.83 | 2 | 37.29 | 3 | 3.0 | 1 | 4.83 | 1 |
| *Tomasz Zarna** | 1.54 | 3 | - | - | - | - | 1.54 | 3 |
| *Caitlin Matthew* | - | - | - | - | 0.50 | 2 | 0.50 | 4 |
| *Sebastien Dubois* | - | - | 63.80 | 1 | - | - | - | - |
| *Miles Parker* | - | - | 55.51 | 2 | - | - | - | - |

## 7.4 Case Study

The purpose of this study was to investigate how well our *cHRev* approach recommends correct reviewers to review a given code change and compare with available alternatives: a code review based *REVFINDER*, a commit based *xFinder* and a combined *RevCom* based on commits and reviews. Next, we present the details of the study design, its execution, and observed results.

### 7.4.1 Design

We conducted a case study to empirically assess our approach according to the design and reporting guidelines presented in [89]. The case of our study is the event of assigning reviewers to code changes in closed and open source systems. The units of analysis are the code changes considered from four systems. Therefore, this study would allow us to compare code reviews and commits with respect to the reviewer recommendation task. We addressed the following research questions:

**RQ1**: What is the accuracy of *cHRev* in recommending reviewers on real software systems across closed and open source projects?

**RQ2**: How do the accuracies of *cHRev* (trained from the code review history), *REVFINDER* (also, trained from the code review history, albeit differently), *xFinder* (trained from the commit history), and *RevCom* (trained from a combination of the code review and commit histories) compare in recommending code reviewers?

Guided by the Goal-Question-Metric (GQM) method, the main goal of the first part of our study is to assess the effectiveness of our approach, i.e., asking how accurate are the reviewers recommendations when applied to the change requests of real systems across domains? The main focus of the quantitative analysis is on addressing different viewpoints, i.e., theory triangulation, of recommendation accuracy. We collected a fixed datasets, i.e., code changes, from the software review archives found in modern peer review systems. We used a data triangulation approach to include a variety of factors from closed and open source subject systems. These systems represent different main implementation languages (e.g., C/C++ and Java), sizes, review systems, and development environments. We used four metrics (precision, recall, F-score, and MRR) to cover different perspectives of accuracy.

### 7.4.2 Compared Approaches: *REVFINDER*, *xFinder* and *RevCom*

*REVFINDER* is a recently reported code-review based review recommendation approach. Its model is based on finding reviewers of source files with similar names and paths to those submitted in a given code change. The degree of file name and path similarity is determined with string comparison techniques and the reviewers are scored with the string similarity score. *REVFINDER* was shown to perform better than Balachandran's *REVIEWBOT* [14]. We also compare with a previous approach, namely *xFinder*, for developer recommendation that uses past commits on source code. These recommendations are used for reviewer recommendation for the source code submitted for change review. *xFinder* builds the developer expertise based on the number of commits, and their number of workdays and recency. *xFinder* subsumes the default reviewer recommender

in *Gerrit*[2]. *xFinder* was shown to be competitive with other developer recommendation approaches [65]. To assess the potential orthogonally between the commits and review, we devised a combined approach, namely *RevCom*, which is based on the factors of *cHRev* and *xFinder*. *RevCom* considers three metrics from reviews and another three from commits. The presence of orthogonality between different sources have been leveraged in several other software engineering tasks previously [41], which served as an inspiration to emulate the combination for the reviewer recommendation task.

### 7.4.3 Subject Systems and Evaluation Datasets

Our evaluation datasets were derived from three open and one closed source systems.

**Open Source: *Android Platform*, *Eclipse Platform*, and *Mylyn***

*Android* contains 7 years of code review related to different sub projects. In this study we considered the code review history of *Android Platform*[3] sub project between February 7, 2015 and March 26, 2015. During the defined period, there were a total of 2,052 source code changes and 2680 code reviews that include at least one source file. We considered this period of history because it contains a similar number of code reviews used in the evaluation of *REVFINDER* on *Android*. Reviewers provided 23181 review comments. We considered the author of the commit (and not the committer) for *xFinder*.

*Eclipse* contains six different sub projects. In this study, we consider *Eclipse Platform*, because it has the largest code review history available in comparison with the other sub projects[4]. Its code review history in *Gerrit* is available from March 2013. We considered the history between March 5, 2013 (the first day of a code review history in *Gerrit*) and November 28, 2014. The *Eclipse Platform* project consists of 1854 code reviews uploaded in *Gerrit* repository, each of which includes at least one Java file. After removing the noise (e.g., automatically submitted comments by tools such as Hudson) a total of 10506 review comments are written in *Eclipse Platform*.

[2]https://gerrit-review.googlesource.com/#/admin/projects/plugins/reviewers-by-blame

[3]https://android-review.googlesource.com/#/q/platform

[4]https://git.eclipse.org/r/#/q/platform,n,z

The *Eclipse Platform* project consists of 3155 commits in the commit history (during the defined period), each commit contains a change to at least one Java file.

*Mylyn* contains about 2 years of code review data in *Gerrit* and is an Eclipse Foundation project. Its commit history in the *git* repository is available from June 2005. Its code review history in *Gerrit* is available from March 2012. We considered the history between March 2, 2012 (the first day of a code review history in *Gerrit*) and November 28, 2014. The *Mylyn* project consists of 1589 code reviews uploaded in *Gerrit*, each of which includes at least one Java file [5]. Similar to *Eclipse Platform*, noisy review comments were discarded. A total of 10157 review comments were written in *Mylyn*. *Mylyn* consists of 1838 commits in the commit history (during the defined period), each commit contains a change to at least one Java file.

**Closed Source: *MS Office***

We also evaluated all approaches on activity from one milestone development cycle on one of the main development branches of *MS Office*. We gathered source code repository data from *CODEMINE* [36] our development analytics database and code review data from *CodeFlow Analytics* [24], an internal data collection system for code reviews across Microsoft. It is common for automated systems to make source code changes (e.g., updating copyright dates in headers) in *MS Office*. In addition, some teams at Microsoft use automated "Review Bots" in reviews similar to VMWare [14]. We remove such code change authors and reviewers from the data as the rules for their inclusion are automatic and they do not represent humans that a reviewer could assign a review to. After cleansing the data, there were a total of 2,651 source code changes that include at least one source file (C# or C++) and 1,886 code reviews. Reviewers provided 10,746 review comments in 845 of the reviews.

Table 7.3 gives the test benchmarks for all the four systems considered in our study. It consists of code changes and reviewers who contributed to review those code changes. That is, a reviewer who provided at least one comment on the code change is considered a true positive. Note that in tools, such as *Gerrit*, the patch author can pick the potential reviewers; however, there is

[5] https://git.eclipse.org/r/#/q/mylyn,n,z

no guarantee that all (or any) of them would actually contributed. Thus, we do not consider such names as a gold set, and only consider the ones who actually contribute regardless of whether they were originally picked by the patch author or not. To investigate the difference between the lists of assigned reviewers by the owner and the list of participated reviewers, we calculated the Jaccard similarity between these two lists of reviewers for all the three open source projects used in our study. The average Jaccard similarity values for *Android Platform*, *Eclipse Platform*, and *Mylyn* are 0.58, 0.80, and 0.85 respectively. These values indicate that the two lists are not identical and are quite dissimilar.

The only code change information we use, is the files in the code change. The goal of the compared recommendation techniques is to predict reviewers for each of these code changes from the previous commits and/or reviews in the history periods considered for each subject system. Note that we only considered the original version of the code change. Including files from other subsequent revisions of the original version (e.g., to address the review feedback) would be forward looking information with a limited (or no) value in predicting reviewers. Therefore, our benchmark is a set of code changes, each code change includes several unique files. After a manual investigation of reviews in the open source systems in our study, we found that there are several code changes that included only test files. We also found that these code changes did not receive any review comments, whereby indicating that they did not need to be reviewed. We discarded them from our benchmarks. Furthermore, there were code changes that contained a mix of source code and test files. On manual examination, we found that code changes with a majority of source code files were reviewed. To provide a conservative bound, we included such mixed cases in our benchmark.

### 7.4.4 Evaluation Protocol for *cHRev*

The source code changes from *Gerrit* and *CodeFlow* are used for evaluation purposes. Our general evaluation procedure consists of the following steps:

**Step 1:** Select a test code change from the code review history that is resolved and its actual reviewers are known (Described in section 7.4.3).

**Step 2:** Select completed code reviews from the review system before the test code change was submitted but not yet reviewed.

**Step 3:** Use *cHRev* to collect a ranked list of reviewers from Step 2.

**Step 4:** Compare the results of Step 3 with the baseline. The reviewers who reviewed the test code change are considered the baseline.

**Step 5:** Repeat the above steps for N test code changes in the established benchmark.

**Step 6:** Compute precision, recall, F-score, and MRR metrics from Steps 4 and 5.

*REVFINDER*, *RevCom*, and *xFinder* are evaluated with the same protocol except that *RevCom*, and *xFinder* form their expertise models with the inclusion of past commits. *xFinder* uses past commits instead of past reviews, and *RevCom* uses a combination of past commits and reviews.

### 7.4.5 Accuracy Metrics and Hypothesis Testing

To investigate the research question *RQ1*, we evaluated the accuracy of *cHRev*, *REVFINDER*, *xFinder*, and *RevCom* for all of the code changes in our benchmark using the precision, recall, Mean Reciprocal Rank (MRR), and F-score (derived from precision and recall) metrics, which were used previously [9, 65, 106]. For each code change *p*, in a set of code changes $P$ of size $n$, from the benchmark of each subject system and $m$ number of recommended reviewers, the formula for precision@m, recall@m, and F-score@m are given below:

$$precision@m = \frac{\mid RR(p) \cap AR(p) \mid}{\mid RR(p) \mid} \tag{7.7}$$

$$recall@m = \frac{\mid RR(p) \cap AR(p) \mid}{\mid AR(p) \mid} \tag{7.8}$$

$$F\_score@m = 2.\frac{precision@m.recall@m}{precision@m + recall@m} \tag{7.9}$$

where *RR(p)* and *AR(p)* are the recommended reviewers and the actual reviewers who contributed in the review process of the code change *p* respectively. This metric is computed for

recommendation lists of reviewers with different sizes (e.g., $m = 1$, $m = 2$, $m = 3$, and $m = 5$ reviewers).

Table 7.3 shows the frequency distribution of reviewers for each subject software system in our benchmark[6]. 69% of code changes for *Android Platform*, 76% of code changes for *Eclipse Platform*, 71% of code changes for *Mylyn*, and 64% of code changes for *MS Office* are reviewed by a single (and not necessarily the same) reviewer. In such a scenario, each increment to *m* in pursuit of a correct reviewer could add to the proportion of false positives. A complimentary measure is also needed to assess the potential effort in addressing noise (false positives). We focused on evaluating the ranked positions of the correct reviewers for each code change for each benchmark from a cumulative perspective regardless of the cutoff point $m$. Mean Reciprocal Rank (MRR) is one such measure that can be used for evaluating any process that produces a list of possible responses to a sample of queries, ordered by probability of correctness. The reciprocal rank of a query response is the multiplicative inverse of the rank of the first correct answer. Intuitively, the lower the value (between 0 and 1), the farther down the list, examining incorrect responses along the way, one needs to search to find a correct response.

$$MRR = \frac{1}{\mid n \mid} \sum_{i=1}^{|n|} \frac{1}{rank_i} \tag{7.10}$$

Here, the reciprocal rank for a query (code change) is the reciprocal of the position of the correct reviewer in the returned ranked list of reviewers ($rank_i$) and $n$ is the total number of code changes in our benchmark. When the correct reviewer for a code change is not recommended at all, we consider its inverse rank to be a zero. When there are multiple correct reviewers, we consider the highest/first ranked position. The higher the value of MRR, the better it speaks of the potential effort spent in noise. For example, the MRR value of 0.5 suggests that the average correct answer is found at the second rank.

Further, we define the following null hypotheses for our study for both closed and open

[6]http://serl.cs.wichita.edu/svn/projects/CodeReview/CodeReview/trunk/Data

Table 7.3: Evaluation benchmarks and the distribution of reviewers per review (code change)

| System | Frequency distribution | | | | | | Total Reviews |
|---|---|---|---|---|---|---|---|
| | # 1 | # 2 | # 3 | # 4 | # 5 | # 6 | |
| *Mylyn* | 113 | 33 | 12 | 1 | 1 | 0 | 160 |
| *Eclipse Platform* | 98 | 24 | 7 | 0 | 0 | 0 | 129 |
| *Android Platform* | 105 | 30 | 15 | 3 | 0 | 0 | 153 |
| *MS Office* | 538 | 219 | 66 | 16 | 6 | 0 | 845 |

source domains to assess the statistical validity of the results (the alternative hypotheses can be easily derived from the respective null hypotheses):

**H-1:** There is no SSD between the precision@m, recall@m, F-score@m, and MRR values of *cHRev* and *REVFINDER*.

**H-2:** There is no SSD between the precision@m, recall@m, F-score@m, and MRR values of *cHRev* and *xFinder*.

**H-3:** There is no SSD between the precision@m, recall@m, F-score@m, and MRR values of *cHRev* and *RevCom*.

**H-4:** There is no SSD between the precision@m, recall@m, F-score@m, and MRR values of *RevCom* and *xFinder*.

We applied the One Way ANOVA test to assess the statistically significant difference (SSD) with $\alpha = 0.05$ between the results of precision, recall and MRR values of the compared approaches. For MRR, we considered the ranks of correct answers of the approaches for each code change (data point). The purpose of the test is to assess whether the distribution of one of the two samples is stochastically greater than the other.

### 7.4.6 Results

The number of recommended reviewers is the only user defined parameter for our approach. As can be seen from Table 7.3, the maximum number of reviewers in both closed and open source systems is bounded by five in the benchmarks. Therefore, the experiment was run for $m = 1$,

Table 7.4: Average of precision, recall, and F-score @1, 2, 3 and 5 values of the approaches *cHRev*, *REVFINDER*, *xFinder*, and *RevCom* measured on the benchmarks.

| System | | Precision@m | | | | Recall@m | | | | F-score@m | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $m$ | *cHRev* | *REVFINDER* | *xFinder* | *RevCom* | *cHRev* | *REVFINDER* | *xFinder* | *RevCom* | *cHRev* | *REVFINDER* | *xFinder* | *RevCom* |
| ***Mylyn*** | 1 | 0.59 | 0.36 | 0.43 | 0.55 | 0.48 | 0.26 | 0.34 | 0.45 | 0.53 | 0.30 | 0.38 | 50 |
| | 2 | 0.50 | 0.27 | 0.43 | 0.50 | 0.64 | 0.37 | 0.45 | 0.66 | 0.56 | 0.31 | 0.44 | 0.57 |
| | 3 | 0.48 | 0.23 | 0.40 | 0.48 | 0.81 | 0.47 | 0.48 | 0.81 | 0.60 | 0.31 | 0.44 | 0.60 |
| | 5 | 0.41 | 0.19 | 0.39 | 0.41 | 0.87 | 0.67 | 0.56 | 0.87 | 0.56 | 0.30 | 0.46 | 0.56 |
| ***Eclipse Platform*** | 1 | 0.44 | 0.44 | 0.28 | 0.43 | 0.38 | 0.36 | 0.25 | 0.36 | 0.41 | 0.40 | 0.26 | 0.39 |
| | 2 | 0.40 | 0.33 | 0.30 | 0.38 | 0.61 | 0.55 | 0.46 | 0.60 | 0.48 | 0.41 | 0.36 | 0.47 |
| | 3 | 0.37 | 0.27 | 0.26 | 0.34 | 0.76 | 0.67 | 0.5 | 0.72 | 0.50 | 0.38 | 0.34 | 0.46 |
| | 5 | 0.31 | 0.20 | 0.24 | 0.28 | 0.82 | 0.75 | 0.62 | 0.80 | 0.45 | 0.32 | 0.35 | 0.41 |
| ***Android Platform*** | 1 | 0.50 | 0.34 | 0.23 | 0.48 | 0.27 | 0.18 | 0.19 | 0.26 | 0.35 | 0.24 | 0.21 | 0.34 |
| | 2 | 0.41 | 0.29 | 0.17 | 0.39 | 0.42 | 0.31 | 0.28 | 0.41 | 0.41 | 0.30 | 0.21 | 0.40 |
| | 3 | 0.35 | 0.25 | 0.14 | 0.34 | 0.50 | 0.39 | 0.31 | 0.49 | 0.41 | 0.30 | 0.19 | 0.40 |
| | 5 | 0.30 | 0.22 | 0.11 | 0.28 | 0.61 | 0.48 | 0.37 | 0.60 | 0.40 | 0.30 | 0.17 | 0.38 |
| ***MS Office*** | 1 | 0.59 | 0.38 | 0.23 | 0.57 | 0.42 | 0.25 | 0.16 | 0.40 | 0.49 | 0.30 | 0.19 | 0.47 |
| | 2 | 0.47 | 0.33 | 0.18 | 0.46 | 0.60 | 0.43 | 0.23 | 0.60 | 0.53 | 0.37 | 0.20 | 0.52 |
| | 3 | 0.37 | 0.26 | 0.16 | 0.37 | 0.68 | 0.51 | 0.27 | 0.68 | 0.48 | 0.34 | 0.20 | 0.48 |
| | 5 | 0.29 | 0.22 | 0.13 | 0.28 | 0.75 | 0.72 | 0.29 | 0.75 | 0.42 | 0.34 | 0.18 | 0.41 |

Table 7.5: Average of precision, recall, and F-score gains @1, 2, 3 and 5 values of the approaches *cHRev*, *REVFINDER*, *xFinder*, and *RevCom* measured on the benchmarks.

| System | | Precision | | | | Recall | | | | F-score | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Gain*PcHRev-* | | | Gain*PRevCom-* | Gain*RcHRev-* | | | Gain*RRevCom-* | Gain*FcHRev-* | | | Gain*FRevCom-* |
| | $m$ | *REVFINDER%* | *xFinder%* | *RevCom%* | *xFinder%* | *REVFINDER%* | *xFinder%* | *RevCom%* | *xFinder%* | *REVFINDER%* | *xFinder%* | *RevCom%* | *xFinder%* |
| ***Mylyn*** | 1 | 63.88 | 37.21 | 7.27 | 27.90 | 84.61 | 41.18 | 6.66 | 32.25 | 76.66 | 39.47 | 6 | 31.57 |
| | 2 | 85.18 | 16.28 | 0 | 16.28 | 72.97 | 42.22 | -3.03 | 46.66 | 80.64 | 27.27 | -1.78 | 29.54 |
| | 3 | 108.69 | 20.00 | 0 | 20.00 | 72.34 | 68.75 | 0 | 68.75 | 93.54 | 36.36 | 0 | 36.36 |
| | 5 | 115.78 | 5.13 | 0 | 5.13 | 29.85 | 55.35 | 0 | 55.35 | 86.66 | 21.73 | 0 | 21.73 |
| ***Eclipse Platform*** | 1 | 0 | 57.14 | 2.32 | 53.57 | 5.55 | 52.00 | 5.55 | 44.00 | 2.5 | 57.69 | 5.12 | 50.00 |
| | 2 | 21.21 | 33.33 | 5.26 | 26.66 | 10.90 | 32.61 | 1.66 | 30.43 | 17.07 | 33.33 | 2.12 | 30.55 |
| | 3 | 37.03 | 42.31 | 8.82 | 30.76 | 13.43 | 52.00 | 5.55 | 44.00 | 31.57 | 47.05 | 8.69 | 35.29 |
| | 5 | 55 | 29.17 | 10.71 | 16.66 | 9.33 | 32.26 | 2.5 | 29.03 | 40.62 | 28.57 | 9.75 | 17.14 |
| ***Android Platform*** | 1 | 47.05 | 117.39 | 4.16 | 108.69 | 50.00 | 42.10 | 3.84 | 36.84 | 45.83 | 66.67 | 2.94 | .61.90 |
| | 2 | 41.37 | 141.17 | 5.12 | 129.41 | 35.48 | 50.00 | 2.43 | 46.42 | 36.67 | 95.24 | 2.50 | 90.48 |
| | 3 | 40.00 | 150.00 | 2.94 | 142.85 | 28.20 | 61.29 | 2.04 | 58.66 | 36.67 | 115.79 | 2.50 | 110.53 |
| | 5 | 36.36 | 172.72 | 7.14 | 154.54 | 27.08 | 64.86 | 1.66 | 62.16 | 33.33 | 135.29 | 5.26 | 123.53 |
| ***MS Office*** | 1 | 55.26 | 156.52 | 3.51 | 147.83 | 68.00 | 162.5 | 5 | 150.00 | 63.33 | 157.89 | 4.26 | 147.37 |
| | 2 | 42.42 | 161.11 | 2.17 | 155.55 | 39.53 | 160.87 | 0 | 160.87 | 43.24 | 165.00 | 1.92 | 160.00 |
| | 3 | 42.30 | 131.25 | 0 | 131.25 | 33.33 | 151.85 | 0 | 151.85 | 41.18 | 140.00 | 0.00 | 140.00 |
| | 5 | 31.81 | 123.07 | 3.57 | 115.38 | 4.16 | 158.62 | 0 | 158.62 | 23.53 | 133.33 | 2.44 | 127.78 |

Table 7.6: Mean Reciprocal Rank of the approaches *cHRev*, *REVFINDER*, *xFinder*, and *RevCom* measured on the benchmarks.

| **System** | **MRR** | | | | **Gain*cHRev-*** | | **Gain*RevCom-*** | |
|---|---|---|---|---|---|---|---|---|
| | *cHRev* | *REVFINDER* | *xFinder* | *RevCom* | ***REVFINDER%*** | ***xFinder%*** | ***RevCom%*** | ***xFinder%*** |
| ***Mylyn*** | 0.72 | 0.52 | 0.51 | 0.71 | 38.46 | 41.18 | 1.40 | 39.20 |
| **Eclipse** | 0.63 | 0.58 | 0.46 | 0.62 | 8.62 | 36.96 | 1.61 | 34.78 |
| ***Android*** | 0.65 | 0.49 | 0.35 | 0.63 | 32.65 | 85.71 | 3.17 | 80.00 |
| ***MS Office*** | 0.70 | 0.56 | 0.29 | 0.69 | 25.00 | 141.37 | 1.44 | 137.93 |

Table 7.7: p-values from applying one way ANOVA on Precision@m and Recall@m values for each subject system.

| **System** | $m$ | **Precision p-value** | | | | **Recall p-value** | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | ***cHRev-*** | | | ***RevCom-*** | ***cHRev-*** | | | ***RevCom-*** |
| | | ***REVFINDER*** | ***xFinder*** | ***RevCom*** | ***xFinder*** | ***REVFINDER*** | ***xFinder*** | ***RevCom*** | ***xFinder*** |
| ***Mylyn*** | 1 | $< 0.01$ | $< 0.01$ | $\leq 0.4$ | $< 0.01$ | $< 0.01$ | $< 0.01$ | $\leq 0.8$ | $< 0.01$ |
| | 2 | $< 0.01$ | $< 0.01$ | $\leq 0.9$ | $< 0.01$ | $< 0.01$ | $\equiv 0.00$ | $\leq 0.9$ | $\equiv 0.00$ |
| | 3 | $\equiv 0.00$ | $< 0.01$ | $\leq 0.9$ | $< 0.01$ | $\equiv 0.00$ | $\equiv 0.00$ | $\leq 1$ | $\equiv 0.00$ |
| | 5 | $\equiv 0.00$ | $\leq 0.7$ | $\leq 0.9$ | $\leq 0.7$ | $\equiv 0.00$ | $\equiv 0.00$ | $\leq 1$ | $\equiv 0.00$ |
| ***Eclipse Platform*** | 1 | $\leq 0.1$ | $< 0.01$ | $\leq 0.8$ | $< 0.01$ | $\leq 0.4$ | $< 0.02$ | $\leq 0.7$ | $< 0.04$ |
| | 2 | $< 0.02$ | $< 0.01$ | $\leq 0.5$ | $< 0.03$ | $< 0.04$ | $< 0.01$ | $\leq 0.9$ | $< 0.01$ |
| | 3 | $< 0.02$ | $< 0.01$ | $\leq 0.3$ | $< 0.01$ | $< 0.03$ | $\equiv 0.00$ | $\leq 0.4$ | $< 0.01$ |
| | 5 | $\equiv 0.00$ | $< 0.04$ | $\leq 0.3$ | $< 0.03$ | $< 0.02$ | $\equiv 0.00$ | $\leq 0.9$ | $\equiv 0.00$ |
| ***Android Platform*** | 1 | $< 0.01$ | $< 0.01$ | $\leq 0.8$ | $< 0.01$ | $< 0.03$ | $< 0.01$ | $\leq 0.9$ | $< 0.01$ |
| | 2 | $< 0.01$ | $< 0.01$ | $\leq 0.7$ | $< 0.01$ | $< 0.02$ | $\equiv 0.00$ | $\leq 0.9$ | $\equiv 0.00$ |
| | 3 | $< 0.01$ | $\equiv 0.00$ | $\leq 0.7$ | $\equiv 0.00$ | $< 0.01$ | $\equiv 0.00$ | $\leq 0.9$ | $\equiv 0.00$ |
| | 5 | $< 0.01$ | $\equiv 0.00$ | $\leq 0.7$ | $\equiv 0.00$ | $< 0.01$ | $\equiv 0.00$ | $\leq 0.9$ | $\equiv 0.00$ |
| ***MS Office*** | 1 | $< 0.01$ | $\equiv 0.00$ | $\leq 0.7$ | $\equiv 0.00$ | $< 0.01$ | $\equiv 0.00$ | $\leq 0.7$ | $\equiv 0.00$ |
| | 2 | $< 0.02$ | $\equiv 0.00$ | $\leq 0.8$ | $\equiv 0.00$ | $< 0.02$ | $\equiv 0.00$ | $\leq 0.9$ | $\equiv 0.00$ |
| | 3 | $< 0.02$ | $\equiv 0.00$ | $\leq 0.9$ | $\equiv 0.00$ | $< 0.02$ | $\equiv 0.00$ | $\leq 0.9$ | $\equiv 0.00$ |
| | 5 | $< 0.03$ | $\equiv 0.00$ | $\leq 0.9$ | $\equiv 0.00$ | $\leq 0.04$ | $\equiv 0.00$ | $\leq 0.9$ | $\equiv 0.00$ |

Table 7.8: p-values from applying one way ANOVA on MRR values for each subject system.

| **System** | **MRR p-value** | | | |
|---|---|---|---|---|
| | ***cHRev-*** | | | ***RevCom-*** |
| | ***REVFINDER*** | ***xFinder*** | ***RevCom*** | ***xFinder*** |
| ***Mylyn*** | $< 0.01$ | $\equiv 0.00$ | $\leq 0.7$ | $\equiv 0.00$ |
| ***Eclipse*** | $< 0.04$ | $< 0.01$ | $\leq 0.9$ | $< 0.01$ |
| ***Android*** | $\equiv 0.00$ | $\equiv 0.00$ | $\leq 0.7$ | $\equiv 0.00$ |
| ***MS Office*** | $< 0.01$ | $\equiv 0.00$ | $\leq 0.9$ | $\equiv 0.00$ |

$m = 2$, $m = 3$, and $m = 5$, where $m$ is the number of recommended reviewers to provide the realistic view of the performance.

To answer the research $RQ1$, we consult Table 7.4. The highest precision is for the lowest value of $m$ and the highest recall is for the highest value of $m$. The decrease or increase in precision and recall with increase in the value of $m$ is gradual (and no drastic changes were noted). Note that while computing recall for lower values of $m$ (e.g., $RR(p)$=1 for $m$=1), we considered all the correct reviewers for a patch (e.g., $AR(p)$=3). Therefore, the recall at such values could be lower despite making all the correct recommendations. Furthermore, the accuracy performance of *cHRev* is consistent across closed (*MS Office*) and open source (*Android Platform*, *Eclipse Platform*, and *Mylyn*) systems. With regard to MRR values, we consult Table 7.6. *cHRev* gives the value of greater than 0.5 for all the four systems. That is, on average a maximum of two recommendations need to be examined to get the first correct reviewer. These results indicate the stability of *cHRev* across systems with different sizes, test sets, and domains.

***RQ1*** *cHRev makes accurate reviewer recommendations in terms of precision and recall. On average, less than two recommendations are needed to find the first correct reviewer in both closed and open source systems.*

To investigate the research question *RQ2*, we computed the metric gain of *cHRev* (i.e., X equals to precision, recall, F-score, or MRR) over another compared approach (i.e., Y equals to *REVFINDER*, *xFinder*, or *RevCom*) using the following formula:

$$GainX@m_{cHRev-Y} = \frac{X@m_{cHRev} - X@m_Y}{X@m_Y} \times 100 \tag{7.11}$$

Tables 7.5 and 7.6 show the precision, recall, F-score, and MRR gain values. Clearly, *cHRev* outperforms *REVFINDER* across precision, recall, F-socre, and MRR values in all the four systems. *cHRev* records positive gains with statistical significance (with p-values$<$0.05) in all cases, except precision@$m = 1$, recall@$m = 1$, and F-score@$m = 1$ for *Eclipse Platform* (see Tables 7.7 and 7.8). In these exceptional cases, both were statistically equivalent. The gains in *Eclipse Platform* are generally lower than those in *Android Platform*, *Mylyn*, and *MS Office*. We only considered a single component of *Eclipse Platform* and was the smallest dataset in our evaluation. The met-

hodology of *REVFINDER* should have favored such a dataset because the file names in a single component are typically similar (and thus, the reviewers). However, our *cHRev* approach was able to perform better than *REVFINDER* in even such a favorable setting. Therefore, these results suggest that the amount of comments, workdays need to make them, and their recency contribute to higher accuracy than simply looking at similar file names and paths. Therefore, we find support to reject Hypothesis **H-1** in favor of *cHRev*.

Clearly, *cHRev* outperforms *xFinder* across precision, recall, F-score, and MRR values in all the four systems. It is remarkable to note that the precision and recall gains of *cHRev* over *xFinder* on *MS Office* (well over 100%) are substantially better than those achieved on *Android Platform*, *Eclipse Platform*, and *Mylyn* (well below 100%). This fact suggests that *cHRev* could offer a much better solution in the commercial domain. All the precision and recall gains for different values of $m$ (with the exception of *Mylyn* precision at $m$=5), and MRR gains are statistically significant (i.e., p-values$<$0.05). The only case of *Mylyn* where there is no statistically significant gain happens at the largest value of $m$, where precision was the lowest in both approaches. Nonetheless, *cHRev* is no worse than *xFinder* in this exceptional case. Therefore, we find support to reject Hypothesis **H-2** in favor of *cHRev*. Note that the same cannot be said about the gains of *REVFINDER* over *xFinder*. *REVFINDER* did not register a single positive precision or recall gain over *xFinder* in *Mylyn*, which was the largest considered open source dataset.

In case of the comparison between *cHRev* and *RevCom*, a negative gain would indicate *RevCom* doing better than *cHRev* and a positive gain would indicate *cHRev* doing better than *RevCom*. Clearly, the gains (with the exception of *Mylyn* recall and F-score at $m$=2) are positive. Contrary (and perhaps surprisingly) to many successful results from various combined approaches in other tasks studies, the combination of reviews and commits was not very effective. In fact, our results indicate that a combined approach could be detrimental (i.e., could lead to a drop in precision and recall). The statistical testing showed that the gains are not significant (p-values $>$0.05). Nonetheless, the results show that the combined approach *RevCom* is no better than our approach *cHRev*. Therefore, we find support to accept Hypothesis **H-3** in favor of *cHRev*.

Concerned with the potential drop in precision and recall, we continued our investigation of the research question $RQ2$. We did a similar analysis to compute the gains of *RevCom* over *xFinder* to ascertain that the combination would be more effective than *xFinder*. On a successful note, we found that all the gains are statistically significant (with the exception of *Mylyn* precision at $m$=5). Therefore, *RevCom* outperforms *xFinder*. It is worth noting, however, that the gains of *RevCom* over *xFinder* could be lower than those of *cHRev* over *xFinder*. This behavior can be seen in the precision and recall results of *Android Platform*, *Eclipse Platform*, and *MS Office*. Our results suggest to exercise caution about treating the combination and review based recommenders to be identical in performance. Overall, we find support to reject Hypothesis **H-4**.

***RQ2:*** *cHRev performs much better than REVFINDER which is based on reviewers of files with similar names and paths and xFinder which relies on source code repository data, and cHRev is statistically equivalent to RevCom which requires both past reviews and commits.*

### 7.4.7 Discussion

Here we discuss a few points that would help in our understanding of the rationale behind the improved performance with using reviews in *cHRev*. The reasons could be attributed to two-fold aspects.

First, unlike *REVFINDER*, *cHRev* includes the number of individual days that a reviewer provided feedback and also the time since the most recent review on each file. Both techniques use the number of past reviews on a changed file under current review to model expertise; however *cHRev* also uses the number of comments in each review, the number of days that a reviewer has made comments on a file under review (sometimes multiple workdays for one review) because prolonged examination of a source code file could indicate the increased level of expertise. Further, research has shown that expertise in an area of code dwindles with time [39] and thus we incorporate recency, the amount of time since the last review of a file by a potential reviewer, into our approach. Moreover, *cHRev* was able to recommend reviewers in an overwhelming majority of the cases at the file level.

Second, for all of the projects studied, we found many cases where reviewers provided quality feedback despite the fact that they had never made changes to the files or directories under

review. We manually investigated reasons why these people might have the expertise to provide such feedback as reviewers.

**The Broader Community**: In *Android Platform*, *Mylyn*, and *Eclipse Platform* there are a limited number of people with permissions to make code changes, but a larger group that contributes patches, participates in bug reporting, or provide feedback on future design plans. Because they are still involved in the project in a technical way, they have expertise that is useful for reviewing changes.

**Project Leaders**: In all of the projects that we examined there are project leaders (known as "development leads" at Microsoft) who are experienced developers and have intimate knowledge of the different systems within the project. They often act as reviewers and provide feedback about changes even though they had never changed the actual files under review.

**Testers and Managers**: In *MS Office*, we observed that testers and program managers participated in reviews quite often. While these people do not work on shipping code, their job responsibilities require that they take an active interest in changes. Testers must write tests that exercise the code under review and program managers manage dependencies and interfaces between various systems in the code.

**Developers of Related Code**: Source code files do not exist in a vacuum. Most source code depends on and is depended on by other parts of the system. The associated developers have an interest in such changes. We observed many cases in which the change to a piece of code is reviewed by developers that work in related code (e.g., code that has a dependency on the changed code). This occurred in all four projects studied.

**Developers of Unaccepted Contributions**: Given the nature of OSS, there are often multiple attempts at resolving a given change request. For example, multiple (a few incomplete or incorrect) fixes are attempted by perhaps multiple developers. In the end, only a complete and correct resolution is accepted and/or merged into the source code repository (i.e., the main development trunk or branch). Of course, the commit history only records the final outcome (i.e., only the things committed). Gousios et al. [43] observed that in GitHub some issues receive multi-

ple pull-requests with solutions, but not all are accepted and merged. Our results show that past experiences (including failures) are important ingredients in shaping reviewer expertise.

In all of the cases described above, the source code repository does not capture activity reflective of the expertise of various team members. These people do participate in code reviews either as reviewers (most cases) or as authors of changes that are never accepted, but which are examined by the community. Thus, there are traces of their expertise in the review history. This is not surprising, as Rigby et al. [82] found that project participants in both industrial and OSS contexts are exposed to more source code through review than through making changes to the code (exposed to 44% to 150% more files on average). All of these observations support the conclusion that relying on the commit history of a source code repository carries the threat of missing potential reviewers that have valuable expertise. Similarly the number of developers who authored commits and the number of reviewers who participated in review process reveal the difference between the provided information from commit history and review history. We calculated the The Jaccard similarity between the list of reviewers from the review history and the list of developers from the commit history to explore their differences. The Jaccard values for *Android Platform*, *Eclipse Platform*, and *Mylyn* are 0.55, 0.45, and 0.67 respectively. These values show that these two lists are quite dissimilar.

During our study, we noted that the impact of using review data over commit data was more pronounced in *MS Office*. This is most likely due to the way that responsibility of code is handled at Microsoft. Bird et al. [26] found that strong ownership practices are employed at Microsoft. As a result, it is quite common that the owner of a particular piece of code may be the only one to have touched it in a long time and in some cases ever. In these cases, commit history is unlikely to provide much help in identifying a reviewer. However, expertise as exhibited in prior reviews from members falling into the groups discussed above are captured by *cHRev*, leading to improved recommendations. The fact that *cHRev* outperformed *xFinder* in our study reveals that considering only the code ownership (code commits) will provide a suboptimal solution for recommending reviewers. Additionally, *cHRev* outperformed *RevCom*, which indicates that

combining the code ownership and review features may not necessarily improve the accuracy of recommending reviewers.

## 7.5 Threats to Validity

We will now discuss the internal, construct, and external threats to validity of the results of our empirical study.

**Considering Review Comments for Entire Patch**: We only considered the review comments that were written for the entire code change because it was the case in our subject systems. For *Android Platform*, *Eclipse Platform*, and *Mylyn* projects, most review comments are for the entire code change and the number of in-line comments is low in comparison (20% in-line comments). We did not do a precise mapping of these comments to individual files. This fact may have given less relevant and more irrelevant weights to certain files. We plan to study systems with inline review comments in the future.

**Verifying or Reviewing the Code**: In *Gerrit* code review system, reviewers can get two different roles: reviewing the code or verifying the code. Verification is generally related to running the test cases. We did not separate reviewers based on their roles. It is possible that separating the reviewers based on their role could affect the results.

**Correctness of Reviewer Recommendations**: We considered a gold-set to be reviewers who contributed in reviews to a given code change and not those reviewers who are assigned to review the code change. We considered reviewers who contributed at least a comment and assigned reviewers because not all (or any) of the assigned reviewers may eventually participate in the review process. We do not know that these reviewers were the best nor other reviewers were equally capable (but did not contribute due to issues such as workload and schedule). However, creating a gold set accounting for such factors is a challenging issue. Recently different metrics for recommender systems were defined such as diversity [109]. We plan to incorporate this metric in the future work.

**Reviewer Identity Mismatch**: Although we carefully examined the available sources of information to match the different identities of the same reviewer, it is possible that we missed or

mismatched a few cases. There are several cases which developer's IDs are different in *git* and *Gerrit* repositories.

**Incongruent History**: Although, a common period was considered for extracting the review and commit datasets, the number of commit transactions is higher than the number of review transactions. There are several cases in which the code change was directly merged into the git repository without going through the review process. For the open source projects commits were available before the reviews. It is possible that these datasets are not reflective of the optimum results.

**Single Period of History**: We considered one period of history for each system (see Section 7.4.3 for the specific reasons); however, we do not claim that our results would hold equally well for other chosen periods of history. A different history period might produce different results in terms of their relative performance.

**Generalization**: Although we investigated both closed and open source systems, we do not claim that our results would generalize to every software system in these domains.

## 7.6 Related Work

The reviewer recommendation task has not been examined much in the literature yet.

There are only three approaches reported for reviewer recommendations. Balachandran [14] proposed a GIT blame like line oriented approach. Recently, Thongtanunam et al. [96] proposed an approach, namely *REVFINDER*, which is based on the past reviews of files with similar names and paths. They showed that *REVFINDER* outperformed Balachandran's approach on open source systems. *REVFINDER* finds past reviews with files whose paths and names are similar (based on string comparison) to the ones in the patch under review. It assigns all the reviewers from each such past review the same (string comparison) score. All the reviewers are ranked based on the sum of their scores. It does not look into other attributes of the past contributions of the reviews (e.g., how much and when) and is limited to whether a reviewer contributed or not. In summary, *REVFINDER*'s expertise model favors *breadth or generality* of review contributions. In parallel to our work, Xia et al. [98] proposed another approach for reviewer recommendation, namely TIE.

The intuition of TIE is that the same reviewers are likely to review changes containing similar terms (words) and reviewers are likely to review changes to the same files or files in similar locations. TIE outperformed *REVFINDER* on open source systems. Similar to *REVFINDER*, TIE approach just look into the similarity of patch description and file path. Unlike them, *cHRev* does not need textual information from code changes. Furthermore, TIE does not account for attributes such as the amount of comments and their recency). Furthermore, our empirical evaluation included the closed-source domain and comparison with a closely related methodology of developer recommendation.

**Key difference between *REVFINDER* and our *cHRev* approach**

*cHRev* looks at the specific contribution of each reviewer in past reviews on the code under review. This contribution is quantified using the numbers of feedback comments and days, and their recency. In summary, *cHRev*'s expertise model favors *depth or specificity* of review contributions. The implication of the breadth and depth difference on the performance is that *REVFINDER* may end up with too many generic recommendations of package/subsystem owners (or gatekeepers) at the expense of too few specific contributors, including developers and leads who focus on a narrower code base. Our results on commercial and open source systems suggest that the depth analysis of past reviews leads to improved accuracy of recommendations.

### 7.6.1 Developer Recommendation

The task of automatically assigning issues or change requests (e.g., bug fixes or new feature) to the developer(s) who are most likely to resolve them has been studied under the umbrella of issue triaging. A number of approaches exit in the literature [9, 106, 47, 7, 35, 16, 8, 10, 111]. Approaches for developer recommendation typically operate on software repositories (e.g., models trained from past bugs/issues or source-code changes), the source-code authorship, or their combinations.

While similar to developer recommendation, reviewer recommendation is in a different domain. The developer recommendation task is in domain of resolving a bug and the reviewer recommendation is in domain of reviewing a change. Issue triage is a crucial activity in addressing

change requests in an effective manner (e.g., within time, priority, and quality factors). One simple way to view the difference is that the developer recommendation occurs (developers/owners to resolve the change request) before the reviewer recommendation (reviewers to review the changed code to address the change request). Our results from both open and closed source domains show that commit history is insufficient for gauging the needed reviewers. It is necessary to utilize past code reviews to find the appropriate reviewers accurately.

# CHAPTER 8

## Patch Acceptance Predictor

Large-scale open source projects typically receive numerous patches to address change requests (e.g., bug fixes). It is not uncommon for a submitted patch to undergo a peer-review process before it can be accepted as a valid solution and merged to the code base. A peer-code review process has been shown to have impact on the software quality. Determining whether a submitted patch will be accepted or not could improve the efficiency of the peer-review process, e.g., help developers and reviewers prioritize and focus their efforts. In this chapter, we examine bug, patch, developer, and reviewer features that characterize the patches that get accepted or not. These features are extracted from the macro and micro repositories. We use logistic regression to form a descriptive model from these features. Moreover, we present a predictive model that classifies whether a patch will be accepted or not as soon as it is submitted. Logistic regression and support vector machine were employed to form patch predictors.

To validate our approach, we conducted an empirical study on three open source projects *Android Platform*, *Eclipse Platform*, and *Mylyn*. These systems require a peer review of submitted patches, which is managed by *Gerrit*. Our results show that features such as, owner bug assignee match (e.g., the developer who is assigned to resolve the issue in the bug tracking system is the same as the one who implemented the code change and submitted it for review), the reputation of developers in terms of their successful history of patch acceptance, the number of revisions made to the patch, and the number of reviewers greatly influence the likelihood of a patch getting accepted. Our predictor model achieves the average accuracy values of 68%, 93%, and 92% on *Android Platform*, *Eclipse Platform*, and *Mylyn* respectively. Furthermore, we note that support vector machine performs better than logistic regression for patch acceptance prediction.

## 8.1 Introduction

Modern Code Review (MCR) is becoming prevalent across commercial and open source projects [81]. It is a peer-reviewed process of inspecting the code changes, i.e., patches, submitted to resolve changes requests. The benefits of MCR are shown in improving software quality and knowledge transfer, among other things [13]. It is not uncommon in large open source projects to receive numerous change requests (e.g., bug reports or feature requests) daily [9]. Developers submit the code changes or patches to resolve these change requests. These code changes go through an intensive MCR process before they can be merged (or not) to the main code base of the system. It involves human discussion and discourse on critiquing and improving the changes. Human reviewers inspect the code and offer suggestions. The developers of the code changes may revise their patches in response to the reviewers' feedback and submit it for another round of review.

Clearly, MCR is quite human intensive, which may require a substantial amount of effort [82]. Not all changes or patches are eventually accepted and integrated to the code base. For example, in project *Android Platform* only 54% of patches were eventually accepted from 01/21/2009 to 01/26/2014. Therefore, the question arises as what makes patches get accepted or not? There has been a few previous efforts in addressing this question; however, they have not been in the context of MCR [17], [50], [44], [19]. Also, the predictive recommendation and assessment of whether a patch will be eventually accepted or not as soon as it is submitted for MCR were not investigated previously.

In this chapter, we first investigate the factors that influence the patch acceptance. We take into account several direct and indirect measures, which are derived from the structured and unstructured data available from bug tracking (cause and importance of the issue) and code-review repositories (who, what, and how of the quality control of the fixes). As we mentioned in Chapter 2 bug repository is an example of macro repository and code review repository is an example of micro repositories. These measures are then used to formulate a descriptive model. Secondly, we build a classifier to predict whether a patch will be accepted or not as soon as it is submitted for MCR? Understanding the factors and developing a patch-acceptance predictor may help improve MCR,

including the tasks related to developers and reviewers. Knowing the positive likelihood may encourage developers to submit their patches for review promptly. In the case of not so favorable likelihood, they could proactively revise their patches without the need of waiting for reviewers' feedback. From the reviewers' perspective, the likelihood indicator may help tailor their effort and focus on promising or challenging patches. We present predictive models based on two different techniques: Support Vector Machine and Logistic Regression. Additionally, we grouped the features from different domains and analyze their impact within and across groups. Results from several evaluation metrics, such as accuracy, recall, and precision, are reported. More specifically, we investigate the following two research questions:

**RQ1**. Which features characterize the patch that get accepted using a statistical model? Logistic regression is used to infer the fix likelihood from features available throughout the bug tracking system and the patch lifecycle from the *Gerrit* code review system, i.e., a **descriptive model**.

**RQ2**. How well a statistical model can predict whether a patch will be accepted or not? Logistic regression and support vector machine are used to build predictors from the most influential factors available at the time of patch submission, i.e., a **predictive model**.

We formulated and validated the descriptive and predictive models on three open source projects *Android Platform*, *Eclipse Platform*, and *Mylyn*. These systems require a peer review of submitted patches, which is managed by *Gerrit*. Results from the descriptive model show that factors such as, owner bug assignee match (e.g., the developer who is assigned to resolve the issue in the bug tracking system is the same as the one who implement the code change and submit it for review), the reputation of developers in terms of their successful history of patch acceptance, the number of revisions made to the patch, the number of reviewers have a statistically significant role. Factors such as the component, severity, and priority of the reported issue, and the number of changed lines and files in the patch, and reviewers' reputations based on past accepted patches did not have a statistically significant role. Our predictor model achieves the accuracy values of 71%, 94% and 93% on *Android Platform*, *Eclipse Platform*, and *Mylyn* respectively.

The rest of this chapter is organized as follows: Background on modern code review (MCR) and subject systems are discussed in Section 8.2. Features used in the study are explained in Section 8.3. Patch acceptance descriptive model is defined and analyzed in Section 8.4. Patch acceptance predictive model is explained in Section 8.5. Predictive results are presented in Section 8.6. Threats to validity are listed and analyzed in Section 8.7.

## 8.2 Background on Modern Code Review (MCR) and Subject Systems

We first define key terms and then provide a brief background on the patch life cycle in Modern Code Review (MCR).

### 8.2.1 Definitions of Key Terms

We use the following terms:

*Patch:* A patch is a set of changed files submitted to fix a bug or add a new feature.

*Patch Review:* A review is a session or record in the peer review process (e.g., in the *Gerrit* system) to manage the lifecycle of a submitted patch. For example, the author/developer of a patch creates a review while uploading it to the *Gerrit* code review system. The patch lifecycle is maintained in this review entry.

*Merged Patch:* A merged patch is the one that successfully gets through the review process and is merged to the source code repository of the system.

*Abandoned Patch:* An abandoned patch is the one for which none of its revisions are accepted to be merged to the source code repository of the system.

*Patch Owner:* A patch owner or simply owner is the developer who submits the patch for review.

*Patch Reviewer:* A patch reviewer is the developer who is assigned and/or contributed to the patch review.

*Assigned Reviewer:* An assigned reviewer is the one who is selected by the patch owner while uploading the patch.

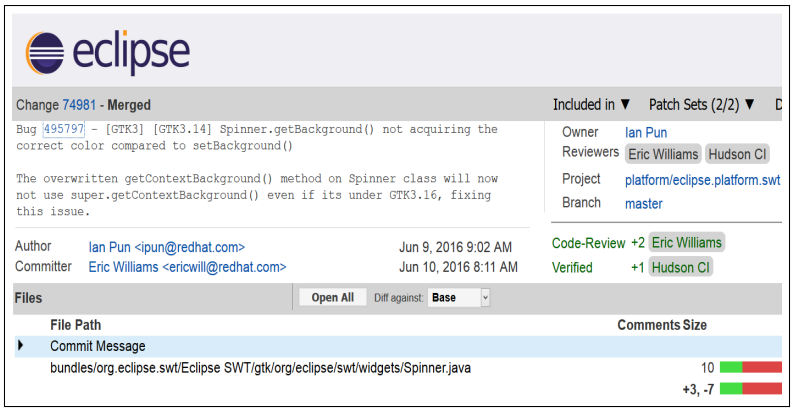


Figure 8.1: An example from *Gerrit* code review# 74981 related to *Eclipse Platform* project

*Contributing Reviewer:* A contributing reviewer is the one who actually participates in the code review process (e.g., by commenting on the patch).

Note that all the assigned reviewers for a patch may not be contributing reviewers. Also, contributing reviewers may not necessarily be assigned reviewers.

### 8.2.2 Patch Lifecycle in *Gerrit*

*Gerrit* is a modern peer-review tool that facilitates a traceable review process for *git*-based software projects [1]. Developers make local changes in their private *git* repositories and then submit these changes as a patch for review [82]. The owner may indicate the intended reviewers, who are subsequently notified about the review invitation. Alternatively, the owner can broadcast a request for review to find reviewers for their patch. It should be noted that the invited reviewers do not necessarily accept the invitation and contribute to the review. The approval step requires human contributions (e.g., to ascertain the patch meets project guidelines and intent), whereas, the verification step is largely automatic (e.g., the build works and complies, and passes unit test cases). If it fails either of these steps, it is abandoned. Figure 8.1 shows an example from *Eclipse Platform Gerrit* for patch review # 74981. It was accepted and merged to the projects's *git* repository. This patch is for bug # 495797 which was assigned to *Ian Pun* for a fix and he patched it. The number of line changes is 10 and the number of changed files is 1. *Eric Williams* reviewed and accepted this patch.

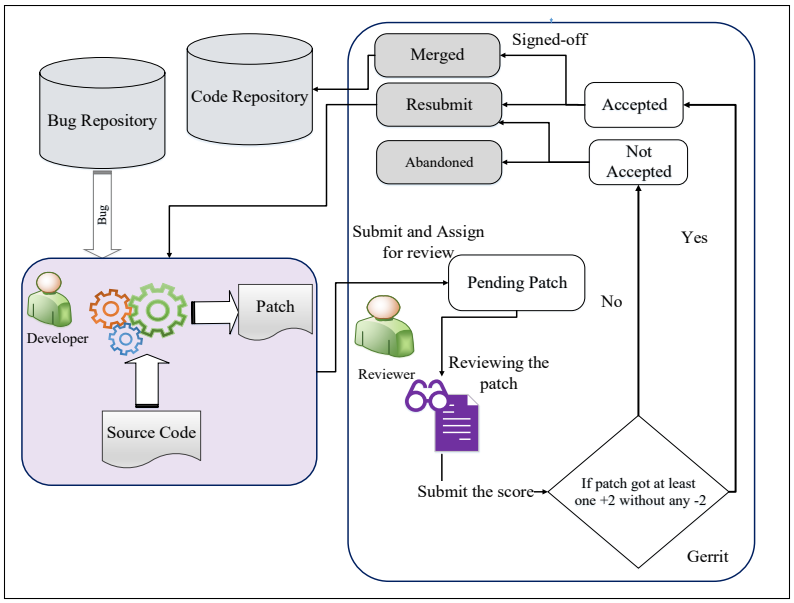


Figure 8.2: The patch life cycle in *Gerrit*

Figure 8.2 shows the patch life cycle in *Gerrit*. Initially, a developer (the patch owner) makes changes to the source code in response to a bug report or feature request. They submit these code changes for review. The patch owner may indicate the intended reviewers (assigned reviewers), who are subsequently notified about the review invitation. It should be noted that the invited reviewers do not necessarily accept the invitation and contribute to the review. Contributing reviewers then inspect the change through the code review tool *Gerrit* and provide feedback in the form of review comments and scores to the owner. A submitted patch can have different statuses such as *Needs Code Review, Ready to Submit, Abandoned, and Merged*. For a patch to get accepted and finally merged to repository, it is necessary that it receive at least one +2 and not a single -2 score.

If a patch receives a score of -2, its status will be marked abandoned. If patch has not received a score -2, then the patch owner can rework the patch and resubmit the new version of patch. The review process for a particular submission may include multiple iterations between the patch owner and contributing reviewers. Eventually, a reviewer signs-off on the review after they perceive the code change to be of a sufficient quality and should be checked into the code

repository. If a change never received sign-offs, it is abandoned. The number of sign-offs required to check in a code change is typically dependent on a particular team's policy. Automatic tools typically perform the verification part.

### 8.2.3 Merged and Abandoned patches

In our work, we need to characterize (describe) and predict whether a given patch will be merged to repository or not. That is, we need two labels: Merged and Abandoned, which are derived from the *Gerrit* code review status. As discussed earlier, reviews in *Gerrit* can have different statuses. In the formulation of our predictor model, we only consider the code reviews with their status Merged or Abandoned. The other statuses are not indicator of a definitive outcome and may indicate work in progress.

### 8.2.4 Subject Software Systems

We focused on *Android Platform*, *Eclipse Platform*, and *Mylyn* in this study. Table 8.1 shows review statistics for these three projects.

***Android Platform*:** Out of different *Android* projects we selected *Android Platform* which contains 6 years of code review started by January 2009 [1]. *Android Platform* is written in C, C++ and java and consists of 4511 code reviews uploaded in *Gerrit* repository which includes at least one source code file. 3660 Out of 4511 reviews have the status "MERGED" or "ABANDONED" which we just consider these reviews in our study. 55% of reviews in *Android Platform* includes patches which finally merged to repository and 45% of reviews are abandoned. totally 44054 review comments are written for *Android Platform* project.

***Eclipse Platform*:** *Eclipse* contains 6 different sub projects, which in this study we just consider *Eclipse Platform*, because it has the most code review history available in compare with other sub projects [2]. *Eclipse Platform* contains about 3 years of code review data in *Gerrit* started by Feb 2012. The *Eclipse Platform* project is written in java and consists of 1902 code reviews uploaded in *Gerrit* repository which includes at least one Java file. 1750 Out of 1902 reviews have

[1] https://android-review.googlesource.com/#/q/platform/frameworks
[2] https://git.eclipse.org/r/#/q/platform,n,z

Table 8.1: Descriptive statistics of the reviews considered from three open source projects in our Study.

| | *Android Platform* | *Eclipse Platform* | *Mylyn* |
|---|---|---|---|
| Start Date | 01/21/2009 | 02/28/2012 | 02/29/2012 |
| End Date | 01/26/2014 | 12/22/2014 | 12/26/2014 |
| #Submitted Patches | 4511 | 1902 | 1617 |
| #Patches Analyzed | 3660 | 1750 | 1549 |
| #Merged Patches | 2010 | 1419 | 1212 |
| #Abandoned Patches | 1650 | 331 | 337 |

the status ”MERGED” or ”ABANDONED” which we just consider these reviews in our study. 81% of reviews in *Eclipse Platform* includes patches which finally merged to repository and 19% of reviews are abandoned. In *Eclipse Platform* project after creating each patch an automatic review comment by tool Hudson will be added to the review. In our experiment these review comments are deleted as noise. After removing noise, totally 14857 review comments are written for *Eclipse Platform* project.

***Mylyn*:** *Mylyn* contains about 3 years of code review data in *Gerrit* and is an Eclipse Foundation project which source code is written in java. Code review history in *Gerrit* started by Feb 2012. The *Mylyn* project consists of 1617 code reviews uploaded in *Gerrit* which includes at least one Java file [3]. 1549 Out of 1617 reviews have the status ”MERGED” or ”ABANDONED” which we just consider these reviews in our study. 78% of reviews in *Mylyn* includes patches which finally merged to the repository and 22% of reviews are abandoned. Same as *Eclipse Platform* project noises are removed from the review comments. After removing noise, totally 14691 review comments are written for *Mylyn* project.

For each of these systems, we removed the patches and reviews related to license updates and major branch restructuring or merges.

[3]https://git.eclipse.org/r/#/q/mylyn,n,z

## 8.3 Features Used in the Study

We describe the features that were extracted (direct) and/or computed (derived) from the code-review (*Gerrit*) and issue-tracking (*Bugzilla*) repositories. These features include the human effort, issue (bug) characteristics, and patch (review) characteristics, and their traceable relationships. These features are divided into three groups: bug features, patch features, and human features.

For selecting features we used some features from [17, 19] and [44] and we extends the features. Features such as severity, owner bug assignee match, file size, entropy score, number of assigned and contributing reviewers, and assigned reviewer's reputation are not investigated at [17, 19]. The only feature which has been investigated in [44] is the entropy score.

### 8.3.1 Bug Features

This category of features is related to the specific bug for which the patch was written to resolve it. In *Mylyn* and *Eclipse Platform*, the traceability between the bug and patch review systems is explicit. That is, the bug id in the bug-tracking system is usually mentioned in the patch description in the code-review system. Some of the reviews in *Android Platform* include the bug id but these bug ids are related to *Android Platform*'s internal bug tracking system. Bug ids in *Android Platform* internal bug tracking system are not the same ids in *Android Platform* public bug tracking system [4]. Therefore, we could not extract bug features for *Android Platform*. This bug information for *Mylyn* and *Eclipse Platform* is available at *Eclipse* bug tracking system [5].

**Bug Component:** In the *Eclipse* bug tracking system, bugs are categorized into classification, products, and components. Components are defined as the second-level of organization. We did not consider the product level because *Eclipse Platform* is already a product of *Eclipse*, hence defining the product level for *Eclipse Platform* would be meaningless. As discussed earlier, there is no trace to the bug id for each review in android. Therefore, the product level can not be considered for *Android Platform* and *Eclipse Platform*.

*Eclipse Platform* and *Mylyn* have different components and different frequencies of bugs

[4] https://code.google.com/p/android/issues/list
[5] https://bugs.eclipse.org/bugs/

Table 8.2: Descriptive statistics of the feature components considered from *Mylyn* and *Eclipse Platform*.

| *Mylyn* | | *Eclipse Platform* | |
|---|---|---|---|
| **Component** | **%FrequencyDistribution** | **Component** | **%FrequencyDistribution** |
| Framework | 22 | UI | 71 |
| Gerrit Connector | 13 | SWT | 11 |
| Wikitext | 11 | Runtime | 4 |
| UI | 9 | IDE | 2 |
| Bugzilla | 5 | Resources | 2 |
| Other | 15 | Other | 3 |
| None | 25 | None | 7 |

and patch reviews. TABLE 8.2 shows the component frequency distribution for analyzed reviews from *Mylyn* and *Eclipse Platform*. The *None* type is for reviews with no explicit link to bug id. The *Other* type accounts for those reviews with less than 1% share. In *Mylyn*, the highest share is 22% for the *Framework* component. In *Eclipse Platform*, the highest share is 71% for the *UI* component. This feature allows us to investigate weather patches for certain components are more successful than others. This feature is treated as categorical in our descriptive and predictive models.

**Bug Severity:** It describes the importance of an issue/bug from the reporter's perspective (e.g., the loss of critical functionality or purely related to vanity) and it shows the level of impact the bug could have. Note that a reporter could be an end user, a developer, or a tester. Therefore, the perspectives span across both application and solution domains. TABLE 8.3 lists the different severity types, which reporters provide while reporting a bug in open source systems [45]. For examples, the first entry is from the solution/implementation domain, the last entry is an enhancement request, and the others indicate different intermediate levels of bug severity. The severity gets increasingly stringent from enhancement to blocker. Figure 8.3 shows the severity distribution between merged and abandoned reviews for both *Mylyn* and *Eclipse Platform* projects. *None* are related to reviews with no traceable bug ids. In *Mylyn*, the severity enhancement has the highest merged reviews and the severity normal has the highest abandoned reviews. In *Eclipse Platform*

Table 8.3: Explanation of bug severity values

| Severity value | Explanation |
|---|---|
| blocker | Blocks development and/or testing work |
| critical | Crashes, loss of data, severe memory leak |
| major | Major loss of function |
| normal | Regular issue, some loss of functionality under specific circumstances |
| minor | Minor loss of function, or other problem where easy workaround is present |
| trivial | Cosmetic problem like misspelled words or misaligned text |
| enhancement | Request for enhancement |

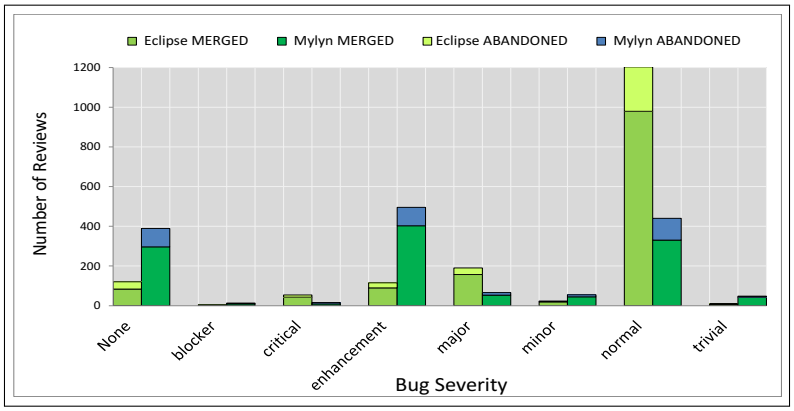


Figure 8.3: Bug severity frequency distribution for merged and abandoned reviews in *Eclipse Platform* and *Mylyn*.

the severity normal has the highest share of both merged and abandoned reviews. In our models, we handle severity as a categorical feature.

**Bug Priority:** It is the assigned urgency of resolving a bug. The bug priority describes the importance and order in which a bug should be fixed compared to other bugs that are also open. Developers use this field to prioritize their work to be done, where P1 is considered to be the highest and P5 the lowest in open source projects. *Eclipse* documentation [6] provides these levels: *P1 - "stop ship" defect i.e. we won't ship if not fixed, P2 - intent is to fix before shipping but we*

[6]https://wiki.eclipse.org/Bug_Reporting_FAQ

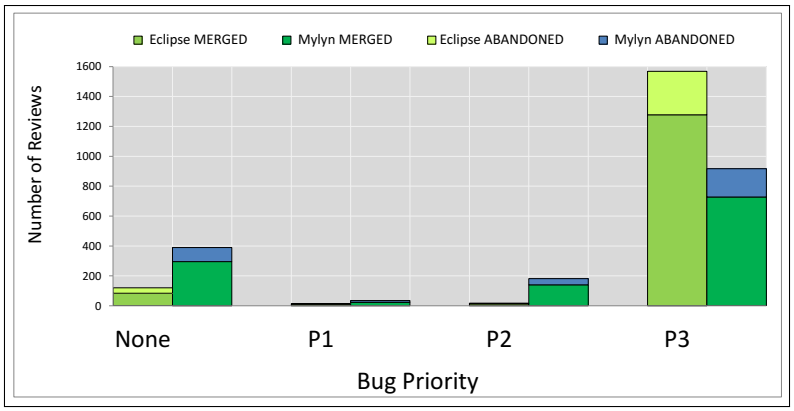


Figure 8.4: Bug priority frequency distribution for merged and abandoned reviews in *Eclipse Platform* and *Mylyn*.

*will not delay the milestone or release, P3 - nice to have, P4 - low priority, P5 - lowest priority*. Figure 8.4 shows the priority distribution between merged and abandoned reviews for both *Mylyn* and *Eclipse Platform*. *None* are related for those reviews which we could not find the related bug id. For both *Mylyn* and *Eclipse Platform*, the highest distribution in both merged and abandoned reviews belong to P3. Previous work [45] found that developers used at most three levels of priority in *Eclipse* and the use of priority/severity fields is inconsistent. They also found that the P3 has the shorter time to close than P1 and P2. We discarded priorities P4 and P5 because the frequency distribution was very low (below 1%). The percentage distribution of accepted patches for *Eclipse Platform* is 0.78%, 0.70%, and 0.81% for P1, P2, and P3 respectively. The percentage distribution of accepted patches for *Mylyn* is 0.70%, 0.77%, and 0.80% for P1, P2, and P3 respectively. There is not much difference between the acceptance rates of different bug priority levels. In our models, we handle *Priority* as a categorical feature.

**Owner Bug Assignee Match:** This feature is compounded from the assignee field from the bug tracking and the owner field from the code review repositories. It takes a binary value from evaluating whether these two fields are the same or not. That is, the assignee is the same person submitting the patch or not.

Table 8.4 shows the Percentage distribution of merged and abandoned reviews based on this feature. In *Eclipse Platform* 33% of abandoned reviews belong to owners who are also the bug

Table 8.4: Percentage distribution of merged and abandoned reviews based on owner bug assignee match feature in *Eclipse Platform* and *Mylyn*.

| | *Merged*% | | | *Abandoned*% | | |
|---|---|---|---|---|---|---|
| | Owner Bug Assignee | | | Owner Bug Assignee | | |
| | None | Match | Does not Match | None | Match | Does not Match |
| *Eclipse Platform* | 31 | 63 | 6 | 10 | 33 | 57 |
| *Mylyn* | 24 | 68 | 8 | 28 | 50 | 22 |

assignees, 57% of abandoned reviews belong to owners who are not the bug assignees, and for 10% of abandoned reviews we could not find the associated bug ids. 63% of *Eclipse Platform* merged reviews belong to owners who are also the bug assignees, 6% of merged reviews belong to owners who are not the bug assignees, and for 31% of merged reviews, we could not find the associated bug ids. In *Mylyn* 68% of merged reviews belong to owners who are the also bug assignee, 8% of merged reviews belong to owners who are not the bug assignees, and for 24% of merged reviews we could not find the associated bug ids. 50% of *Mylyn* abandoned reviews belong to owners who are also the bug assignees. 22% of abandoned reviews belong to owners who are not the bug assignees and for 28% of abandoned reviews we could not find the associated bug ids. For *Android Platform*, we do not have this feature because of the lack of traceability between patches and their bug reports. This quantitative analysis from *Eclipse Platform* and *Mylyn* shows that a substantial portion of merged reviews belongs to owners who are assigned to resolve the change request.

### 8.3.2 Patch Features

These features are related to the submitted patches in *Gerrit*, which were extracted for all three projects: *Android Platform*, *Eclipse Platform*, and *Mylyn*. All of the patch features except *Number of patch revisions* are extracted from the first submitted version of patch.

**Patch Size:** The patch size is the sum of the number of lines added to and deleted from source code files in a patch. The size of patch could have an impact on the outcome of a review. Larger patches could be tedious and require more effort to review [17]. Previous studies have found that smaller patches are more likely to be accepted [103].

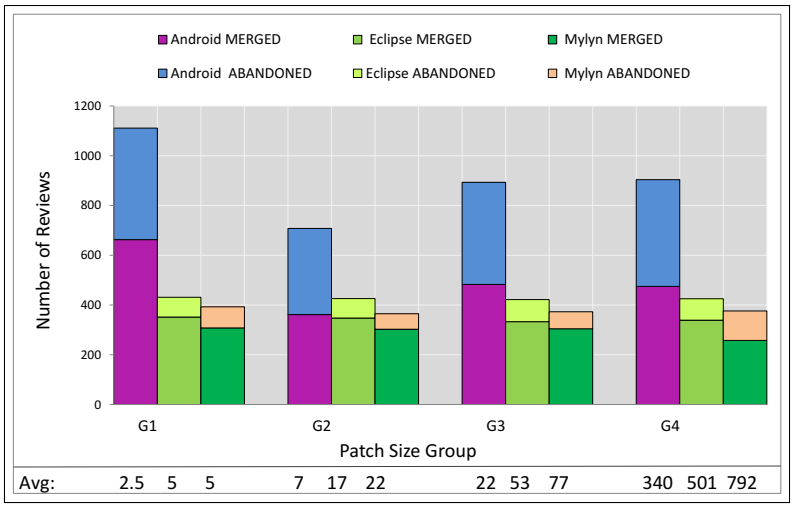


Figure 8.5: Patch size group frequency distribution for merged and abandoned reviews in *Android Platform*, *Eclipse Platform*, and *Mylyn*

Figure 8.5 shows the patch size frequency distributions for merged and abandoned reviews. Patches are divided into different groups based on the inter quartile range (IQR) of their sizes. For each project, the IQR is calculated separately. For example the patches with sizes less than equal to $Q_1$ are grouped as $G1$ and the patches with sizes greater than $Q_1$ and less than equal to $Q_2$ are grouped as $G2$ and so on. Additionally, the average value for each group is provided. The highest frequencies of both merged and abandoned patches for *Android Platform* belong to group $G_1$ with the average size of 2.5 lines. in *Eclipse Platform*, abandoned and merged patches follow the same frequency distribution between different groups. In *Mylyn*, the frequencies for merged patches across different groups are quiet similar and the highest frequency for abandoned patches belongs to the group $G4$ with the average value of 792 lines.

**File Size:** It is the number of source code files in the firs revision of submitted patch. Figure 8.6 shows the box plots related to *Android Platform*, *Eclipse Platform*, and *Mylyn*. There is not much difference in file size distributions of abandoned and merged patch for each project.

**Number of patch revisions:** It is the number of resubmitted revisions for a patch review. It shows how many times the patch owner had to revise the patch (probably to address the review

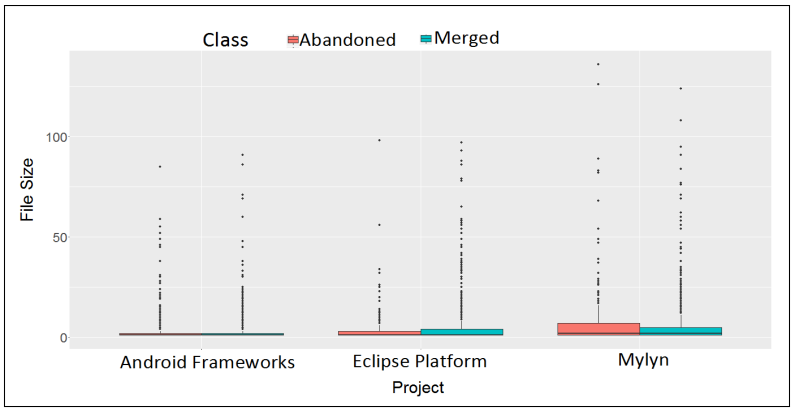


Figure 8.6: File size Box Plot for merged and abandoned patches in *Android Platform*, *Eclipse Platform*, and *Mylyn*.

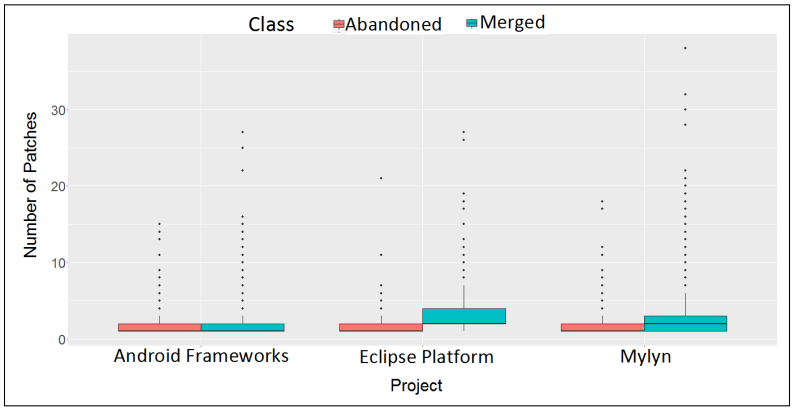


Figure 8.7: Number of patch revisions Box Plot for merged and abandoned patches in *Android Platform*, *Eclipse Platform*, and *Mylyn*.

comments) and resubmit it for another round of review. Figure 8.7 shows the box plots for all three systems. *Eclipse Platform* has the highest variation in terms of the number of patch revisions.

**Entropy Score:** This feature is based on previous work of Hellendoorn et al. [44]. The study proposed that the project's coding style is one of the important features which reviewers consider. They used language models to capture the stylistics aspects of code. The finding from this study shows that the abandoned patches are significantly less similar to the project than those merged patches. The main goal of using this feature is to quantitatively evaluate the influence of

stylistic similarity of submitted code (to existing code) on the code review outcome. Our use of this feature is in the same vein.

Hellendoorn et al.[44] states that to judge the similarity of a sequence of tokens with respect to a corpus, a language model assigns it a probability by counting the relative frequency of the sequence of tokens in the corpus. In the natural language setting, these models are used for tasks such as speech recognition, machine translation, and spelling correction. Hellendoorn et al. [44] explain the probability of a sequential language fragment $s$ of $N$ tokens $w_1....w_N$ as:

$$\begin{aligned} p(s) &= p(w_1).p(w_2|w_1)....p(w_N|w_1...w_{N-1}) \\ &= \prod_{i=1}^{N} p(w_i|w_1....w_{i-1}) \end{aligned} \tag{8.1}$$

Each $p(w_i|w_1....w_{i-1})$ can then be estimated as:

$$p(w_i|w_1....w_{i-1}) = \frac{c(w_1...w_i)}{c(w_1...w_{i-1})}. \tag{8.2}$$

Where $c$ means count. As it is explained by Hellendoorn et al. [44], as the context of a token lengthens, it becomes increasingly less likely that the sequence has been observed in the training data, which is detrimental to the performance of the model. N-gram models approach this problem by approximating the probability of each token based on a context of the last $n$ tokens only:

$$p(s) = \prod_{i=1}^{N} p(w_i|w_1....w_{i-1}) \approx \prod_{i=1}^{N} p(w_i|w_{i-n+1}....w_{i-1}) \tag{8.3}$$

Estimating the probabilities in an N-gram model is analogous to Equation 8.2, with the counts only considering the last $n$ words.

In NLP, the most commonly used metrics to measure the performance of a language models $p$ are cross-entropy ($H_p$,measured in bits/token) and perplexity ($PP_p$). Given a sentence $s$ of length $N$ and the probabilities for each word in $s : p(w_i|h)$ (where $h$ denotes a context of some length), the cross-entropy($H_p$) and perplexity ($PP_p$) will be described as below:

$$\begin{aligned} H_p(s) &= -\frac{1}{N}\sum_{i=1}^{N} \log(p(w_i|h) \\ PP_p(s) &= 2^{H_p(s)} \end{aligned} \tag{8.4}$$

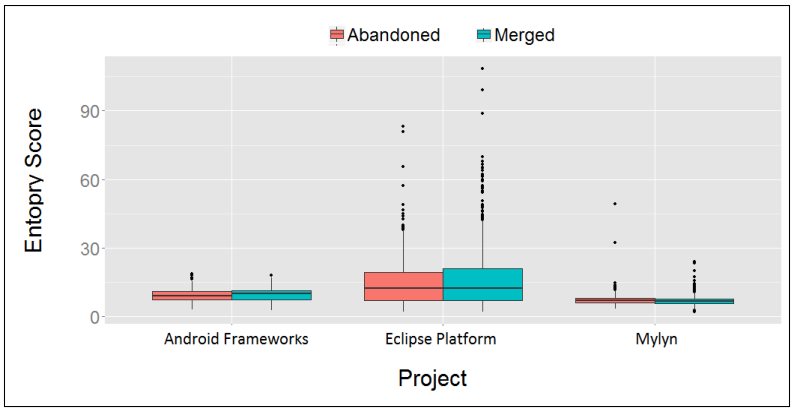


Figure 8.8: Entropy score Box Plot for merged and abandoned patches belongs to *Android Platform*, *Eclipse Platform*, and *Mylyn*.

The calculated entropy score for each patch in our conducted study is based on Equation 8.4. For each review, we consider the source code in first submitted patch version. All the added lines for each source code inside the patch is considered as a test set. For creating the training set, a copy of each relevant project is reverted to the project's state prior to the patch was submitted. All the existing source code files on that revision of project forms the train set. We train a language model on all source code files from the training set and test it on the lines of code in the patch. The output of the this step is an entropy score, reflecting the similarity of the submitted patch to the project code at the time of submission. Figure 8.8 shows the box plots for *Android Platform*, *Eclipse Platform*, and *Mylyn*. *Eclipse Platform* has the highest variation of entropy score. There is not much difference between the entropy scores of merged and abandoned patches in each project. The average entropy for *Mylyn*, *Eclipse Platform*, and *Android Platform* merged and abandoned patches are 6.62, 8.89, 12.45, 12.73, 8.86, and 9.27 respectively.

### 8.3.3 Human Features

These features will help us to investigate how much the effort and contributions of the patch owner and reviewers affect the outcome of code review process. These features are triangulated from the dimensions of patch and review sources.

**Owner's Reputation:** This feature shows the expertise of the patch owner in terms of both

local and global contributions in eventually accepted patches. Equation (8.5) consists of two parts: the first term shows the the expertise of the patch owner from the perspective of their local effort and the second term shows the efforts of other owners. By multiplying these two terms the reputation value is a normalized value.

$$\textit{Owner Reputation(p)} = \frac{\#Merged_{ow}}{\#Submitted_{ow}} \times \frac{\#Merged_{all}}{\#Submitted_{all}} \tag{8.5}$$

$ow$ is a representative for owner of the patch $p$, $\#Merged_{ow}$ is the number of submitted patches by owner $ow$ that are merged to the code repository and $\#Submitted_{ow}$ is the number of all patches submitted for review by owner $ow$. $\#Merged_{all}$ is the number of all patches that are merged to the repository and $\#Submitted_{all}$ is the number of all patches submitted for review by all owners.

This feature is calculated dynamically for each individual patch $p$ at a specific point of history (baseline at the patch's creation date). One possible case is that the status of a review has changed after the baseline, i.e., its creation date. Those reviews that their status is finalized after our baseline date should not be considered as merged or abandoned. Update date feature of review is usually the date which the status of review has been finalized to merged or abandoned. We used update date feature of review to see when the status of review has been changed and compare this date with our baseline. If update date is after the baseline that review will be considered as a review in progress and will be counted only for number of submitted reviews in formula even though if the status is merged or abandon. Figure 8.9 (a) shows the owner reputation box plot for three subject systems. As expected, the owners' reputations for merged patches are higher than those in abandoned patches for all three systems. Figure 8.9 (b) shows the percentage distribution of merged reviews based on the owner reputation. 34%, 12%, and 6 % of merged patches for *Android Platform*, *Eclipse Platform*, and *Mylyn* belong to owners with no established reputation. These cases include owners with no prior successful patch acceptance or with the first submission. They mostly occurred in the early days of adopting *Gerrit* on these projects. As such, there was no prior history of its use.

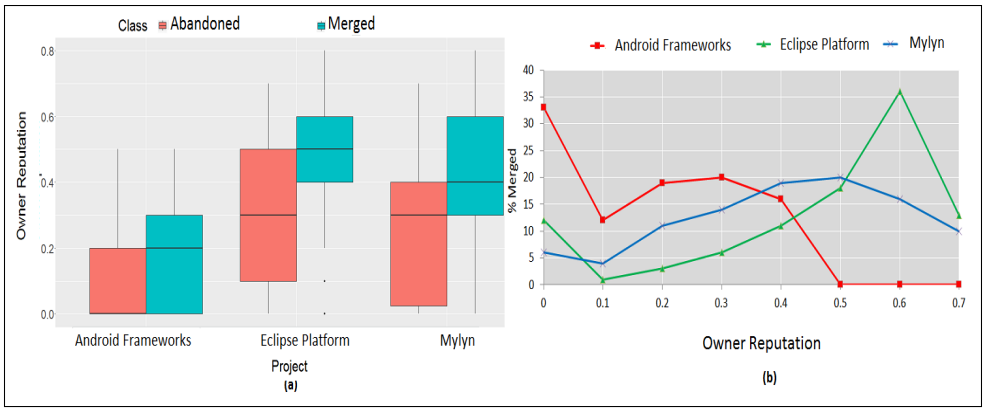


Figure 8.9: (a) Owner reputation Box Plot for merged and abandoned patches belongs to *Android Platform*, *Eclipse Platform*, and *Mylyn*. (b) Percentage distribution of merged reviews based on owner reputation

Baysal et al. [17] calculate the *Patch Writer Experience* based on the number of submitted patches for each developer and then discretizing the patch owners according to their contributions. In our method, the owner's patch acceptance rates (and not simply their submission rates) formulate their expertise.

**Number of Assigned Reviewers:** *Gerrit* enables the patch owner to specify reviewers of their choice for their submitted patch, i.e., assigned reviewers. To investigate the difference between the lists of assigned reviewers and contributing reviewers, we calculated the Jaccard similarity on our three subject systems. The average Jaccard similarity values for *Android Platform*, *Eclipse Platform*, and *Mylyn* are 0.58, 0.80, and 0.80 respectively. These values indicate that the two lists are not identical. Figure 8.10 shows the percentage distributions of the number of assigned/contributing reviewers for Abandoned and merged patches, and their clear differences. Figure 8.11 shows a box plot related to this feature. In these figures the assignee stands for the assigned reviewers and contributing stands for the contributing reviewers.

**Assigned Reviewer's Reputation:** The assigned reviewer's reputation is computed similarly to that of owner's reputation, except that the number of reviewed patches are considered instead of the number of submitted patches. Note that reviewers do not submit patches for review. For each assigned reviewer, $rw$, the reputation metric is calculated according to Equation

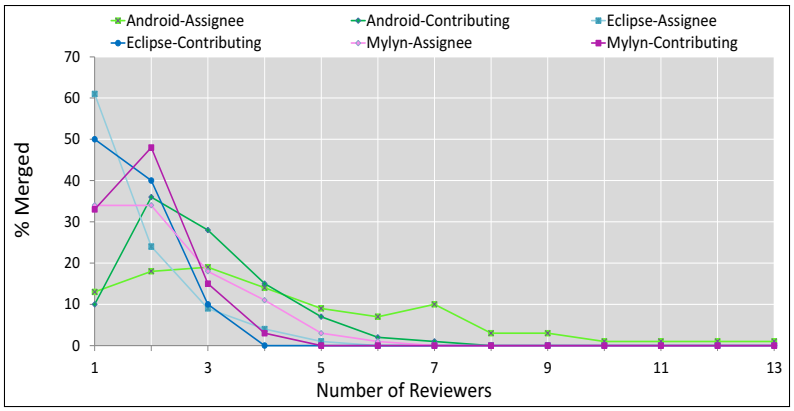


Figure 8.10: Percentage distribution of merged reviews based on number of assigned/contributing reviewers

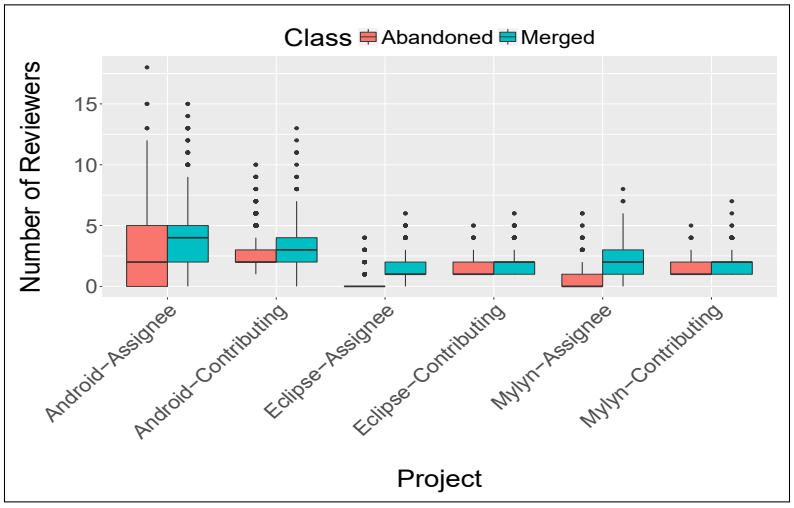


Figure 8.11: A Box Plot for the number of assigned/contributing reviewers for merged and abandoned patches

(8.6). The sum of reviewer reputations for all reviewers of the patch $p$ will be considered as the final answer.

$$\textit{Reviewer Reputation(p)} = \frac{\#Merged_{rw}}{\#Reviewed_{rw}} \times \frac{\#Merged_{all}}{\#Reviewed_{all}} \tag{8.6}$$

$\#Merged_{rw}$ is the number of all patches reviewed by reviewer $rw$ and have been merged to the repository and $\#Reviewed_{rw}$ is the number of all patches reviewed by reviewer $rw$. $\#Reviewed_{all}$ is the number of all patches reviewed by any reviewer in the review history. In our training model,

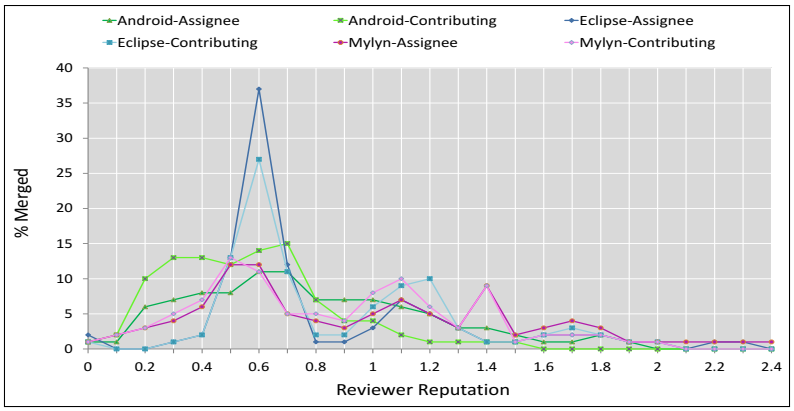


Figure 8.12: Percentage distribution of merged reviews based on assigned/contributing reviewers reputation

this feature is calculated dynamically for each individual patch $p$ at a specific point of history (baselined at the patch's creation date). Similar to owner's reputation, we used the updated date field for calculating the reputation. Figure 8.12 shows the percentage distribution of merged reviews based on assigned reviewers reputation. Figure 8.13 shows the differences of assigned reviewers reputation for merged and abandoned patches. The reviewer reputation of zero can be caused due to two reasons: 1) no established reputation of patch acceptance, and 2) no one was assigned to review the submitted patch. Baysal et al. [17] calculated the *Reviewer Activity* based on the number of previously reviewed patches and then discretized them according to their reviewing efforts using quartiles. Similar to the owner's reputation, we consider the acceptance rates of reviewers in formulating their expertise.

**Number of Contributing Reviewers:** Contributing reviewer is the one who actually participated in the review process, including writing review comments and submitting review scores. It is typically of the patch owner to be also a reviewer. In *Mylyn*, 60% of reviews are reviewed by someone other than the owner. 71% of these cases are reviewed by one reviewer and 23% of them are reviewed by two reviewers. In *Eclipse Platform*, 50% of reviews are reviewed by someone other than the owner. 70% of these cases are reviewed by one reviewer and 22% of them are reviewed by two reviewers. In *Android Platform*, 70% of reviews are reviewed by someone other than the owner. 40% of these cases are reviewed by one reviewer and 30% of them are reviewed

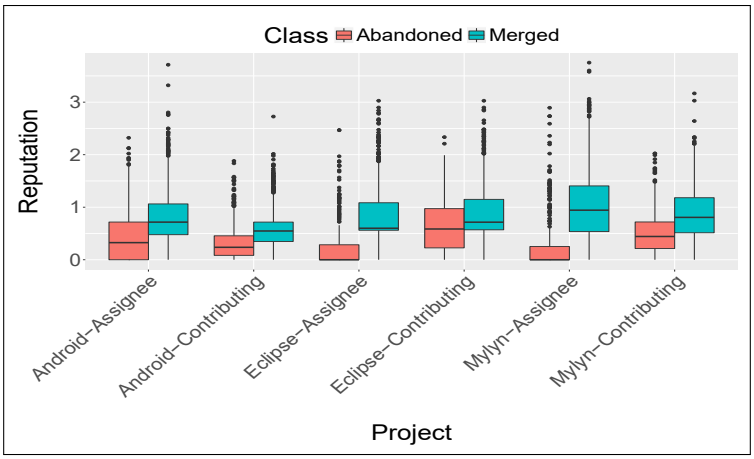


Figure 8.13: A Box Plot for the assigned/contributing reviewers reputation for merged and abandoned patches

by two reviewers, and 15% of them reviewed by three reviewers. Figure 8.10 and 8.11 show more detailed information related to this feature.

**Contributing Reviewer's Reputation:** This feature only includes the contributing reviewers. Their reputation is calculated using Equation (8.6). Although, both assigned and contributing reviewer reputations are calculated using the same formula, their values for the same reviewer can be different. The reputation is calculated dynamically, i.e., when a reviewer is associated with the patch review. For the same reviewer, their assigned reputation is calculated when the patch is submitted, whereas, their contributing reputation is calculated when they start contributing to the review. The difference in these two times could result in different values of their two reputations (e.g., due to the same reviewer contributing on other patches in the meantime, whereby affecting their reputation). Figure 8.12 and 8.13 show more detailed information related to this feature.

## 8.4 Patch Acceptance Descriptive model

The purpose of this model and study was to assess how well, if at all, the features extracted from bug and code-review repositories impact the likelihood of patches getting accepted (merged) or rejected (abandoned) in open source systems, i.e., to investigate the research question: **RQ1**: Which features **characterize** the patches that get accepted (or rejected) using a statistical model? We use logistic regression that takes the studied features (independent variables) described

in Section 8.3 and gives a **descriptive model** to characterize the impact on the binary outcome (dependent) variable, i.e., a patch is *accepted* or not. Specifically, we analyze the coefficients of features and their statistical significance to determine whether they (positively or negatively) influence or not. We use the odds ratios to infer how much do they impact the patch-acceptance likelihood.

### 8.4.1 Logistic Regression

Logistic regression is used as a classification method for analyzing a dataset in which there are one or more independent variables that determine an outcome. It trains a model from one or several variables and gives the probability of the likely outcome/label for a previously unknown test example. In the logit model the log odds of the outcome is modeled as a linear combination of the predictor variables [34]. In our study, we use logit model for binary classification, a patch is predicted as to be accepted when the model gives a probability that is greater than 0.6.

### 8.4.2 Data Extraction

In this section we briefly explain how we extract the data from *Gerrit*, and *Bugzilla*.

***Gerrit:*** To collect code review data for *Android Platform*, *Eclipse Platform*, and *Mylyn* we reverse engineered the *Grrit JSON API* and queried the *Gerrit* servers for data regarding the review for each project. *Gerrit* works by initially sending a web page skeleton and some Javascript to the browser. The Javascript then makes a number of web requests back to the *Gerrit* server and requests information about code review, which is returned in JSON format. Gerrit API provides an interface to JSON formatted review data [7]. This JSON data must still be parsed [73]. We also need to determine which fields in the displayed web page correspond to which fields within the JSON. The JSON response is fairly complex, deep, and redundant. After reading Gerrit Code Review tutorial documentation in section "*REST API/Query Changes*" we found that there is a GET command ('GET/changes') [8] which we can use for querying the changes submitted to the

[7] https://gerrit-review.googlesource.com/Documentation/rest-api.html
[8] https://gerrit-review.googlesource.com/Documentation/rest-api-changes.html#list-changes

Table 8.5: Coefficient value and Odds ratio of each feature used in the descriptive model with Logistic Regression .

| | Coefficient | | | Odds ratio | | |
|---|---|---|---|---|---|---|
| | *Mylyn* | *Eclipse Platform* | *Android Platform* | *Mylyn* | *Eclipse Platform* | *Android Platform* |
| Priority(P2) | 0.31 | -1.142 | - | 1.36 | 0.31 | - |
| Priority(P3) | 0.91 | 0.32 | - | 2.48 | 1.37 | - |
| Severity(critical) | 0.69 | -12.9 | - | 1.98 | $\approx 0$ | - |
| Severity(enhancement) | 0.63 | -13.68 | - | 1.87 | $\approx 0$ | - |
| Severity(major) | 1.62 | -13.22 | - | 5.07 | $\approx 0$ | - |
| Severity(minor) | 0.54 | -13.24 | - | 1.71 | $\approx 0$ | - |
| Severity(normal) | 0.89 | -13.25 | - | 2.44 | $\approx 0$ | - |
| Severity(trivial) | 1.65 | -12.71 | - | 5.20 | $\approx 0$ | - |
| Owner Bug Assignee Match(1) | 1.76 | 0.68 | - | 5.79 | 1.96 | - |
| Patch Size | -0.0008 | -0.0001 | -0.0005 | 1 | 0.99 | 0.99 |
| Number of Patch Revisions | 0.23 | 0.523 | 0.22 | 1.26 | 1.68 | 1.25 |
| File Size | -0.009 | -0.008 | -0.01 | 0.99 | 0.99 | 0.98 |
| Entropy Score | -0.007 | -0.011 | 0.04 | 0.99 | 0.98 | 1.03 |
| Owner's Reputation | 2.14 | 1.24 | 2.58 | 8.55 | 3.46 | 13.24 |
| Number of Assigned Reviewers | 2.74 | 1.76 | 0.26 | 15.55 | 5.81 | 1.29 |
| Number of Contributing Reviewers | -3.01 | -1.45 | -0.63 | 0.049 | 0.23 | 0.52 |
| Assigned Reviewer's Reputation | 1.47 | 1.88 | 0.05 | 4.37 | 6.54 | 1.05 |
| Contributing Reviewer's Reputation | 1.82 | 0.01 | 3.67 | 6.22 | 1.01 | 39.28 |

*Gerrit*. A query string must be provided by the $q$ parameter. The $n$ parameter can be used to limit the returned results.

Each review in *Gerrit* has a unique *number* which can be used to query the information related to that review. Not all of the data fields in the JSON response query meet our needs. We filter out the interesting fields which are the features we used for our predictor.

***Bugzilla:*** In out study, the bug reports for *Mylyn* and *Eclipse Platform* are extracted from the online bug tracking system *Bugzilla*[9]. For *Android Platform* we could not find the related bug Id from reviews.

[9]https://bugs.eclipse.org/bugs/

### 8.4.3 Results

For each feature in our descriptive model three different values; coefficient value, the odd ratio, and magnitude of p-value; have been reported in Tables 8.5, 8.6 respectively. Not all the features we considered are numeric. Features: *Component*, *Priority*, *Severity*, and *Owner bug assignee match* are categorical features. In logistic regression model these features are treated as factors. In logistic regression, one value for a factor is treated as a baseline and other values assume their coefficients relatively. Coefficient values for each value of these features are reported in table 8.5. The coefficients returned from a logistic regression model are log-odds ratios. They tell us how the log-odds of a ”success” change with a one-unit change in the independent variable. Increasing the log-odds of a success means increasing the probability, and vice-versa, i.e., decreasing the log-odds of a success means decreasing the probability. Therefore, the sign of the log-odds ratio indicates the direction of its relationship: a + sign means a positive relationship between the specific feature and the likelihood of the patch getting accepted, and a - sign means a negative relationship.

Next, we wanted to see how much each feature affects the outcome likelihood of a patch getting accepted or not. Table 8.5 shows the odds ratio for each feature used in the descriptive model. The odds ratio of accepting a patch is defined as the probability of accepting to the probability of not accepting. For a feature, the odds ratio is calculated from the exponent of its coefficient. The odds ratio gives the change in outcome for a one unit increase in the feature. For example, the odds ratio of 8.55 for the numeric feature *Owner's Reputation* in *Mylyn* states that every one unit change in it, the odds of the submitted patch getting accepted (over remaining unaccepted) increases by 755%. If the value of odds ratio is less than 1 means the odds of the submitted patch getting accepted decreases. For factors (ordinal features), the interpretation is with respect to the base value. That is, the odds ratio gives the change in outcome for a change to a specific value of a feature from its baseline value.

Finally, we wanted to see the impact of each feature on the outcome value of a patch getting accepted or not. Table 8.6 shows the magnitude of p-values from statistical testing for each feature. Dot represents the p-value is between 0.05 and 0.1, one star represents the p-value is between 0.01

Table 8.6: P-values from statistical significance test for features used in the descriptive model with Logistic Regression.

| | *Mylyn* | *Eclipse Platform* | *Android Platform* |
|---|---|---|---|
| Component | | * | - |
| Priority | | | - |
| Severity | | | - |
| Owner Bug Assignee Match | *** | *** | - |
| Patch Size | | . | ** |
| Number of Patch Revisions | * | *** | *** |
| File Size | | | |
| Entropy Score | | | * |
| Owner's Reputation | * | . | *** |
| Number of Assigned Reviewers | *** | *** | *** |
| Number of Contributing Reviewers | *** | *** | *** |
| Assigned Reviewer's Reputation | | ** | |
| Contributing Reviewer's Reputation | | | *** |

and 0.05, two stars means the p-value is between 0.001 and 0.01, three stars indicate that the p-value is less than 0.001, and a blank entry means that the p-value is greater than 0.05. For factors, the p-value was determined collectively for all the values (using Wald test in R). A feature with at least one star is considered statistically significant. It is clear from Table 8.6 that each system has a different set of significant features. Therefore, we needed to assess how good was the model in terms of describing the outcome for each system. We analyzed the null and deviance residuals with the chi-square test. In each system, the p-value was very close to zero, which indicates that the model as a whole is statistically better than a null model (i.e., the empty model with no features). In summary, logistic regression on the considered features provided a good fit model for patches that get accepted or not.

Now, we explain the coefficient value, the odd ratio, and magnitude of p-value for each feature in detail.

With regard to the feature *Component*, *Framework* for *Mylyn* and *UI* for *Eclipse Platform* are considered as the baseline in our descriptive model. Considering Table 8.2, these two components have the highest frequency distributions of reviews in our dataset; therefore, we consider them as the baseline.

The highest coefficient value of the feature *Component* for *Mylyn* is 16.16, which is for *build*. Although the *build* component does not have the highest frequency distribution of reviews, all the patches related to this component are finally merged to the repository. The highest coefficient value of the feature *Component* for *Eclipse Platform* is 15.67, which is for *Ant*. Although the *Ant* component does not have the highest frequency distribution of reviews, all the patches related to this component are finally merged to the repository. For both *Mylyn* and *Eclipse Platform*, there are components which have either negative or positive coefficients.

The feature *Component* is not significant for *Mylyn* but it is significant for *Eclipse Platform*. This difference can be attributed to the nature of these two systems. *Mylyn* is a project from *Eclipse Foundation* but *Eclipse Platform* is a product. That is, *Mylyn* is at a higher level of organization than *Eclipse Platform*. Referring to Table 8.2, the frequency distributions of reviews in components of *Mylyn* and *Eclipse Platform* are different. For example , 71% of *Eclipse Platform* patches are in the component *UI*. Baysal et al. [19] also found that the feature *Component* is not significant in other open source projects *WebKit* and *Blink*.

With regard to the feature *Priority*, we consider P1, which has the highest priority, as the baseline for both *Mylyn* and *Eclipse Platform*. We want to assess if changing the priority level to P2 or P3, changes the chance of acceptance or not. The *Priority* coefficient values of *Mylyn* for both P2 and P3 are positives. Although these values are not particularly high, they still show the increase in the probability of the patches getting accepted. As we mentioned in Section 8.3.1 the patch acceptance rates for *Mylyn* are 0.70%, 0.77%, and 0.80% for P1, P2, and P3 respectively. These acceptance rates explain the positive coefficient values for *Mylyn*. In *Eclipse Platform* the

coefficient values for P2 is negative and P3 is positive. The patch acceptance rates for *Eclipse Platform* are 0.78%, 0.70%, and 0.81% for P1, P2, and P3 respectively. These acceptance rates explain the positive and negative coefficient values for *Eclipse Platform*. The feature *Priority* is not significant for *Mylyn* or *Eclipse Platform*. Baysal et al. [19] found that Feature *Priority* is significant for *WebKit* but not available for *Blink*. This difference can be attributed to the nature of *WebKit* and *Eclipse Foundation* Projects.

With regard to the feature *Severity*, we consider blocker, which has the highest severity, as the baseline for both *Mylyn* and *Eclipse Platform*. We want to assess if changing the severity level to less sever level, the chance of acceptance changes or not. Blocker has the patch acceptance rate of 100% for *Eclipse Platform*, but this is not the case for other severity levels. Therefore, the reported coefficient values for *Eclipse Platform*'s severity in Table 8.5 are all negatives. In case of *Mylyn*, Blocker has the lowest patch acceptance rate. Therefore, the reported coefficient values for other *Mylyn*'s severity levels in Table 8.5 are all positive. We manually checked the patches with severity blocker for *Mylyn* that were abandoned to see what is the reason for this low acceptance rate. All of them were abandoned because they are moved and merged to other reviews with the same severity level (blocker). For example review id #28749 related to bug id #436398 with severity level of blocker is abandoned and moved to review id #27759 and merged to repository. This is the similar case for all of *Mylyn* abandoned patches related to bugs with severity of blocker. Therefore, the actual acceptance rate for *Mylyn*'s blocker level reviews is similar to *Eclipse Platform* and is 100%. Figure 8.3 shows the frequency distribution of merged and abandoned reviews for *Mylyn* and *Eclipse Platform* regarding different severity values. The highest patch acceptance rate for *Mylyn* belongs to blocker (indirectly) and trivial, and for *Eclipse Platform* it belongs to blocker. Regarding Table 8.6, the feature *Severity* is neither significant for *Mylyn* nor *Eclipse Platform*. The odds ratios from Table 8.5 for *Eclipse Platform* are all close to zero, this means changing the severity level from blocker to any other severity level, dramatically decreases the odds of patch getting accepted. This feature is not investigated in [19].

With regard to the feature *Owner Bug Assignee Match*, we consider the value 0 (i.e., the

assignee is not the same person submitting the patch). The coefficient values related to this feature, shown in Table 8.5, are positive for both *Mylyn* and *Eclipse Platform*. This fact implies that the chances of a patch getting accepted is higher when the assignee is the same person as the patch submitter. Table 8.4 shows the percentage distribution of merged and abandoned reviews for *Mylyn* and *Eclipse Platform* regarding this feature. This distribution explains the positive coefficient values. Regarding Table 8.6, the feature *Owner Bug Assignee Match* is significant for both *Mylyn* and *Eclipse Platform*. [19] did not investigate this feature.

With regard to the feature *Patch Size*, the coefficient values related to this feature are negative for all three systems. Based on Figure 8.5, the highest acceptance rate for *Mylyn* is 78% and belongs to Group $G2$ with the average size of 7 lines. The highest acceptance rates for *Eclipse Platform* and *Android Platform* are 81% and 60%, which belong to Group $G1$ with the average sizes of 5 and 2.5 lines respectively. Increasing the size of a patch associates with decreasing the chance of its acceptance. This finding is consistent with finding from previous studies that smaller patches are more likely to be accepted [103]. Similarly, Tao et al. [95] found that, patch size too large is one of the rejection reason for *Eclipse* project. In Table 8.6, *Patch Size* is significant only for *Android Platform*. The effect of this feature on the positivity of the review outcome is not available in [19].

The coefficient values for the feature *Number of Patch Revisions* are positive for all three systems and this feature is significant for all three systems. Considering Figure8.7, *Mylyn* and *Eclipse Platform* merged patches have gone through more round of revisions in comparison with abandoned patches. This explains the positive coefficient values. It is expected that successful patches should go through less round of revisions, but the fact is when reviewer are quite sure that the submitted patch has a potential to merge to code repository, they try to be more precise with the code change and request the author of code change to submit the best quality of code. This will cause the author to rework the patch and submit several revision of patches. This is not the case when reviewers don't see the merge potential in code change. In general *Android Platform*'s patches have gone through fewer round of revisions than those in *Eclipse Platform* and *Mylyn*.

*Eclipse Platform* patches have the highest round of revisions. The effect of this feature on the positivity of the review outcome is not available in [19].

The coefficient values for the feature *File Size* are negative for all three systems and this feature is not significant for any of the three systems. Considering Figure 8.6, patches submitted for *Android Platform* have lowest number of files and are smallest in comparison with *Mylyn* and *Eclipse Platform*. This feature is not investigated in [19].

Hellendoorn et al. [44] found that the average entropy of rejected patches is significantly higher than of accepted patches. In our study, there is not much difference between average entropies of merged and abandoned patches in all three systems. But the average entropy of merged patches is slightly higher than abandoned patches. *Mylyn* patches have the lowest entropy score, which means the patches written in *Mylyn* are more similar to its codebase than in the two other systems. The coefficient values reported for *Entropy Score* feature are negative for *Mylyn* and *Eclipse Platform* this means increasing the Entropy score will decrease the chance of acceptance. This is consistent with findings in [44]. Coefficient values is positive for *Android Platform* but the odds ratio is 1.03 which means increasing one unit in Entropy score will increase the chance of patch acceptance only by 3%. Literally, it means increase in Entropy score will not change the odds of patch getting accepted. This feature is only significant for *Android Platform*. This feature is not investigated in [19].

Perhaps expected, the feature *Owner's Reputation*, the coefficient values are positive (see Table 8.5). Similarly, Figure 8.9 (a) shows that owners of merged patches for all three systems have higher reputations than those of abandoned patches. The owners for the *Android Platform* patches have lower reputations than those of *Mylyn* and *Eclipse Platform*. This feature is statistically significant for *Mylyn* and *Android Platform*. For *Eclipse Platform* the reported p-value is 0.059, which is close to 0.05. The odds ratio values reported for this feature is significant for all three systems. For example, one unit increase in the owner's reputation (e.g., 0.1) will cause 1224% increase in odds of the patch getting accepted for *Android Platform*. Our finding is consistent with [19].

The coefficient values for feature *Number of Assigned Reviewers* are positive for all three

systems. It means increasing the number of assigned reviewers will increase the chance of patch getting accepted. Figure 8.11 shows that the usually merged patches have more number of assigned reviewers in comparison with abandoned patches. In case of *Eclipse Platform* the big portion of abandoned patches do not have assigned reviewers. Some of these cases belongs to those developers who are the main developers of the project and have commit permission hence they do not need other reviewers to review their code. They uploaded their patch to the *Gerrit* and then later they abandoned their patch for a specific reason. *Android Platform* has the highest variation in terms of number of assigned reviewers. Figures 8.10 shows the majority of merged patches for *Eclipse Platform* have one assigned reviewer but this value for *Mylyn* is one and two and for *Android Platform* is two and three. This feature is statistically significant for all three systems. This feature is not investigated in [19].

Coefficient values for feature *Number of Contributing Reviewers* are all negatives. Which means increasing the number of reviewers in review discussion will have negative impact on the outcome of review. Rigby et al. [84] address this as ”bike shed painting” or Parkinson’s law of triviality, which means too many reviewers contribute in review discussion and lead to unimportant changes being discussed to a deadlock (I don’t know if we can relate this to the reason that patch will be abandoned?). Figure 8.11 shows that for *Mylyn* and *Eclipse Platform* the number of contributing reviewers for merged and abandoned patches are quite similar. This Feature is statistically significant for all three systems. This feature is not investigated in [19]. There is a negative correlation between number of assigned reviewers and number of contributing reviewers. Increasing the number of assigned reviewers will increase the chance of patch get accepted and increasing the number of contributing reviewers will decrease the chance of patch get accepted. We compared the number of assigned reviewers with the number of participated reviewers for each review for all three systems. 40%, 87%, and 83% of abandoned reviews for *Android Platform*, *Eclipse Platform*, and *Mylyn* the number of participated reviewers is higher than number of assigned reviewers while these ratios are 4%, 20%, and 5% for merged reviews for *Android Platform*, *Eclipse Platform*, and *Mylyn* respectively.

Feature *Assigned Reviewer's Reputation* considers the reputation for assigned reviewers. For all three systems the coefficient values are positive. Similarly Figure 8.13 shows that assigned reviewer's of merged patches for all three systems have higher reputation in comparison with abandoned patches. *Android Platform* has less variation between the reputation of merged and abandoned patches in comparison to *Mylyn* and *Eclipse Platform* and this explains why *Android Platform* has the lowest coefficient value (e.g., 0.05) and lowest odds ratio between three systems (e.g., 1.05). Figure 8.12 shows the percentage distribution of merged patches based on assigned reviewers reputation. This is interesting that for all three systems the highest percentage distribution belongs to reputation value of 0.6. This feature is statistically significant only for *Eclipse Platform*. Baysal et al. [19] found that reviewer activity is not statistically significant for *WebKit* and *Blink*.

Coefficient values related to the feature *Contributing Reviewer's Reputation* are positive for all three systems. We would like to compare this feature with *Assigned Reviewer's Reputation*. It's clear from Figure 8.13 that contributing reviewer's reputation between merged and abandoned patch has less variance in comparison with assigned reviewer's reputation between merged and abandoned patch for all three systems. Figure 8.12 shows that for *Mylyn* and *Eclipse Platform*, contributing reviewer's reputation and assigned reviewer's reputation follow the same pattern but for *Android Platform* the pattern is quiet different. As we mentioned earlier 8.3.3 the the average jaccard similarity values between two list of contributing reviewer's and assigned reviewer's for *Android Platform*, *Eclipse Platform*, and *Mylyn* are 0.58, 0.80, and 0.80 respectively. *Android Platform* has the less similarity, this will explains the difference for *Android Platform*. This feature is statistically significant only for *Android Platform*. Baysal et al. [19] found that reviewer activity is not statistically significant for *WebKit* and *Blink*.

It is clear from Table 8.6 that each system has a different set of significant features. Therefore, we needed to assess how good was the model in terms of describing the outcome for each system. We analyzed the null and deviance residuals with the chi-square test. In each system, the p-value was very close to zero, which indicates that the model as a whole is statistically significant

better than a null model (i.e., empty model with no features). In summary, logistic regression on the considered features provided a good fit model for bugs that get fixed or not.

## 8.5 Patch Acceptance Predictive Model

The purpose of this model and study was to assess how well does the prediction model perform on the subject software systems, i.e., the research question **RQ2**: How well whether a patch will be accepted or not can be **predicted** using a statistical model? Additionally, we wanted to compare the results of two different classification techniques: Logistic Regression (LR) and Support Vector Machine (SVM). The choice of SVM was driven by the fact that it was used in other related empirical studies [57]

The problem of predicting whether a given patch will be accepted or rejected can be considered as an instance of classification. More specifically, it is an instance of binary classification whose prediction outcome is *accept* or *reject*.

There are previous studies [17], [50] that attempted to study the impact of different features on the outcome of the code review process. We additionally created a predictive model that compares different groups of features on the patch acceptance outcome. We formulated and evaluated our predictive model on different combinations of bug, patch and human groups of features. Next, we discuss in description of the prediction model and the specific features used.

### 8.5.1 Support Vector Machines

Support vector machines (SVM) is a supervised learning model for two-group classification problems. Input vectors are non-linearly mapped to a very high dimension feature space. In this feature space a linear decision surface is constructed. Special properties of the decision surface ensures high generalization ability of the learning machine. A Support Vector Machine (SVM) performs classification by finding the hyperplane that maximizes the margin between the two classes. The vectors that define the hyperplane are the support vectors. New examples are then mapped into that same space and predicted to belong to a category based on which side of the margin they fall on [33].

### 8.5.2 Predictive Featured Used

Not all the features we used in the descriptive model are available at the patch submission time. Hence, we considered only those features that are available at the patch submission time. Specifically, the features *Component*, *Priority*, *Severity*, *Owner Bug Assignee Match*, *Patch Size*, *File Size*, *Entropy Score*, *Owner's Reputation*, *Number of Assigned Reviewers*, and *Assigned Reviewer's Reputation* were used in the predictive model. Note that, *Component*, *Priority*, *Severity*, and *Owner Bug Assignee Match* are not applicable for *Android Platform*.

### 8.5.3 Training and Test Sets

To conduct our evaluation, we divided the dataset for each system into training and test sets. For creating the training and test sets, we first sorted the data set based on the review creation date. We can not use the information from the future reviews in the training set to predict the past reviews. The dataset was split into 10 different folds and conducted 10 runs. In each run, 2/3$^{rd}$ and 1/3$^{rd}$of the data were used as training and test sets, and one fold was dropped for the next run. That is, the first run had 10 folds and the last run had 1 fold.

### 8.5.4 Evaluation Metrics

Our patch classifier predictive model should correctly predict both accepted and rejected patches. We need to calculate the evaluation metrics separately for both accepted and rejected predictions. Hence, we defined four possible conclusions from the patch prediction model.

$Patch_{aa}$: Number of patches that predicted accepted and were actually accepted.

$Patch_{au}$: Number of patches that predicted accepted and were actually unaccepted.

$Patch_{uu}$: Number of patches that predicted unaccepted and were actually unaccepted.

$Patch_{ua}$: Number of patches that predicted unaccepted and were actually accepted.

We used well defined metrics precision and recall for two different outputs of classifier. The accuracy metrics evaluates the overall performance of the models.

$Accuracy$ is the overall ability to correctly predict both accepted and unaccepted patches over the total predictions 8.7a. It accounts for both type I (false positives) and type II errors (false

negatives).

$$Accuracy = \frac{Patch_{aa} + Patch_{uu}}{Patch_{aa} + Patch_{au} + Patch_{uu} + Patch_{ua}} \tag{8.7a}$$

$$Precision_{ap} = \frac{Patch_{aa}}{Patch_{aa} + Patch_{au}} \tag{8.7b}$$

$$Recall_{ap} = \frac{Patch_{aa}}{Patch_{aa} + Patch_{ua}} \tag{8.7c}$$

$$Precision_{uap} = \frac{Patch_{uu}}{Patch_{uu} + Patch_{ua}} \tag{8.7d}$$

$$Recall_{uap} = \frac{Patch_{uu}}{Patch_{uu} + Patch_{au}} \tag{8.7e}$$

In our context, precision and recall allow us to evaluate the model from the prospectives of accepted patches, similarly we need to evaluate the model from the prospectives of unaccepted patches. Hence we report the precision and recall value for both accepted and unaccepted patches.

$Precision_{ap}$ and $Recall_{ap}$ are the precision and recall values for accepted patches 8.7b and 8.7c.

$Precision_{uap}$ and $Recall_{uap}$ are the precision and recall values for unaccepted patches 8.7d and 8.7e.

Comparing these four metrics give us the capability to compare the performance of predictive model for predicting both accepted and unaccepted patches in terms of both precision and recall. Accuracy metric evaluate the overall performance of the model.

## 8.6 Predictive Results

We used either a Logistic Regression(LR) model or a Support Vector Machine(SVM) model with the training sets, and use the test sets to assess the performance of the models afterwards. Each technique is ran for four different groups of features. Average, minimum and maximum values of $Accuracy$, $Precision_{ap}$, $Recall_{ap}$, $Precision_{uap}$, and $Recall_{uap}$ of 10 different test sets are reported in tables 8.7 and 8.8.

In the case of logistic regression, we assessed if the model formed from each of the training sets in each groups of features was a good fit. We employed chi-square statistical test and the p-values for each model was much less than 0.05, which implied the model was a good fit. The

Table 8.7: Performance of Logistic Regression (LR).

| Logistic Regression | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Bug+Patch+Human | | | Patch+Human | | | Bug+Patch | | | Bug+Human | | |
| | | Maximum | Average | Minimum | Maximum | Average | Minimum | Maximum | Average | Minimum | Maximum | Average | Minimum |
| $Accuracy$ | *Mylyn* | 0.91 | 0.84 | 0.76 | 0.92 | 0.85 | 0.76 | 0.84 | 0.74 | 0.43 | 0.91 | 0.87 | 0.77 |
| | *Eclipse Platform* | 0.97 | 0.91 | 0.84 | 0.96 | 0.93 | 0.84 | 0.88 | 0.80 | 0.72 | 0.98 | 0.91 | 0.75 |
| | *Android Platform* | - | - | - | 0.77 | 0.65 | 0.47 | 0.65 | 0.46 | 0.23 | 0.76 | 0.65 | 0.46 |
| $Precision_{ap}$ | *Mylyn* | 0.92 | 0.86 | 0.80 | 0.91 | 0.86 | 0.80 | 0.86 | 0.79 | 0.54 | 0.92 | 0.88 | 0.80 |
| | *Eclipse Platform* | 0.97 | 0.92 | 0.83 | 0.97 | 0.93 | 0.83 | 0.89 | 0.80 | 0.72 | 0.97 | 0.92 | 0.83 |
| | *Android Platform* | - | - | - | 0.97 | 0.66 | 0.45 | 1 | 0.43 | 0.31 | 0.87 | 0.66 | 0.44 |
| $Recall_{ap}$ | *Mylyn* | 0.99 | 0.94 | 0.81 | 1 | 0.96 | 0.72 | 0.98 | 0.88 | 0.47 | 0.99 | 0.96 | 0.91 |
| | *Eclipse Platform* | 1 | 0.97 | 0.89 | 1 | 0.99 | 0.97 | 1 | 0.99 | 0.96 | 1 | 0.96 | 0.76 |
| | *Android Platform* | - | - | - | 0.81 | 0.65 | 0.50 | 0.63 | 0.08 | 0 | 0.8 | 0.65 | 0.5 |
| $Precision_{uap}$ | *Mylyn* | 0.90 | 0.71 | 0.27 | 0.98 | 0.86 | 0.67 | 0.59 | 0.40 | 0.1 | 0.95 | 0.76 | 0.43 |
| | *Eclipse Platform* | 1 | 0.86 | 0.48 | 1 | 0.92 | 0.86 | 0.5 | 0.16 | 0 | 1 | 0.88 | 0.31 |
| | *Android Platform* | - | - | - | 0.87 | 0.62 | 0.48 | 0.70 | 0.48 | 0.23 | 0.86 | 0.62 | 0.48 |
| $Recall_{uap}$ | *Mylyn* | 0.81 | 0.45 | 0.15 | 0.81 | 0.42 | 0.14 | 0.43 | 0.19 | 0.05 | 0.81 | 0.51 | 0.17 |
| | *Eclipse Platform* | 0.80 | 0.67 | 0.47 | 0.82 | 0.72 | 0.47 | 0.08 | 0.01 | 0 | 0.80 | 0.69 | 0.47 |
| | *Android Platform* | - | - | - | 0.79 | 0.64 | 0.41 | 1 | 0.92 | 0.37 | 0.79 | 0.65 | 0.40 |

results from each predictor model (LR,SVM) based one different groups of feature, trained by 10 different training sets for each system compared with each others with the Kruskal-Wallis tests and all p-values were are less than 0.05 which means there is statistical significant differences between the results.

For evaluating the classifier we compare the results from the predictive models with a dummy predictor. Dummy predictor randomly predicts the label for patch. Based on the distribution of accepted and unaccepted patch for each system, a related weight is considered for the randomize operation in dummy predictor. For example in case of *Eclipse Platform*. distribution of accepted patch is 81% and distribution of unaccepted patch is 19%. Hence the dummy predictor picks 81% of test set as accepted and 19% of test set as unaccepted. We run the dummy predictor 20 times for each test set and then calculate the average, maximum, and minimum.

We also compare our result with a Zero model. This model has been used in previous studies [49]. Zero model always predicts ”accepted”. For example, if there are 80% accepted patches, then a zero model would have accuracy and recall of 80%. Table 8.9 shows the result for dummy predictor and Zero model.

Table 8.8: Performance of Support Vector Machine (SVM).

| | | Support Vector Machine | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Bug+Patch+Human | | | Patch+Human | | | Bug+Patch | | | Bug+Human | | |
| | | Maximum | Average | Minimum | Maximum | Average | Minimum | Maximum | Average | Minimum | Maximum | Average | Minimum |
| $Accuracy$ | *Mylyn* | 0.95 | 0.92 | 0.77 | 0.95 | 0.91 | 0.78 | 0.84 | 0.75 | 0.44 | 0.95 | 0.93 | 0.82 |
| | *Eclipse Platform* | 0.98 | 0.93 | 0.85 | 0.98 | 0.93 | 0.85 | 0.89 | 0.80 | 0.72 | 0.98 | 0.94 | 0.85 |
| | *Android Platform* | - | - | - | 0.79 | 0.68 | 0.56 | 0.64 | 0.47 | 0.28 | 0.79 | 0.68 | 0.54 |
| $Precision_{ap}$ | *Mylyn* | 0.95 | 0.92 | 0.83 | 0.95 | 0.91 | 0.81 | 0.85 | 0.78 | 0.52 | 0.95 | 0.92 | 0.85 |
| | *Eclipse Platform* | 0.97 | 0.93 | 0.83 | 0.97 | 0.93 | 0.83 | 0.89 | 0.80 | 0.72 | 0.97 | 0.94 | 0.83 |
| | *Android Platform* | - | - | - | 0.90 | 0.67 | 0.50 | 0.86 | 0.52 | 0 | 0.88 | 0.68 | 0.50 |
| $Recall_{ap}$ | *Mylyn* | 1 | 0.96 | 0.75 | 1 | 0.98 | 0.81 | 1 | 0.90 | 0.52 | 1 | 0.98 | 0.85 |
| | *Eclipse Platform* | 1 | 0.99 | 0.97 | 1 | 0.99 | 0.97 | 1 | 1 | 1 | 1 | 0.99 | 0.98 |
| | *Android Platform* | - | - | - | 0.87 | 0.76 | 0.61 | 0.48 | 0.11 | 0 | 0.89 | 0.77 | 0.61 |
| $Precision_{uap}$ | *Mylyn* | 0.82 | 0.72 | 0.57 | 0.81 | 0.65 | 0.43 | 0.39 | 0.11 | 0 | 0.81 | 0.72 | 0.57 |
| | *Eclipse Platform* | 0.82 | 0.72 | 0.47 | 0.81 | 0.71 | 0.47 | 0 | 0 | 0 | 0.83 | 0.73 | 0.47 |
| | *Android Platform* | - | - | - | 0.72 | 0.61 | 0.35 | 0.99 | 0.91 | 0.56 | 0.81 | 0.62 | 0.29 |
| $Recall_{uap}$ | *Mylyn* | 1 | 0.91 | 0.70 | 1 | 0.95 | 0.73 | 0.39 | 0.17 | 0.29 | 1 | 0.96 | 0.79 |
| | *Eclipse Platform* | 1 | 0.94 | 0.86 | 1 | 0.96 | 0.88 | 0 | 0 | 0 | 1 | 0.96 | 0.87 |
| | *Android Platform* | - | - | - | 0.92 | 0.68 | 0.49 | 0.71 | 0.48 | 0.24 | 0.93 | 0.69 | 0.50 |

Table 8.9: Performance of Dummy Predictor and Zero model

| | | Dummy Predictor | | | Zero Model | | |
|---|---|---|---|---|---|---|---|
| | | Maximum | Minimum | Average | Maximum | Minimum | Average |
| Accuracy | *Mylyn* | 0.46 | 0.38 | 0.42 | 0.85 | 0.59 | 0.78 |
| | *Eclipse Platform* | 0.58 | 0.50 | 0.54 | 0.89 | 0.72 | 0.81 |
| | *Android Platform* | 0.52 | 0.43 | 0.47 | 0.77 | 0.30 | 0.55 |
| Recall | *Mylyn* | 0.82 | 0.77 | 0.78 | 1 | 1 | 1 |
| | *Eclipse Platform* | 0.81 | 0.79 | 0.80 | 1 | 1 | 1 |
| | *Android Platform* | 0.56 | 0.49 | 0.50 | 1 | 1 | 1 |
| Precision | *Mylyn* | 0.42 | 0.28 | 0.36 | 0.85 | 0.59 | 0.78 |
| | *Eclipse Platform* | 0.62 | 0.51 | 0.57 | 0.89 | 0.72 | 0.81 |
| | *Android Platform* | 0.70 | 0.30 | 0.50 | 0.77 | 0.30 | 0.55 |
| Specificity | *Mylyn* | 0.23 | 0.21 | 0.22 | 0 | 0 | 0 |
| | *Eclipse Platform* | 0.21 | 0.18 | 0.20 | 0 | 0 | 0 |
| | *Android Platform* | 0.44 | 0.42 | 0.44 | 0 | 0 | 0 |
| NPV | *Mylyn* | 0.75 | 0.57 | 0.65 | 0 | 0 | 0 |
| | *Eclipse Platform* | 0.50 | 0.35 | 0.43 | 0 | 0 | 0 |
| | *Android Platform* | 0.70 | 0.22 | 0.43 | 0 | 0 | 0 |

Comparing the results of LR and SVM from Tables 8.7 and 8.8, the average results achieved by performing SVM are better than LR. Among the four different groups of SVM results, the results of Bug+Patch+Human, Patch+Human, and Bug+Human groups are pretty much similar. These three groups could capture the average $Accuracy$ of 92%, 93%, and 68% for *Mylyn*, *Eclipse Platform*, and *Android Platform* respectively. The results by these three groups could capture the average $Accuracy$ gain of 119%, 72%, and 44% in comparison with the dummy predictor. Similary the gain of SVM predictor results over the Zero model are, 18%, 15%, and 24% for *Mylyn*, *Eclipse Platform*, and *Android Platform* respectively.

In overall the predictor model based on group Patch+Human ran by SVM is a good predictor model for all three systems because 1) with out adding the cost of mining and extracting the bug features from macro repositories we could achieve the same results in comparison with the predictor model based on the information from bug repositories. 2) it's applicable for the systems such as *Android Platform* which the relevant bug id for a patch is not traceable in code review repository.

## 8.7 Threats to Validity

*Different Review Processes and Tools:* We only considered *Gerrit*'s review process. We did not consider other code review tools and process such as emails, and pull requests in *git* or *GitHub*.

*Developer and Reviewers feedback:* We did not ask the developers and reviewers of these three systems to provide comments and feedback to our findings.

*Logistic Regression Model:* In our study, a patch is classifies as accepted when the logistic regression model gives a probability that is greater than 0.6. Different thresholds can provide different results.

*Android Platform's Bugs Information:* We could not provide the complete statistics and results for *Android Platform* because of the lack of traceability between code review and bug repository.

*Only Considering the p-value from Logistic Regression Model:* To explore what features have statistically significant impact on the output of code review we only consider the p-values

from logistic regression model. There are different statistical tests such as Kruskal-Wallis analysis of variance [61] which can be used instead.

# CHAPTER 9

## Conclusions and Future Work

In our conducted research we investigated and presented automated solutions for several maintenance tasks such as, Change Impact Analysis, Developer Recommendation, Code Reviewer Recommendation, and Patch Acceptance Predictor. In this chapter, we outline the contributions of this dissertation and suggest opportunities for future research.

### 9.1 Contribution and Findings

The goal of the conducted research is to provide automated solutions for several software maintenance tasks to assist developers and reviewers during the maintenance phase. The resulting solutions from the previous studies are sub-optimal, e.g., in terms of accuracy and coverage, whereby limiting or questioning their practical ubiquity in the software development workflow. We conjecture that the past solutions are severely hampered due to their principal reliance on a limited source of analysis. In order to improve the accuracy and coverage of the solutions we investigated different source of information from developers and reviewers maintenance activities which captures the means, i.e., micro events, associated to achieve the maintenance and evolution tasks. Our formulated methodology consists of the restrained use of machine learning techniques, lightweight source code analysis, and mathematical quantification of different markers of developer and reviewer expertise from these micro events. We ran several empirical studies on large open source and commercial systems to assess the effectiveness of our presented solutions. Below, we reiterate the empirical observations of this dissertation:

#### 9.1.1 *InComIA*, an Approach for Change Impact Analysis

We presented an approach based on a combination of interaction (e.g., *Mylyn*) and commit (e.g., CVS) histories to perform impact analysis (IA) of an incoming change request on source code. The approach requires only the entities that were interacted and/or committed in the past, which differs from the previous solutions that require indexing of a complete snapshot (e.g., a release)

or past commits alone. We conducted an empirical study on the open source system *Mylyn*. We empirically compared our approach to those using commits and the entire source code from a single snapshot. The results show that the presented approach outperformed the baseline competitors. Lowest recall gains of 28% and 44%, highest recall gains of 225% and 350%, lowest precision gains of 28% and 33%, and highest precision gains of 250% and 350% were recorded.

### 9.1.2 *RevIA*, an Approach for Change Impact Analysis

We presented an approach which uses the code review comments provided by the reviewers in code review phase as an additional textual information to form the corpus. Such an approach that combines the code review comments with the textual information from the source code entities for IA at the change request level was not previously investigated. The presented approach differs from the previous solutions that require indexing of a complete snapshot (e.g., a release) or past commits alone. We conducted an empirical study on the open source system *Mylyn*. We empirically compared our approach to the approach that uses only the textual information from the source code entities. The results show that the presented approach outperformed the baseline competitor in terms of precision, recall, and mean average precision. The results show that adding review comments to the corpus improves the performance. Lowest recall, precision, and mean average precision gains of 67%, 100%, and 50% and highest recall, precision, and mean average precision gains of 126%, 166%, and 260% were recorded.

### 9.1.3 *iHDev*, an Approach for Developer Recommendation

We presented the *iHDev* approach to recommend developers who are most likely to implement the incoming change requests. *iHDev* utilizes the archives of developers' interactions within an integrated development environment (e.g., software entities viewed or modified) associated with past change requests. Such use of interaction histories in conjunction with machine learning techniques to recommend developers was not investigated previously. Furthermore, a comparative study with two previous approaches that use commit histories and/or the source code authorship information for developer recommendation was performed. Results show that *iHDev* could provide a

statistically significant recall gain of up to 127.27% with equivalent or improved MRR values by up to 112.5%.

### 9.1.4 *rDevX*, the Developer Expertise Model

We presented a developer-expertise model based on lightweight markers of developer expertise from past code reviews of a software system. We also showed an application of this model, *rDevX*, in automatically recommending the developers who should resolve an incoming software change request (e.g., a bug report or feature request). Our empirical investigation on three open source projects, *Eclipse Platform*, *Mylyn*, and *OpenStack Nova* shows that *rDevX* performs better than two other approaches based on historical commits and their combination. *rDevX* registers recall gains as much as 75% over the other approach. These gains came without incurring any decreased Mean Reciprocal Rank (MRR) values; rather *rDevX* recorded improvements in them. That is, *rDevX* would typically recommend the correct developers at higher ranks than the competitor.

We found evidence to attribute the gains of *rDevX* to the exclusive (human roles and discourse, and alternative attempts in code changes) and subsuming (authors and accepted changes – commits) information captured in code review archives.

### 9.1.5 *cHRev*, the Code Reviewer Recommendation Model

We presented an approach *cHRev* to automatically recommend peer reviewers in modern code review. An empirical study on one commercial and three open source systems showed that our approach provides improved precision and recall over the state of the art competitors. Our results show the added value of analyzing specific information available from previously completed reviews (i.e., quantification of review comments and their recency) for peer reviewer recommendations. We also observed that a developer recommendation approach based on past source code commits was inadequate in effectively supporting this task. However, our experience from this investigation shows that the general principles of frequency, workdays, and recency from such a developer recommendation approach are transformative. That is, these measures when computed on past code reviews instead of commits are very effective for the task of reviewer recommendation.

### 9.1.6 Patch Acceptance Predictor

We examined bug, patch, developer, and reviewer features that characterize the patches that get accepted or not. We used logistic regression to form a descriptive model from these features. Moreover, we presented a predictive model that classifies whether a patch will be accepted or not as soon as it is submitted. Logistic regression and support vector machine were employed to form the patch predictors.

To validate our approach, we conducted an empirical study on three open source projects *Android Platform*, *Eclipse Platform*, and *Mylyn*. These systems require a peer review of submitted patches, which is managed by *Gerrit*. Our results showed that features such as, owner bug assignee match (e.g., the developer who is assigned to resolve the issue in the bug tracking system is the same as the one who implemented the code change and submitted it for review), the reputation of developers in terms of their successful history of patch acceptance, the number of revisions made to the patch, and the number of reviewers greatly influence the likelihood of a patch getting accepted. Our predictor model achieved the average accuracy values of 68%, 93%, and 92% on *Android Platform*, *Eclipse Platform*, and *Mylyn* respectively. Furthermore, we noted that support vector machine performs better than logistic regression for patch acceptance prediction.

## 9.2 Opportunities for Future Research

In our conducted research we presented several automated solutions for different software maintenance tasks by utilizing software systems repositories. Our empirical studies prove that these solutions outperform the state of art and make a major contribution towards helping developers and improving Software maintenance process. However, there are several opportunities for future work which are listed below.

### 9.2.1 Combining Static and Dynamic Analysis with Textual Information

In Chapters 3 and 4 we presented two different approaches for IA based on combination of textual information from different Software repositories. However, there are previous studies that provide the benefits of of using static and dynamic analysis for task IA [41]. Thus, future

work should include this additional information to present an approach for task IA. Similarly, the future work can investigate an approach based on combination of both interaction and code review history.

### 9.2.2 a Developer Recommendation Approach Considering the Workload Balancing

In Chapters 5 and 6 we presented two different approaches for DR. The presented approaches recommend a list of ranked expert developers based on their expertise score to resolve the change request. Developers with the higher expertise and broad knowledge of Software project are more likely to be selected as suitable developers to fix the bug. This may cause developers feel overwhelmed by the large amount of assigned tasks. Thus, future work should consider the developer's workload in recommendation.

### 9.2.3 Considering the Contribution of Developers of Similar Change Requests

The presented approaches in Chapters 5 and 6 consider developers activities of the relevant source code files to the change request. The future work should also consider the contribution of developers who were involved in the resolution of the similar change requests in bug repository.

### 9.2.4 Developer and Reviewers Expertise Markers from Code Review Contents

The presented approaches in Chapters 6 and 7 use code review activities of developers to build the developer and reviewer expertise models. Previous studies suggested that code review comments are indicators of developers expertise [82]. The future work can perform the textual analysis and modeling of review comments to obtain any possible expertise markers from the review content. These expertise can then be used in future developers expertise models.

### 9.2.5 Revisiting Code Reviewer Recommendation

The presented approach in Chapter 7 recommends peer reviewers in modern code review for a specific source code under review. The assignment of reviewer to a specific task is based on his/her previous review activity. This assignment can be improved by providing more detail information about the code change itself. For example, the specific part of code change under

review which has more complexity and needs more attention during the review process. The future work can provide this information in addition of recommending the reviewers for a review task.

### 9.2.6 Additional Empirical Studies Considering Different MCR Tools

The code review data we used in our studies in Chapters 6, 7, and 8 was extracted from *Gerrit* and *CodeFlow*. Although we used several open source systems and one industrial project in our empirical studies, our observations may not be generalized to all Software systems and MCR tools. The future work should revisit code review studies that are used in other systems with different MCR tools such as pull requests in GitHub and perform additional empirical studies based on those.

### 9.2.7 Revisiting the Patch Acceptance Predictor

We presented a patch predictor classifier in Chapter 8 which determines whether a submitted patch will be accepted or not. This predictor considers several features, such as bug, patch, developer, and reviewer features. Although the patch predictor classify the patch as accepted or not with a high accuracy, but it does not provide suggestions to developers on how to increase the acceptance chance of their submitted patches. The future work can provide these suggestions to developers.

# REFERENCES

## LIST OF REFERENCES


[1] Gerrit. https://code.google.com/p/gerrit/. Accessed: 2014-12-08.

[2] Google mondrian: web-based code review and storage. http://www.niallkennedy.com/blog/2006/11/google-mondrian.html. Accessed: 2014-12-08.

[3] Mylyn reviews. https://projects.eclipse.org/projects/mylyn.reviews. Accessed: 2014-12-08.

[4] Phabricator. http://phabricator.org/. Accessed: 2014-12-08.

[5] A. Frank Ackerman, Lynne S. Buchwald, and Frank H. Lewski. Software inspections: An effective verification process. *IEEE Softw.*, 6(3):31–36, May 1989.

[6] A. Frank Ackerman, Priscilla J. Fowler, and Robert G. Ebenau. Software inspections and the industrial production of software. In *Proc. Of a Symposium on Software Validation: Inspection-testing-verification-alternatives*, pages 13–40, New York, NY, USA, 1984. Elsevier North-Holland, Inc.

[7] J. Anvik and G.C. Murphy. Determining implementation expertise from bug reports. In *proceedings of fourth International Workshop on Mining Software Repositories (MSR), 2007 ICSE Workshops MSR '07*, pages 2–2, 2007.

[8] John Anvik. Automating bug report assignment. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 937–940, New York, NY, USA, 2006. ACM.

[9] John Anvik, Lyndon Hiew, and Gail C. Murphy. Who should fix this bug? In *proceedings of 28th ACM International Conference on Software Engineering*, ICSE '06, pages 361–370, 2006.

[10] John Anvik and Gail C. Murphy. Reducing the effort of bug report triage: Recommenders for development-oriented decisions. *ACM Trans. Softw. Eng. Methodol.*, 20(3):10:1–10:35, August 2011.

[11] Robert S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.

[12] Robert S. Arnold and Shawn A. Bohner. Impact analysis - towards a framework for comparison. In *Proceedings of the Conference on Software Maintenance*, ICSM '93, pages 292–301, Washington, DC, USA, 1993. IEEE Computer Society.

## LIST OF REFERENCES (continued)


[13] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 35th International Conference on Software Engineering*, 2013.

[14] Vipin Balachandran. Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 931–940. IEEE Press, 2013.

[15] F. Bantelay, M.B. Zanjani, and H. Kagdi. Comparing and combining evolutionary couplings from interactions and commits. In *Proceedings of the 20th Working Conference on Reverse Engineering (WCRE)*, pages 311–320, Oct 2013.

[16] O. Baysal, M.W. Godfrey, and R. Cohen. A bug you like: A framework for automated assignment of bugs. In *Program Comprehension, 2009. ICPC '09. IEEE 17th International Conference on*, pages 297–298, May 2009.

[17] O. Baysal, O. Kononenko, R. Holmes, and M.W. Godfrey. The influence of non-technical factors on code review. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*, pages 122–131, Oct 2013.

[18] Olga Baysal, Oleksii Kononenko, Reid Holmes, and Michael W. Godfrey. The secret life of patches: A firefox case study. In *Proceedings of the 2012 19th Working Conference on Reverse Engineering*, WCRE '12, pages 447–455, Washington, DC, USA, 2012. IEEE Computer Society.

[19] Olga Baysal, Oleksii Kononenko, Reid Holmes, and Michael W. Godfrey. Investigating technical and non-technical factors influencing modern code review. *Empirical Software Engineering*, 21(3):932–959, 2016.

[20] Andrew Begel, Yit Phang Khoo, and Thomas Zimmermann. Codebook: Discovering and exploiting relationships in software repositories. In *proceedings of 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 125–134, 2010.

[21] Moritz Beller, Alberto Bacchelli, Andy Zaidman, and Elmar Juergens. Modern code reviews in open-source projects: Which problems do they fix? In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 202–211, New York, NY, USA, 2014. ACM.

[22] M. Bernhart, A. Mauczka, and T. Grechenig. Adopting code reviews for agile software development. In *Agile Conference (AGILE), 2010*, pages 44–47, Aug 2010.

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

## LIST OF REFERENCES (continued)


[24] Christian Bird, Trevor Carnahan, and Michaela Greiler. Lessons learned from deploying a code review analytics platform. Technical Report MSR-TR-2015-22 (Under submission to MSR 2015), Microsoft Research, `http://research.microsoft.com/apps/pubs/default.aspx?id=241497`, February 2015.

[25] Christian Bird, Alex Gourley, Prem Devanbu, Michael Gertz, and Anand Swaminathan. Mining email social networks. In *proceedings of 2006 International Workshop on Mining Software Repositories*, MSR '06, pages 137–143, 2006.

[26] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. Don't Touch My Code! Examining the Effects of Ownership on Software Quality. In *Proceedings of the the eighth joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*. ACM, 2011.

[27] Gerald Bortis and André van der Hoek. Porchlight: A tag-based approach to bug triaging. In *proceedings of 2013 International Conference on Software Engineering*, ICSE '13, pages 342–351. IEEE Press, 2013.

[28] Roger B. Bradford. An empirical study of required dimensionality for large-scale latent semantic indexing applications. In *Proceedings of the 17th ACM Conference on Information and Knowledge Management*, CIKM '08, pages 153–162, New York, NY, USA, 2008. ACM.

[29] L.C. Briand, J. Wust, and H. Lounis. Using coupling measurement for impact analysis in object-oriented systems. In *Proceedings of the IEEE International Conference on Software Maintenance, 1999. (ICSM '99)*, pages 475–482, 1999.

[30] Gerardo Canfora and Luigi Cerulo. Fine grained indexing of software repositories to support impact analysis. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, MSR '06, pages 105–111, New York, NY, USA, 2006. ACM.

[31] Li-Te Cheng, Cleidson R.B. de Souza, Susanne Hupfer, John Patterson, and Steven Ross. Building collaboration into ides. *Queue*, 1(9):40–50, December 2003.

[32] J. Cohen. *Best Kept Secrets of Peer Code Review*. Smart Bear Inc, Austin, TX, USA, 2006.

[33] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995.

[34] D. R. Cox. The regression analysis of binary sequences. *Journal of the Royal Statistical Society. Series B (Methodological)*, 20(2):215–242, 1958.

## LIST OF REFERENCES (continued)


[35] Davor Cubranic. Automatic bug triage using text categorization. In *In SEKE 2004: Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering*, pages 92–97. KSI Press, 2004.

[36] Jacek Czerwonka, Nachiappan Nagappan, Wolfram Schulte, and Brendan Murphy. CODEMINE: building a software development data analytics platform at microsoft. *IEEE Software*, 30(4):64–71, 2013.

[37] R. DeLine, Mary Czerwinski, and G. Robertson. Easing program comprehension by sharing navigation data. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 241–248, Sept 2005.

[38] M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 38(2.3):258–287, 1999.

[39] Thomas Fritz, Jingwen Ou, Gail C. Murphy, and Emerson Murphy-Hill. 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, pages 385–394, New York, NY, USA, 2010. ACM.

[40] D.M. German. An empirical study of fine-grained software modifications. In *proceedings of 20th IEEE International Conference on Software Maintenance, 2004.*, pages 316–325, 2004.

[41] Malcom Gethers, Bogdan Dit, Huzefa Kagdi, and Denys Poshyvanyk. Integrated impact analysis for managing software changes. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 430–440, Piscataway, NJ, USA, 2012. IEEE Press.

[42] Jesús M. González-Barahona, Daniel Izquierdo-Cortazar, Gregorio Robles, and Alvaro del Castillo. Analyzing gerrit code review parameters with bicho. *ECEASST*, pages –1–1, 2014.

[43] Georgios Gousios, Martin Pinzger, and Arie van Deursen. An exploratory study of the pull-based software development model. In *ICSE*, pages 345–355, 2014.

[44] V.J. Hellendoorn, P.T. Devanbu, and A. Bacchelli. Will they like this? evaluating code contributions with language models. In *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on*, pages 157–167, May 2015.

[45] Israel Herraiz, Daniel M. German, Jesus M. Gonzalez-Barahona, and Gregorio Robles. Towards a simplification of the bug report form in eclipse. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, MSR '08, pages 145–148, New York, NY, USA, 2008. ACM.

[46] K. Hossen, H. Kagdi, and D Poshyvanyk. Amalgamating source code authors, maintainers, and change proneness to triage change requests. In *Proceedings of the 22th IEEE International Conference on Program Comprehension, ICPC*, 2014.

[47] Gaeul Jeong, Sunghun Kim, and Thomas Zimmermann. Improving bug triage with bug tossing graphs. In *proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 111–120, New York, NY, USA, 2009. ACM.

[48] Gaeul Jeong, Sunghun Kim, Thomas Zimmermann, and Kwangkeun Yi. Improving code review by predicting reviewers and acceptance of patches. Technical Report ROSAEC MEMO 2009-006, Research on Software Analysis for Error-free Computing Center, September 2009.

[49] Yujuan Jiang, B. Adams, and D.M. German. Will my patch make it? and how fast? case study on the linux kernel. In *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*, pages 101–110, May 2013.

[50] Yujuan Jiang, B. Adams, and D.M. German. Will my patch make it? and how fast? case study on the linux kernel. In *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*, pages 101–110, May 2013.

[51] H. Kagdi, M. Gethers, D. Poshyvanyk, and M.L. Collard. Blending conceptual and evolutionary couplings to support change impact analysis in source code. In *Proceedings of the 17th Working Conference on Reverse Engineering (WCRE)*, pages 119–128, Oct 2010.

[52] H. Kagdi, M. Hammad, and J.I. Maletic. Who can help me with this source code change? In *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pages 157–166, Sept 2008.

[53] Huzefa Kagdi, Malcom Gethers, Denys Poshyvanyk, and Maen Hammad. Assigning change requests to software developers. *Journal of Software: Evolution and Process*, 24(1):3–33, 2012.

[54] Huzefa Kagdi, Malcom Gethers, Denys Poshyvanyk, and Maen Hammad. Assigning change requests to software developers. *Journal of Software: Evolution and Process*, 24(1):3–33, 2012.

[55] C.F. Kemerer and M.C. Paulk. The impact of design and code reviews on software quality: An empirical study based on psp data. *Software Engineering, IEEE Transactions on*, 35(4):534–550, July 2009.

[56] Miryung Kim and David Notkin. Discovering and representing systematic code changes. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 309–319, Washington, DC, USA, 2009. IEEE Computer Society.

[57] S. Kim, E. J. Whitehead Jr., and Y. Zhang. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering*, 34(2):181–196, March 2008.

[58] Sunghun Kim, E.J. Whitehead, and Yi Zhang. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering*, 34(2):181–196, March 2008.

[59] T. Kobayashi, N. Kato, and K. Agusa. Interaction histories mining for software change guide. In *Proceedings of the Third International Workshop on Recommendation Systems for Software Engineering (RSSE)*, pages 73–77, June 2012.

[60] Sami Kollanus and Jussi Koskinen. Survey of software inspection research. *The Open Software Engineering Journal*, 3:15–34, 2009.

[61] W. H. Kruskal and W. A. Wallis. Use of ranks in one-criterion variance analysis. *IEEE Transactions on Software Engineering*, 47(260):583–621, June 1952.

[62] J. Law and G. Rothermel. Whole program path-based dynamic impact analysis. In *Proceedings of the 25th International Conference on Software Engineering*, pages 308–318, May 2003.

[63] Michelle Lee, A. Jefferson Offutt, and Roger T. Alexander. Algorithmic analysis of the impacts of changes to object-oriented software. In *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 34'00)*, TOOLS '00, pages 61–, Washington, DC, USA, 2000. IEEE Computer Society.

[64] Taek Lee, Jaechang Nam, DongGyun Han, Sunghun Kim, and Hoh Peter In. Micro interaction metrics for defect prediction. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 311–321, New York, NY, USA, 2011. ACM.

[65] M. Linares-Vasquez, K. Hossen, Hoang Dang, H. Kagdi, M. Gethers, and D. Poshyvanyk. Triaging incoming change requests: Bug or commit history, or code authorship? In *proceedings of 28th IEEE International Conference on Software Maintenance (ICSM), 2012*, pages 451–460, 2012.

[66] M. Linares-Vasquez, K. Hossen, Hoang Dang, H. Kagdi, M. Gethers, and D. Poshyvanyk. Triaging incoming change requests: Bug or commit history, or code authorship? In *Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 451–460, Sept 2012.

## LIST OF REFERENCES (continued)


[67] D. Ma, D. Schuler, T. Zimmermann, and J. Sillito. Expert recommendation with usage expertise. In *proceedings of IEEE International Conference on Software Maintenance,ICSM 2009.*, pages 535–538, 2009.

[68] D. Matter, A. Kuhn, and O. Nierstrasz. Assigning bug reports using a vocabulary-based expertise model of developers. In *proceedings of 6th IEEE International Working Conference on Mining Software Repositories, 2009. MSR '09.*, pages 131–140, 2009.

[69] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 192–201, New York, NY, USA, 2014. ACM.

[70] Luca Milanesio. *Learning Gerrit Code Review*. Packt Publishing Ltd, 2013.

[71] Audris Mockus and James D. Herbsleb. Expertise browser: A quantitative approach to identifying expertise. In *proceedings of 24th International Conference on Software Engineering*, ICSE '02, pages 503–512, New York, NY, USA, 2002. ACM.

[72] Rodrigo Morales, Shane McIntosh, and Foutse Khomh. Do code review practices impact design quality? a case study of the qt, vtk, and itk projects. In *Proc. of the 22nd Int'l Conf. on Software Analysis, Evolution, and Reengineering (SANER)*, pages 171–180, 2015.

[73] Murtuza Mukadam, Christian Bird, and Peter C. Rigby. Gerrit software code review data from android. In *Proceedings of the International Working Conference on Mining Software Repositories (Data Track)*. IEEE, 2013.

[74] G.C. Murphy, M. Kersten, and L. Findlater. How are java software developers using the elipse ide? *IEEE Transactions on Software Engineering*, 23(4):76–83, July 2006.

[75] Alessandro Orso, Taweesup Apiwattanapong, James Law, Gregg Rothermel, and Mary Jean Harrold. An empirical comparison of dynamic impact analysis algorithms. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 491–500, Washington, DC, USA, 2004. IEEE Computer Society.

[76] C. Parnin and C. Gorg. Building usage contexts during program comprehension. In *Proceedings of the 14th IEEE International Conference on Program Comprehension, ICPC*, pages 13–22, 2006.

[77] Adam Porter, Harvey Siy, Audris Mockus, and Lawrence Votta. Understanding the sources of variation in software inspections. *ACM Trans. Softw. Eng. Methodol.*, 7(1):41–79, January 1998.

[78] Denys Poshyvanyk, Andrian Marcus, Rudolf Ferenc, and Tibor Gyimóthy. Using information retrieval based coupling measures for impact analysis. *Empirical Softw. Engg.*, 14(1):5–32, February 2009.

[79] S. Rastkar and G.C. Murphy. On what basis to recommend: Changesets or interactions? In *Proceedings of the 6th IEEE International Working Conference on Mining Software Repositories, MSR '09*, pages 155–158, May 2009.

[80] Xiaoxia Ren, B.G. Ryder, M. Stoerzer, and F. Tip. Chianti: a change impact analysis tool for java programs. In *Proceedings of the 27th International Conference on Software Engineering,ICSE*, pages 664–665, May 2005.

[81] P.C. Rigby, B. Cleary, F. Painchaud, M. Storey, and D.M. German. Contemporary peer review in action: Lessons from open source development. *Software, IEEE*, 29(6):56–61, Nov 2012.

[82] Peter C. Rigby and Christian Bird. Convergent software peer review practices. In *Proceedings of the the joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE)*. ACM, 2013.

[83] Peter C. Rigby, Daniel M. German, and Margaret-Anne Storey. Open source software peer review practices: A case study of the apache server. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 541–550, New York, NY, USA, 2008. ACM.

[84] Peter C. Rigby and Margaret-Anne Storey. Understanding broadcast based peer review on open source software projects. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 541–550, New York, NY, USA, 2011. ACM.

[85] R. Robbes. Mining a change-based software repository. In *Proceedings of the Fourth International Workshop on Mining Software Repositories, 2007. ICSE Workshops MSR '07.*, pages 15–15, May 2007.

[86] R. Robbes, D. Pollet, and M. Lanza. Replaying ide interactions to evaluate and improve change prediction approaches. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 161–170, May 2010.

[87] Romain Robbes and David Röthlisberger. Using developer interaction data to compare expertise metrics. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 297–300, Piscataway, NJ, USA, 2013. IEEE Press.

[88] D. Rothlisberger, O. Nierstrasz, S. Ducasse, D. Pollet, and R. Robbes. Supporting task-oriented navigation in ides with configurable heatmaps. In *Proceedings of the IEEE 17th International Conference on Program Comprehension,ICPC '09*, pages 253–257, May 2009.

[89] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164, 2009.

[90] Kevin Schneider, Carl Gutwin, Reagan Penner, and David Paquette. Mining a software developer's local interaction history. In *Proceedings of the IEEE International Conference on Software Engineering Workshop on Mining Software Repositories,*, 2004.

[91] M. Shafiei, Singer Wang, R. Zhang, E. Milios, Bin Tang, J. Tougas, and R. Spiteri. Document representation and dimension reduction for text clustering. In *Proceedings of the IEEE 23rd International Conference on Data Engineering Workshop*, pages 770–779, 2007.

[92] Ramin Shokripour, John Anvik, Zarinah Mohd Kasirun, and Sima Zamani. Why so complicated? simple term filtering and weighting for location-based bug report assignment recommendation. In *proceedings of 10th IEEE Working Conference on Mining Software Repositories (MSR), 2013*, pages 2–11, 2013.

[93] J. Singer, R. Elves, and M. Storey. Navtracks: supporting navigation in software maintenance. In *Proceedings of the 21st IEEE International Conference on Software Maintenance, ICSM'05*, pages 325–334, Sept 2005.

[94] Ahmed Tamrawi, Tung Thanh Nguyen, Jafar M. Al-Kofahi, and Tien N. Nguyen. Fuzzy set and cache-based approach for bug triaging. In *proceedings of 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 365–375, 2011.

[95] Y. Tao, D. Han, and S. Kim. Writing acceptable patches: An empirical study of open source project patches. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pages 271–280, Sept 2014.

[96] Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Raula Gaikovina Kula, Norihiro Yoshida, Hajimu Iida, and Ken ichi Matsumoto. Who should review my code? a file location-based code-reviewer recommendation approach for modern code review. In *the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 141–150, 2015.

[97] F. Thung, Shaowei Wang, D. Lo, and Lingxiao Jiang. An empirical study of bugs in machine learning systems. In *Proceedings of the IEEE 23rd International Symposium on Software Reliability Engineering (ISSRE)*, pages 271–280, Nov 2012.

[98] Y. Tian, M. Nagappan, D. Lo, and A.E. Hassan. Who should review this change?: Putting text and file location analyses together for more accurate recommendations. In *Proceedings of the 31st IEEE International Conference on Software Maintenance and Evolution, ICSME 2015*, September 2015.

[99] Yuan Tian, D. Lo, and Chengnian Sun. Information retrieval based nearest neighbor classification for fine-grained bug severity prediction. In *Proceedings of the 19th Working Conference on Reverse Engineering (WCRE)*, pages 215–224, Oct 2012.

[100] Shaowei Wang and David Lo. Version history, similar report, and structure: Putting them together for improved bug localization. In *Proceedings of the 22Nd International Conference on Program Comprehension*, ICPC 2014, pages 53–63, New York, NY, USA, 2014. ACM.

[101] Cathrin Weiss, Rahul Premraj, Thomas Zimmermann, and Andreas Zeller. How long will it take to fix this bug? In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, pages 1–, Washington, DC, USA, 2007. IEEE Computer Society.

[102] P. Weissgerber, M. Pohl, and M. Burch. Visual data mining in software archives to detect how developers work together. In *proceedings of fourth International Workshop on Mining Software Repositories, 2007. ICSE Workshops MSR '07.*, pages 9–9, 2007.

[103] Peter Weissgerber, Daniel Neu, and Stephan Diehl. Small patches get in! In *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, MSR '08, pages 67–76, New York, NY, USA, 2008. ACM.

[104] Murray Wood, Marc Roper, Andrew Brooks, and James Miller. Comparing and combining software defect detection techniques: A replicated empirical study. *SIGSOFT Softw. Eng. Notes*, 22(6):262–277, November 1997.

[105] Wenjin Wu, Wen Zhang, Ye Yang, and Qing Wang. Drex: Developer recommendation with k-nearest-neighbor search and expertise ranking. In *Proceedings of the 18th Asia Pacific Software Engineering Conference (APSEC)*, pages 389–396, Dec 2011.

[106] Xin Xia, D. Lo, Xinyu Wang, and Bo Zhou. Accurate developer recommendation for bug resolution. In *Proceedings of the 20th Working Conference on Reverse Engineering (WCRE)*, pages 72–81, Oct 2013.

[107] Xin Xia, David Lo, Xinyu Wang, and Bo Zhou. Accurate developer recommendation for bug resolution. In *Proceedings of 20th Working Conference on Reverse Engineering (WCRE), 2013*, pages 72–81, 2013.

## LIST OF REFERENCES (continued)


[108] Jifeng Xuan, He Jiang, Zhilei Ren, and Weiqin Zou. Developer prioritization in bug repositories. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 25–35, Piscataway, NJ, USA, 2012. IEEE Press.

[109] Cong Yu, Laks Lakshmanan, and Sihem Amer-Yahia. It takes variety to make a world: Diversification in recommender systems. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '09, pages 368–378, New York, NY, USA, 2009. ACM.

[110] M. Bahrami Zanjani, H. Kagdi, and C. Bird. Automatically recommending peer reviewers in modern code review. *IEEE Transactions on Software Engineering*, PP(99):1–1, 2015.

[111] M.B. Zanjani, H. Kagdi, and C. Bird. Using developer-interaction trails to triage change requests. In *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on*, pages 88–98, May 2015.

[112] M.B. Zanjani, G. Swartzendruber, and H. Kagdi. Impact analysis of change requests on source code based on interaction and commit histories. In *proceedings of the 11th Working Conference on Mining Software Repositories, MSR '14*, MSR '14, 2014.

[113] Min-Ling Zhang and Zhi-Hua Zhou. Ml-knn: A lazy learning approach to multi-label learning. *Pattern Recogn.*, 40(7):2038–2048, July 2007.

[114] Tianyi Zhang, Myoungkyu Song, and Miryung Kim. Critics: An interactive code review tool for searching and inspecting systematic changes. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 755–758, New York, NY, USA, 2014. ACM.

[115] Lijie Zou, M.W. Godfrey, and A.E. Hassan. Detecting interaction coupling from task interaction histories. In *Proceedings of the 15th IEEE International Conference on Program Comprehension,ICPC '07*, pages 135–144, June 2007.