Automatically Recommending Peer Reviewers in Modern Code Review

Motahareh Bahrami Zanjani, *Student Member, IEEE*,
Huzefa Kagdi, *Member, IEEE*, and Christian Bird, *Member, IEEE*

Abstract—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. 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 coderepository data. We also present the insights into why our approach *cHRev* outperforms the existing approaches.

Index Terms—Modern code review, reviewer recommendation, code change, Gerrit

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 [1], [2]. Recently, a number of empirical studies about various facets of the modern code review process have been reported in the literature [3], [4], [5], [6], [7], [8], [9], [10], [11]. 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 and Bird [12] 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 and Storey [11] 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 [4], [12]. 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 [13]. Unfortunately, there has been little effort in building automatic approaches to recommend the most suitable reviewers in modern (tool-based)

- *M.B. Zanjani and H. Kagdi are with the Department of Electrical Engineering and Computer Science, Wichita State University, Wichita, Kansas 6760. E-mail: {mxbahramizanjani, huzefa.kagdi}@wichita.edu.*
- *C. Bird is with Microsoft Research, Redmond, WA. E-mail: cbird@microsoft.com.*

Manuscript received 2 July 2015; accepted 5 Oct. 2015. Date of publication 11 Nov. 2015; date of current version 20 June 2016.
Recommended for acceptance by G. Murphy.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2015.2500238

code review process, which includes the work of Xia et al. [14], Thongtanunam et al. [15], and Balachandran [16]. Balachandran termed the task of identifying the most appropriate reviewers for a change as *Reviewer Recommendation*.

This paper 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* [15], *xFinder* [17], 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 paper 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 the paper is organized as follows: Section 2 presents background of modern code review and associated terminology. Our approach is discussed in Section 3. The empirical study on *Android Platform*, *Eclipse Platform*, *Mylyn*, and *MS Office*, and its results are presented in Section 4. Threats to validity are encountered in Section 5. Related work is discussed in Section 6. Finally, our conclusions and future work are stated in Section 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 [3]. Developers make local changes in their private *git* repositories and then submit these changes as a patch for review [4]. 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 [12] has found that higher expertise of reviewers leads to both.

3 The Proposed *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.

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 [18]. 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 [17]. 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:

$$xFactor(r,f) = \frac{RE_{(r,f)}}{FR_{(f)}} \tag{1}$$

$$xFactor(r,f) = \begin{cases} \frac{C_f}{C'_f} + \frac{W_f}{W'_f} + \frac{1}{|T_f - T'_f|} & \text{if } |T_f - T'_f| \neq 0 \\ \frac{C_f}{C'_f} + \frac{W_f}{W'_f} + 1 & \text{if } |T_f - T'_f| = 0 \end{cases}. \tag{2}$$

The *xFactor* score is computed for each of the source-code files that exist in the code change. According to Equation (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.

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 (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 jth reviewer in the file f_i with l unique reviewers,

$$D_r = \begin{pmatrix} D_{rf_1} \\ D_{rf_2} \\ .. \\ .. \\ D_{rf_n} \end{pmatrix} D_{rf_i} = \{\, r_{i1} \quad r_{i2} \quad \cdots \quad r_{il} \,\}. \tag{3}$$

Although, a single file does not have any duplicate reviewers, two files may have common reviewers. In Equation (4), D_{ru} is the union of all unique reviewers from all files,

$$D_{ru} = \bigcup\nolimits_{i=1}^{n} D_{rf_i} \qquad Score(r) = \sum_{i=1}^{n} xFactor_i(r, f_i). \tag{4}$$

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

In Equation (5), 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:

$$RF = \{(r, Score(r)), \forall r \in D_{ru}\}. \tag{5}$$

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 1
The Reviewers Extracted with *cHRev* from Each of the Files Related to Code Change in the Review # 33689

Files	Reviewers and their *x Factor*
.../TaskListFilteredTree.java	Sam Davis:3.00
.../CustomTaskListDecorationDrawer.java	Frank Becker: 1.83 , Sam Davis:1.62 , Tomasz Zarna:1.52

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.

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 4.2 for details). The review of interest here is the review #33689: *"clean up workspace 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 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 1). The set D_{ru} consists of three 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 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 three unique reviewers, the $Score$ value is calculated according to Equation (4). Table 2 shows the top four $Score$ values and the corresponding reviewers, i.e.,

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

TABLE 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	-	-	-	-

$m = 4$ for four approaches: *cHRev*, *REVFINDER*, *xFinder*, and *RevCom*. *Score* values for *REVFINDER* have been calculated in different way (see Section 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 [17]. Even considering this point, Table 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.

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.

4.1 Design

We conducted a case study to empirically assess our approach according to the design and reporting guidelines presented in [19]. 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.

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* [16]. 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 [20]. 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 [21], which served as an inspiration to emulate the combination for the reviewer recommendation task.

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

4.3 Subject Systems and Evaluation Datasets

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

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

Android contains seven 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 2,680 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 23,181 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 1,854 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 10,506 review comments are written in *Eclipse Platform*. The *Eclipse Platform* project consists of 3,155 commits in the commit history (during the defined period), each commit contains a change to at least one Java file.

Mylyn contains about two 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 1,589 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 10,157 review comments were written in *Mylyn*. *Mylyn* consists of 1,838 commits in the commit history (during the defined period), each commit contains a change to at least one Java file.

4.3.2 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* [22] our development analytics database and code review data from *CodeFlow Analytics* [23], 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 [16]. 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 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

Table 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 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.

3. https://android-review.googlesource.com/#/q/platform
4. https://git.eclipse.org/r/#/q/platform,n,z
5. https://git.eclipse.org/r/#/q/mylyn,n,z

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 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.

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 [20], [24], [25]. 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:

$$precision@m = \frac{|\,RR(p) \cap AR(p)\,|}{|\,RR(p)\,|} \tag{6}$$

$$recall@m = \frac{|\,RR(p) \cap AR(p)\,|}{|\,AR(p)\,|} \tag{7}$$

$$F\_score@m = 2.\frac{precision@m.recall@m}{precision@m + recall@m}, \tag{8}$$

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 3 shows the frequency distribution of reviewers for each subject software system in our benchmark.[6] Sixty nine percent of code changes for *Android Platform*, 76 percent of code changes for *Eclipse Platform*, 71 percent of code changes for *Mylyn*, and 64 percent 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 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,

$$MRR = \frac{1}{|\,n\,|}\sum_{i=1}^{|n|}\frac{1}{rank_i}. \tag{9}$$

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 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.

4.6 Results

The number of recommended reviewers is the only user defined parameter for our approach. As can be seen from Table 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 $m = 1$, $m = 2$, $m = 3$, and $m = 5$, where m is the number

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

TABLE 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	m	Precision@m				Recall@m				F-score@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

of recommended reviewers to provide the realistic view of the performance.

To answer the research $RQ1$, we consult Table 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 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:

$$GainX@m_{cHRev-Y} = \frac{X@m_{cHRev} - X@m_Y}{X@m_Y} \times 100. \quad (10)$$

Tables 5 and 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 and 8). In these exceptional cases, both were statistically equivalent. The gains in *Eclipse*

TABLE 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	m	Precision: Gain*P cHRev-*			Gain*P RevCom-xFinder%*	Recall: Gain*R cHRev-*			Gain*R RevCom-xFinder%*	F-score: Gain*F cHRev-*			Gain*F RevCom-xFinder%*
		REVFINDER%	*xFinder%*	*RevCom%*		*REVFINDER%*	*xFinder%*	*RevCom%*		*REVFINDER%*	*xFinder%*	*RevCom%*	
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 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
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			
		REVFINDER	*cHRev-xFinder*	*RevCom*	*RevCom-xFinder*	*REVFINDER*	*cHRev-xFinder*	*RevCom*	*RevCom-xFinder*
Mylyn	1	< 0.01	< 0.01	≤ 0.4	< 0.01	< 0.01	< 0.01	≤ 0.8	< 0.01
	2	< 0.01	< 0.01	≤ 0.9	< 0.01	< 0.01	$\equiv 0.00$	≤ 0.9	$\equiv 0.00$
	3	$\equiv 0.00$	< 0.01	≤ 0.9	< 0.01	$\equiv 0.00$	$\equiv 0.00$	≤ 1	$\equiv 0.00$
	5	$\equiv 0.00$	≤ 0.7	≤ 0.9	≤ 0.7	$\equiv 0.00$	$\equiv 0.00$	≤ 1	$\equiv 0.00$
Eclipse Platform	1	≤ 0.1	< 0.01	≤ 0.8	< 0.01	≤ 0.4	< 0.02	≤ 0.7	< 0.04
	2	< 0.02	< 0.01	≤ 0.5	< 0.03	< 0.04	< 0.01	≤ 0.9	< 0.01
	3	< 0.02	< 0.01	≤ 0.3	< 0.01	< 0.03	$\equiv 0.00$	≤ 0.4	< 0.01
	5	$\equiv 0.00$	< 0.04	≤ 0.3	< 0.03	< 0.02	$\equiv 0.00$	≤ 0.9	$\equiv 0.00$
Android Platform	1	< 0.01	< 0.01	≤ 0.8	< 0.01	< 0.03	< 0.01	≤ 0.9	< 0.01
	2	< 0.01	< 0.01	≤ 0.7	< 0.01	< 0.02	$\equiv 0.00$	≤ 0.9	$\equiv 0.00$
	3	< 0.01	$\equiv 0.00$	≤ 0.7	$\equiv 0.00$	< 0.01	$\equiv 0.00$	≤ 0.9	$\equiv 0.00$
	5	< 0.01	$\equiv 0.00$	≤ 0.7	$\equiv 0.00$	< 0.01	$\equiv 0.00$	≤ 0.9	$\equiv 0.00$
MS Office	1	< 0.01	$\equiv 0.00$	≤ 0.7	$\equiv 0.00$	< 0.01	$\equiv 0.00$	≤ 0.7	$\equiv 0.00$
	2	< 0.02	$\equiv 0.00$	≤ 0.8	$\equiv 0.00$	< 0.02	$\equiv 0.00$	≤ 0.9	$\equiv 0.00$
	3	< 0.02	$\equiv 0.00$	≤ 0.9	$\equiv 0.00$	< 0.02	$\equiv 0.00$	≤ 0.9	$\equiv 0.00$
	5	< 0.03	$\equiv 0.00$	≤ 0.9	$\equiv 0.00$	≤ 0.04	$\equiv 0.00$	≤ 0.9	$\equiv 0.00$

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 methodology 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 percent) are substantially better than those achieved on *Android Platform*, *Eclipse Platform*, and *Mylyn* (well below 100 percent). 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.

TABLE 8
p-Values from Applying One Way ANOVA on MRR Values for Each Subject System

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

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 *RQ*2. 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*.

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 [26] 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. [27] observed that in GitHub some issues receive multiple 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 and Bird [4] 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 percent 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. [28] 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.

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 percent 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 [29]. 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 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.

6 Related Work

There has been much investigation in the various aspects of modern code review. We briefly discuss a few representative efforts from this investigation. The reviewer recommendation task has not been examined much in the literature yet.

6.1 Code Review

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.

6.1.1 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 [13]. In contrast contemporary or modern code review encompasses a series of less rigid practices [9], [30], [31]. 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. [10] 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. [32] found that the optimal number of reviewers should be two. Rigby and Bird [4] 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 and Storey [11] describe an empirical study to investigate the mechanisms and behaviors that developers use to find code changes they are competent to review. Bacchelli and Bird [12] 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. [8] 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. [33] 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.

6.1.2 Predicting the Outcome of Code Review

There are several 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. [5] described an empirical study of code review process for WEbKit and their result provides 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. [6] performed data mining on email archives of two open source projects to study patch contributions. They found that smaller patches have a higher chance of being accepted than larger ones. Jiang et al. [7] found that patch acceptance is affected by the developer experience, patch maturity, and priori subsystem churn, whereas, the reviewing time is impacted by the submission time, the number of affected subsystems, and the number of suggested reviewers and developer experience. Jeong et al. [34] examined the review process in two Mozilla projects and presented approaches for suggesting reviewers of OSS patches and predicting whether such patches would be accepted.

6.1.3 Influence of Code Review on the Code Quality

McIntosh et al. [3], [35] studied the effect of code reviews on 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. [36] studied the types of defects fixed in modern code review repositories. Kemerer and Paulk [37] 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.

6.1.4 Tool Support for Code Review

Kim and Notkin [38] developed a tool called *LSdiff* to help reviewers inspect program differences. *LSdiff* covers the limitations of program differencing tools by inferring systematic structural differences into logic rules. Zhang et al. [39] 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) [40]. One of these integrated IDE tools is ReviewClipse [41] and another is Mylyn Reviews [42]. A popular review tool is OSS Gerrit [43], offering web-based reviewing for projects using Git [44]. A number of other review tools: CodeFlow [12], Microsoft code review tool; Phabricator is Facebook's open-sourced tool [45]; Mondrian, a tool that Google uses for its closed-source projects [46]. 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. [47] extracted Android peer review data from Gerrit and provide the data for future empirical software engineering questions. Gonzlez et al. [48] presented an approach to retrieve and analyze the information produced by Gerrit based on a previously designed tool called Bicho.

6.1.5 Reviewer Recommendation

There are only three approaches reported for reviewer recommendations. Balachandran [16] proposed a GIT blame like line oriented approach. Recently, Thongtanunam et al. [15] 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. [14] 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.

6.1.6 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.

6.2 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 [24], [25], [49], [50], [51], [52], [53], [54], [55]. 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.

7 Conclusions and Future Work

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 competitor. 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.

In the future, we plan to include the textual analysis of review comments and additional measures of reviewers' contributions and impact (e.g., the specific code elements and their complexity, and the nature of issues identified and addressed) in our approach.

References

[1] A. F. Ackerman, L. S. Buchwald, and F. H. Lewski, "Software inspections: An effective verification process," *IEEE Softw.*, vol. 6, no. 3, pp. 31–36, May 1989.

[2] A. F. Ackerman, P. J. Fowler, and R. G. Ebenau, "Software inspections and the industrial production of software," in *Proc. Symp. Softw. Validation: Inspection-Testing-Verification-Alternatives*, 1984, pp. 13–40.

[3] S. McIntosh, Y. Kamei, B. Adams, and A. 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 *Proc. 11th Working Conf. Mining Softw. Repositories*, 2014, pp. 192–201.

[4] P. C. Rigby and C. Bird, "Convergent software peer review practices," in *Proc. Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng.*, 2013, pp. 202–212.

[5] O. Baysal, O. Kononenko, R. Holmes, and M. Godfrey, "The influence of non-technical factors on code review," in *Proc. 20th Working Conf. Reverse Eng.*, Oct. 2013, pp. 122–131.

[6] P. Weissgerber, D. Neu, and S. Diehl, "Small patches get in!" in *Proc. Int. Working Conf. Mining Softw. Repositories*, 2008, pp. 67–76.

[7] Y. Jiang, B. Adams, and D. German, "Will my patch make it? and how fast? Case study on the Linux kernel," in *Proc. 10th IEEE Working Conf. Mining Softw. Repositories*, May 2013, pp. 101–110.

[8] O. Baysal, O. Kononenko, R. Holmes, and M. W. Godfrey, "The secret life of patches: A Firefox case study," in *Proc. 19th Working Conf. Reverse Eng.*, 2012, pp. 447–455.

[9] P. Rigby, B. Cleary, F. Painchaud, M. Storey, and D. German, "Contemporary peer review in action: Lessons from open source development," *IEEE Software*, vol. 29, no. 6, pp. 56–61, Nov./Dec. 2012.

[10] P. C. Rigby, D. M. German, and M.-A. Storey, "Open source software peer review practices: A case study of the apache server," in *Proc. 30th Int. Conf. Softw. Eng.*, 2008, pp. 541–550.

[11] P. C. Rigby and M.-A. Storey, "Understanding broadcast based peer review on open source software projects," in *Proc. 33rd Int. Conf. Softw. Eng.*, 2011, pp. 541–550.

[12] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *Proc. 35th Int. Conf. Softw. Eng.*, 2013, pp. 712–721.

[13] M. E. Fagan, "Design and code inspections to reduce errors in program development," *IBM Syst. J.*, vol. 38, no. 2.3, pp. 258–287, 1999.

[14] X. Xia, D. Lo, X. Wang, and X. Yang, "Who should review this change?: Putting text and file location analyses together for more accurate recommendations," in *Proc. 31st IEEE Int. Conf. Softw. Maintenance Evol.*, Sep. 2015, pp. 261–270.

[15] P. Thongtanunam, C. Tantithamthavorn, R. G. Kula, N. Yoshida, H. Iida, and K. Ichi Matsumoto, "Who should review my code? A file location-based code-reviewer recommendation approach for modern code review," in *Proc. 22nd IEEE Int. Conf. Softw. Anal., Evol. Reeng.*, 2015, pp. 141–150.

[16] V. Balachandran, "Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation," in *Proc. Int. Conf. Softw. Eng.*, 2013, pp. 931–940.

[17] H. Kagdi, M. Hammad, and J. Maletic, "Who can help me with this source code change?" in *Proc. IEEE Int. Conf. Softw. Maintenance*, Sep. 2008, pp. 157–166.

[18] R. Robbes and D. Röthlisberger, "Using developer interaction data to compare expertise metrics," in *Proc. 10th Working Conf. Mining Softw. Repositories*, 2013, pp. 297–300.

[19] P. Runeson and M. Hst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Softw. Eng.*, vol. 14, no. 2, pp. 131–164, 2009.

[20] M. Linares-Vasquez, K. Hossen, H. Dang, H. Kagdi, M. Gethers, and D. Poshyvanyk, "Triaging incoming change requests: Bug or commit history, or code authorship?" in *Proc. 28th IEEE Int. Conf. Softw. Maintenance*, 2012, pp. 451–460.

[21] M. Gethers, B. Dit, H. Kagdi, and D. Poshyvanyk, "Integrated impact analysis for managing software changes," in *Proc. Int. Conf. Softw. Eng.*, 2012, pp. 430–440.

[22] J. Czerwonka, N. Nagappan, W. Schulte, and B. Murphy, "CODEMINE: Building a software development data analytics platform at Microsoft," *IEEE Software*, vol. 30, no. 4, pp. 64–71, Jul./Aug. 2013.

[23] C. Bird, T. Carnahan, and M. Greiler. (2015, Feb.). "Lessons learned from deploying a code review analytics platform," Microsoft Research. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=241497, Tech. Rep. MSR-TR-2015-22
[24] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in *Proc. 28th ACM Int. Conf. Softw. Eng.*, 2006, pp. 361–370.
[25] X. Xia, D. Lo, X. Wang, and B. Zhou, "Accurate developer recommendation for bug resolution," in *Proc. 20th Working Conf. Reverse Eng.*, Oct. 2013, pp. 72–81.
[26] T. Fritz, J. Ou, G. C. Murphy, and E. Murphy-Hill, "A degree-of-knowledge model to capture source code familiarity," in *Proc. 32Nd ACM/IEEE Int. Conf. Softw. Eng. - Vol. 1*, 2010, pp. 385–394.
[27] G. Gousios, M. Pinzger, and A. van Deursen, "An exploratory study of the pull-based software development model," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 345–355.
[28] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu, "Don't touch my code! Examining the effects of ownership on software quality," in *Proc. 8th Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng.*, 2011, pp. 4–14.
[29] C. Yu, L. Lakshmanan, and S. Amer-Yahia, "It takes variety to make a world: Diversification in recommender systems," in *Proc. 12th Int. Conf. Extending Database Technol.: Adv. Database Technol.*, 2009, pp. 368–378.
[30] S. Kollanus and J. Koskinen, "Survey of software inspection research," *The Open Softw. Eng. J.*, vol. 3, pp. 15–34, 2009.
[31] J. Cohen, *Best Kept Secrets of Peer Code Review*. Austin, TX, USA: Smart Bear Inc, 2006.
[32] M. Wood, M. Roper, A. Brooks, and J. Miller, "Comparing and combining software defect detection techniques: A replicated empirical study," *SIGSOFT Softw. Eng. Notes*, vol. 22, no. 6, pp. 262–277, Nov. 1997.
[33] A. Porter, H. Siy, A. Mockus, and L. Votta, "Understanding the sources of variation in software inspections," *ACM Trans. Softw. Eng. Methodol.*, vol. 7, no. 1, pp. 41–79, Jan. 1998.
[34] G. Jeong, S. Kim, T. Zimmermann, and K. Yi, "Improving code review by predicting reviewers and acceptance of patches," Research on Software Analysis for Error-free Computing Center, Seoul National University, Seoul, Korea, Tech. Rep. ROSAEC MEMO 2009-006, Sep. 2009.
[35] R. Morales, S. McIntosh, and F. Khomh, "Do code review practices impact design quality? A case study of the qt, vtk, and itk projects," in *Proc. 22nd Int. Conf. Softw. Anal., Evol. Reeng.*, 2015, pp. 171–180.
[36] M. Beller, A. Bacchelli, A. Zaidman, and E. Juergens, "Modern code reviews in open-source projects: Which problems do they fix?" in *Proc. 11th Working Conf. Mining Softw. Repositories*, 2014, pp. 202–211.
[37] C. Kemerer and M. Paulk, "The impact of design and code reviews on software quality: An empirical study based on psp data," *IEEE Trans. Softw. Eng.*, vol. 35, no. 4, pp. 534–550, Jul./Aug. 2009.
[38] M. Kim and D. Notkin, "Discovering and representing systematic code changes," in *Proc. 31st Int. Conf. Softw. Eng.*, 2009, pp. 309–319.
[39] T. Zhang, M. Song, and M. Kim, "Critics: An interactive code review tool for searching and inspecting systematic changes," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2014, pp. 755–758.
[40] L.-T. Cheng, C. R. de Souza, S. Hupfer, J. Patterson, and S. Ross, "Building collaboration into ides," *Queue*, vol. 1, no. 9, pp. 40–50, Dec. 2003.
[41] M. Bernhart, A. Mauczka, and T. Grechenig, "Adopting code reviews for agile software development," in *Proc. Agile Conf.*, Aug. 2010, pp. 44–47.
[42] (2015). Mylyn reviews [Online]. Available: https://projects.eclipse.org/projects/mylyn.reviews
[43] (2015). Gerrit [Online]. Available: https://code.google.com/p/gerrit/
[44] L. Milanesio, *Learning Gerrit Code Review*. Packt Publishing Ltd, Livery Place 35 Livery Street Birmingham B3 2PB, UK, 2013.
[45] (2015). Phabricator [Online]. Available: http://phabricator.org/
[46] (2015). Google mondrian: Web-based code review and storage [Online]. Available: http://www.niallkennedy.com/blog/2006/11/google-mondrian.html
[47] M. Mukadam, C. Bird, and P. C. Rigby, "Gerrit software code review data from Android," in *Proc. Int. Working Conf. Mining Softw. Repositories*, 2013.
[48] J. M. Gonzlez-Barahona, D. Izquierdo-Cortazar, G. Robles, and A. del Castillo, "Analyzing gerrit code review parameters with bicho," *ECEASST*, vol. 65, pp. 1–1, 2014.
[49] G. Jeong, S. Kim, and T. Zimmermann, "Improving bug triage with bug tossing graphs," in *Proc. 7th Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng.*, 2009, pp. 111–120.
[50] J. Anvik and G. Murphy, "Determining implementation expertise from bug reports," in *Proc. 4th Int. Workshop Mining Softw. Repositories*, 2007, pp. 2–2.
[51] D. Cubranic, "Automatic bug triage using text categorization," in *Proc. 16th Int. Conf. Softw. Eng. Knowl. Eng*, 2004, pp. 92–97.
[52] O. Baysal, M. Godfrey, and R. Cohen, "A bug you like: A framework for automated assignment of bugs," in *Proc. IEEE 17th Int. Conf. Program Comprehension*, May 2009, pp. 297–298.
[53] J. Anvik, "Automating bug report assignment," in *Proc. 28th Int. Conf. Softw. Eng.*, 2006, pp. 937–940.
[54] J. Anvik, and G. C. Murphy, "Reducing the effort of bug report triage: Recommenders for development-oriented decisions," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 3, pp. 10:1–10:35, Aug. 2011.
[55] M. B. Zanjani, H. Kagdi, and C. Bird, "Using developer-interaction trails to triage change requests," in *Proc. IEEE 12th Work. Conf. Mining Softw. Repositories*, 2015, pp. 88–98.

Motahareh Bahrami Zanjani received the BSc and MSc degrees from Islamic Azad University, Iran in 2006 and 2010, respectively. She is currently working toward the PhD degree and is a member of the Software Engineering Research Laboratory (SERL) in the Department of Electrical Engineering and Computer Science at Wichita State University. Her research interests are in software maintenance and evolution. She is a student member of the IEEE.

Huzefa Kagdi received the PhD degree in computer science from Kent State University. He is an assistant professor in the Department of Electrical Engineering and Computer Science at Wichita State University. His research interests are in software engineering with emphasis on software maintenance and evolution, empirical software engineering, program comprehension, and software analytics. He is a member of the IEEE.

Christian Bird received the bachelor's degree from Brigham Young University and the PhD degree from U.C. Davis. He is a researcher in the Empirical Software Engineering group at Microsoft Research. He focuses on using qualitative and quantitative methods to both understand and help software teams. He is a member of the IEEE.