Who Should Review My Code?

A File Location-Based Code-Reviewer Recommendation Approach for Modern Code Review

Patanamon Thongtanunam*, Chakkrit Tantithamthavorn*, Raula Gaikovina Kula†,
Norihiro Yoshida‡, Hajimu Iida*, Ken-ichi Matsumoto*
*Nara Institute of Science and Technology, †Osaka University, ‡Nagoya University, Japan
{patanamon-t, chakkrit-t, matumoto}@is.naist.jp, iida@itc.naist.jp, raula-k@ist.osaka-u.ac.jp, yoshida@ertl.jp

***Abstract*—Software code review is an inspection of a code change by an independent third-party developer in order to identify and fix defects before an integration. Effectively performing code review can improve the overall software quality. In recent years, Modern Code Review (MCR), a lightweight and tool-based code inspection, has been widely adopted in both proprietary and open-source software systems. Finding appropriate code-reviewers in MCR is a necessary step of reviewing a code change. However, little research is known the difficulty of finding code-reviewers in a distributed software development and its impact on reviewing time. In this paper, we investigate the impact of reviews with code-reviewer assignment problem has on reviewing time. We find that reviews with code-reviewer assignment problem take 12 days longer to approve a code change. To help developers find appropriate code-reviewers, we propose REVFINDER, a file location-based code-reviewer recommendation approach. We leverage a similarity of previously reviewed file path to recommend an appropriate code-reviewer. The intuition is that files that are located in similar file paths would be managed and reviewed by similar experienced code-reviewers. Through an empirical evaluation on a case study of 42,045 reviews of Android Open Source Project (AOSP), OpenStack, Qt and LibreOffice projects, we find that REVFINDER accurately recommended 79% of reviews with a top 10 recommendation. REVFINDER also correctly recommended the code-reviewers with a median rank of 4. The overall ranking of REVFINDER is 3 times better than that of a baseline approach. We believe that REVFINDER could be applied to MCR in order to help developers find appropriate code-reviewers and speed up the overall code review process.**

***Keywords*—Distributed Software Development, Modern Code Review, Code-Reviewer Recommendation**

I. INTRODUCTION

Software code review has been an engineering best practice for over 35 years [1]. It is an inspection of a code change by an independent third-party developer to identify and fix defects before integrating a code change into a system. While a traditional code review, a formal code review involving in-person meetings, has shown to improve the overall quality of software product [2–4], however, the traditional practice is limited in the adoption to the globally-distributed software development [5].

Recently, Modern Code Review (MCR) [6], an informal, lightweight and tool-based code review methodology, has emerged as a widely used tool in both industrial software and open-source software. Generally, when a code change, *i.e., patch*, is submitted for review, the author will invite a set of code-reviewers to review the code change. Then, the code-reviewers will discuss the change and suggest fixes. The code change will be integrated to the main version control system when one or more code-reviewers approve the change. Rigby *et al.* [7] find that code reviews are expensive because they require code-reviewers to read, understand, and critique a code change. To effectively assess a code change, an author should find appropriate code-reviewers who have a deep understanding of the related source code to well examine code changes and find defects [4]. As a huge amount of code changes must be reviewed before the integration, finding appropriate code-reviewers to every piece of code changes can be time-consuming and labor-intensive for developers [8].

In this paper, we first set out to better understand how do reviews with code-reviewer assignment problem impact reviewing time. In particular, we investigate (1) what is the proportion of reviews with code-reviewer assignment problem; (2) how do reviews with code-reviewer assignment problem impact reviewing time; and (3) does a code-reviewer recommendation tool is necessary in distributed software development. We manually examine 7,597 comments from 1,461 representative review samples of four open-source software systems to identify the reviews with a discussion of code-reviewer assignment. Our results show that 4%-30% of reviews have code-reviewer assignment problem. These reviews significantly take 12 days longer to approve a code change. Our findings also show that a code-reviewer recommendation tool is necessary in distributed software developerment to speed up a code review process.

To help developers find appropriate code-reviewers, we propose REVFINDER, a file location-based code-reviewer recommendation approach. We leverage a similarity of previously reviewed file path to recommend an appropriate code-reviewer. The intuition is that *files that are located in similar file paths would be managed and reviewed by similar experienced code-reviewers.* In order to evaluate REVFINDER, we perform a case study on 42,045 reviews of four open-source software systems i.e., Android Open Source Project (AOSP), OpenStack, Qt and LibreOffice. The results show that REVFINDER correctly recommended 79% of reviews with a top 10 recommendation. REVFINDER is 4 times more accurate than a baseline approach, indicating that leveraging a similarity of previously reviewed file path can accurately recommend code-reviewers. REVFINDER also recommended the correct code-reviewers with a median rank of 4. The overall ranking of REVFINDER is 3 times better than that of the baseline approach, indicating that REVFINDER provides a better ranking of recommended code-reviewers. Therefore, we believe that REVFINDER can help developers find appropriate code-reviewers and speed up the overall code review process.

978-1-4799-8469-5/15 © 2015 IEEE

SANER 2015, Montréal, Canada

The main contributions of this paper are:

- An exploratory study on the impact of code-reviewer assignment on reviewing time.
- REVFINDER, a file location-based code-reviewers recommendation approach, with promising evaluation results to automatically suggest appropriate code-reviewers for MCR.
- A rich data set of reviews data in order to encourage future research in the area of code-reviewer recommendation.[1]

The remainder of the paper is organized as follows. Section II describes a background of MCR process and related work. Section III presents an exploratory study of the impact of code-reviewer assignment on reviewing time. Section IV presents our proposed approach, REVFINDER. Section V describes an empirical evaluation of the approach. Section VI presents the results of our empirical evaluation. Section VII discusses the performance and applicability of REVFINDER, and addresses the threats to validity. Finally, Section VIII draws our conclusion and future work.

II. BACKGROUND AND RELATED WORK

A. Modern Code Review

Code review is the manual assessment of source code by humans, mainly intended to identify defects and quality problems [9]. However, the traditional code review practice is limited in the adoption to the globally-distributed software development [5]. In recent years, Modern Code Review (MCR) has been developed as a tool-based code review system which is less formal than the traditional one. MCR becomes popular and widely used in both proprietary software (e.g., Google, Cisco, Microsoft) and open-source software (e.g., Android, Qt, LibreOffice) [6]. Below, we briefly describe the Gerrit-based code review system, which is a prominent tool and widely used in previous studies [10–13].

To illustrate the Gerrit-based code review system, we use an example of Android reviews ID 18767[2] (see Figure 1). The developers' information are blinded for privacy reasons. In general, there are four steps as following:

1) An author (`Smith`) creates a change and submits it for review.
2) The author (`Smith`) invites a set of reviewers (i.e., Code-reviewers and Verifiers) to review the patch.
3) A code-reviewer (`Alex`) will discuss the change and suggest fixes. A verifier (`John`) will execute tests to ensure that: (1) patch truly fix the defect or add the feature that the authors claim to, and (2) do not cause regression of system behavior. The author (`Smith`) needs to revise and re-submit the change to address the suggestion of the reviewers (`Alex` and `John`).
4) The change will be integrated to the main repository when it receives a code-review score of +2 (Approved) from a code-reviewer and a verified score of +1 (Verified) from a verifier. Then, the review will be marked as *"Merged."* Otherwise, the change will be automatically rejected if it receives a code-review score of -2 (Rejected) and the review will be marked as *"Abandoned."*

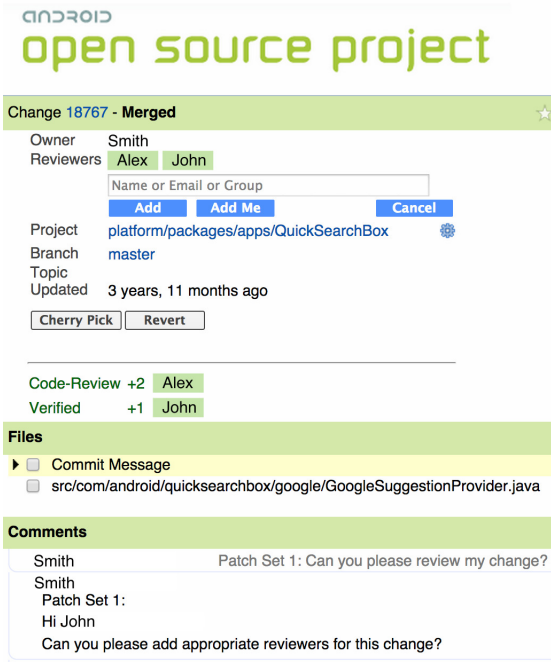

Fig. 1: An example of Gerrit code reviews in Android Open Source Project.

From the example in Figure 1, we observe that finding appropriate code-reviewers is a tedious task for developers. The author (`Smith`) has a code-reviewer assignment problem since he cannot find code-reviewers to review his change. To find code-reviewers, the author (`Smith`) asks other developers in the discussion: *"Can you please add appropriate reviewers for this change?"* Finding an appropriate code-reviewer can increase the reviewing time and decrease the effectiveness of MCR process. Therefore, an automatic code-reviewer recommendation tool would help developers reduce their time and effort.

B. Related Work

We discuss the related work with respect to code review in distributed software development and expert recommendation.

Software Code Review. Understanding software code review practices receive much attention in the last few years. Rigby *et al.* [14] empirically investigate the activity, participation, review interval and quality of code review process in an open-source software. They observe that if a code change is not reviewed immediately, it will not likely be reviewed. Weigerber *et al.* [15] examine the characteristics of patch acceptance and found that smaller code changes are more likely to be accepted. They also observed that some code changes need more than two weeks until being merged. Rigby and Bird [11] also observe that 50% of reviews have reviewing time almost 30 days. A recent study by Tsay *et al.* [16] also find that some code changes are awaiting to be merged for 2 months.

To better understand what influences code review interval, Jiang *et al.* [17] empirically investigate the characteristics of

[1] http://github.com/patanamon/revfinder
[2] https://android-review.googlesource.com/#/c/18767/

TABLE I: A statistical summary of datasets for each studied system.

	Android	OpenStack	Qt	LibreOffice
Studied Period	10/2008 - 01/2012	07/2011 - 05/2012	05/2011 - 05/2012	03/2012 - 06/2014
# Selected Reviews	5,126	6,586	23,810	6,523
# Code-reviewers	94	82	202	63
# Files	26,840	16,953	78,401	35,273
Avg. Code-reviewers per Review	1.06	1.44	1.07	1.01
Avg. Files per Review	8.26	6.04	10.64	11.14

accepted code changes and its reviewing time on a Linux case study. They found that reviewing time is impacted by submission time, the number of affected subsystem, the number of code-reviewers and developer's experience. Bosu *et al.* [18] also found that the reviewing time of a code change submitted by core developers is shorter than peripheral developers. Pinzger *et al.* [19] also found that the reviewing time in Github is influenced by the developer's contribution history, the size of the project, and its test coverage, and the project's openness to external contriubtions. However, none of these studies addresses the difficulty of finding appropriate code-reviewers nor do they investigate the impact of reviews with code-reviewer assignment problem on reviewing time. In this paper, we conduct an exploratory study to investigate the impact of reviews with code-reviewer assignment problem on reviewing time.

Expert Recommendation. Finding relevant expertise is a critical need in collaborative software engineering, particularly in geographically distributed developments [20]. We briefly address two closely related research areas *i.e.,* expert recommendation for bug fixing process and MCR process. To recommend experts for bug fixing process, Anvik *et al.* [21] propose an approach based on machine learning techniques to recommend developers to fix a new bug report. Shokripour *et al.* [22] propose an approach to recommend developers based on information in bug report and history of fixed files. Xia *et al.* [23] propose a developer recommendation using bug report and developer information. Surian *et al.* [24] propose a developer recommendation using developers' collaboration network. In a recent study, Tian *et al.* [25] propose an expert recommendation system for Question & Answer community using topic modeling and collaborative voting scores. In contrast, we focus on the code review process which has limited textual information in the code review systems.

To recommend experts for code review process, Jeong *et al.* [26] extract features from patch information and build prediction model using Baysian network. Yu *et al.* [27] use social relationship among developers to recommend code-reviewers for Github systems. Balachandran *et al.* [28] use a modification history in line-by-line of source code to recommend code-reviewers for industrial environment of VMware, called REVIEWBOT. Our prior study has shown that the performance of the REVIEWBOT is limited in other software systems with no or little modification history in line-by-line of source code [29]. Therefore, code-reviewer recommendation approaches can be further improved. In this paper, we use a similarity of previously reviewed file pathto recommend code-reviewers.

III. AN EXPLORATORY STUDY OF THE IMPACT OF CODE-REVIEWER ASSIGNMENT ON REVIEWING TIME

In this section, we report the results of our exploratory study on the difficulty of finding code-reviewers in a distributed software development and its impact on reviewing time.

(RQ1) How do reviews with code-reviewer assignment problem impact reviewing time?

Motivation. Little is known about the difficulty of finding code-reviewers in a distributed software development and its impact on reviewing time. We suspect that reviews with code-reviewer assignment problem are likely to require more time and discussion in order to identify an appropriate code-reviewer. Hence, we set out to empirically investigate the impact of reviews with code-reviewer assignment problem has on reviewing time as compared to the reviews without code-reviewer assignment problem. In particular, we investigate: (1) what is the proportion of reviews with code-reviewer assignment problem; (2) how do reviews with code-reviewer assignment problem impact reviewing time; and (3) does a code-reviewer recommendation tool is necessary in distributed software development.

Approach. To address RQ1, we first select a representative sample of reviews. We then manually examine the discussion to identify the reviews with code-reviewer assignment problem. We then calculate the reviewing time of the review samples. The results are then analyzed and discussed. We describe how we perform each step in particular below.

(Step 1) Data Collection: We use the review data of Android Open Source Project (AOSP), OpenStack and Qt projects provided by Hamasaki *et al.* [30]. We also expand the dataset to include the review data of LibreOffice project using the same collection technique [30]. After collecting reviews, we select them in the following manner; (1) Reviews are marked as *"Merged"* or *"Abandoned"*; and (2) Reviews contain at least one code change to conform the purpose of code review practice [13]. Table I shows the statistical summary for each studied systems.

(Step 2) Representative Sample Selection: To identify the proportion of reviews with code-reviewer assignment problem, we select a representative sample of reviews for manual analysis, since the full set of reviews is too large to study entirely. To obtain proportion estimates that are within 5% bounds of the actual proportion with a 95% confidence level, we use a sample size $s = \frac{z^2 p(1-p)}{0.05^2}$, where p is the proportion that we want to estimate and $z = 1.96$. Since we did not know the proportion in advance, we use p = 0.5. We further correct

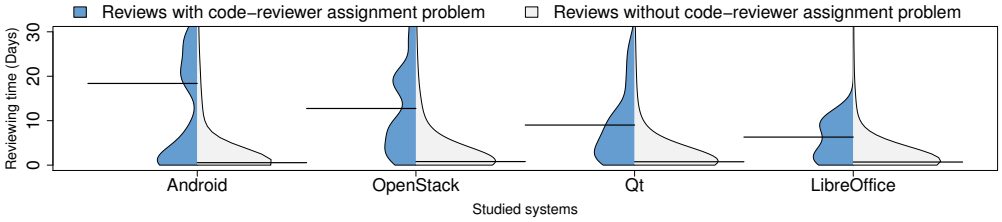

Fig. 2: A comparison of the code reviewing time (days) of reviews with and without code-reviewer assignment problem. The horizontal lines indicate the average (median) review time (days).

for the finite population of reviews P using $ss = \frac{s}{1+\frac{s-1}{P}}$ to obtain our sample for manual analysis. Table II shows the numbers of samples for manual analysis of 357 Android reviews, 363 OpenStack reviews, 378 Qt reviews, and 363 LibreOffice reviews. In total, we manually examine 7,597 comments of 1,461 review samples.

(Step 3) Review Classification: To determine whether a review has a code-reviewer assignment problem, we manually classify the reviews by examining through each of their comments. Since the comments are written in natural language, it is very difficult to automatically analyze them. We define that reviews with code-reviewer assignment problem are the reviews that their discussions address an issue about who should review these code changes.

(Step 4) Data Analysis: Once we manually examine the review samples, we calculate the reviewing time of the review samples. Reviewing time is the time difference from the time that a code change has been submitted to the time that the code changes are approved or abandoned. We then compare the distributions of the reviewing time between reviews with and without code-reviewer assignment problem using beanplots [31]. Beanplots are boxplots in which the vertical curves summarize the distributions of different data set. The horizontal lines indicate the average (median) reviewing time (days). We use a Mann-Whitney's U test ($\alpha = 0.05$), which is a non-parametric test, to statistically determine the difference of the reviewing time distributions. Since reviewing time can be influenced by patch size [15, 19], we divide the reviewing time by the patch size before performing a statistical test.

Results. 4%-30% of reviews have code-reviewer assignment problem. The percentage of reviews with code-reviewer assignment problem is shown in Table II. From our empirical investigation, we found that, for Android, OpenStack, Qt, and LibreOffice, 10%, 5%, 30%, and 4% of reviews have code-reviewer assignment problem, respectively. We observe that Qt has the highest proportion of reviews with code-reviewer assignment problem. It may in part be due to the size of the community and software system (i.e., the amount of reviews, code-reviewers, and files). This indicates that the larger the system is, the more difficult of finding appropriate code-reviewers it is.

On average, reviews with code-reviewer assignment problem require 12 days longer to approve code changes. Figure 2 shows that reviews with code-reviewer assignment problem require 18, 9, 13, and 6 days to make an integration decision of code changes for Android, OpenStack, Qt, and LibreOffice, respectively. In contrast, reviews without code-reviewer assignment problem can be integrated within one day. Mann-Whitney U tests confirm that the differences are statistically significant (p-value < 0.001 for Android, OpenStack, and Qt, and p-value < 0.01 for LibreOffice). This finding indicates that most of reviews with code-reviewer assignment problem could slow down the code review process of distributed software development.

TABLE II: The numbers of statistical representative samples for each studied projects and the percentage of reviews with code-reviewer assignment problem with a 95% confidence level and a confidence interval of ± 5%.

	Android	OpenStack	Qt	LibreOffice
# Review	357	363	378	363
Percentage	10%	5%	30%	4%

A code-reviewer recommendation tool is necessary in distributed software development to speed up the code review process. During our manual examination, we find that finding a code-reviewer is truly a necessary step of MCR process. For example, a Qt developer said: "*You might want to add some approvers to the reviewers list if you want it reviewed/approved.*"[3] Additionally, developers often ask a question of finding appropriate code-reviewers. For example, a Qt developer said that; "*Feel free to add reviewers, I am not sure who needs to review this...*"[4] One of the Android developers also said; "*Can you please add appropriate reviewers for this change?*"[5] Therefore, finding an appropriate code-reviewer to review a code change is a tedious task for developers and posses a problem in distributed software development in recent years. Moreover, a Qt developer suggested an author to add code-reviewers to speed up the code review process: "*for the future, it speeds things up often if you add reviewers for your changes :)*"[6]

4%-30% of reviews have code-reviewer assignment problem. These reviews significantly take 12 days longer to approve a code change. A code-reviewer recommendation tool is necessary in distributed software development to speed up a code review process.

[3]Qt-16803 https://codereview.qt-project.org/#/c/16803
[4]Qt-40477 https://codereview.qt-project.org/#/c/40477
[5]AOSP-18767 https://android-review.googlesource.com/#/c/18767/
[6]Qt-14251 https://codereview.qt-project.org/#/c/14251

IV. REVFINDER: A FILE LOCATION-BASED CODE-REVIEWER RECOMMENDATION APPROACH

A. An Overview of REVFINDER

REVFINDER is a combination of recommended code-reviewers from the Code-Reviewers Ranking Algorithm. REVFINDER aims to recommend code-reviewers who have previously reviewed similar functionality. Therefore, we leverage a similarity of previously reviewed file path to recommend code-reviewers. The intuition is that *files that are located in similar file paths would be managed and reviewed by similar experienced code-reviewers.* REVFINDER has two main parts i.e., the Code-Reviewers Ranking Algorithm and the Combination Technique.

The Code-Reviewers Ranking Algorithm computes code-reviewer scores using a similarity of previously reviewed file path. To illustrate, we use Figure 3 as a calculation example of the algorithm. Given a new review R_3; and two previously closed reviews R_1 and R_2, the algorithm first calculates a review similarity score for each of previous reviews (R_1,R_2) by comparing file paths with the new review R_3. Therefore, we will have two review similarity scores of (R_3, R_1) and (R_3, R_2). To compute a review similarity score, we use a state-of-the-art string comparison technique [32], which is successfully used in computational biology. In this example, it is obvious that the file path of R_3 and R_2 share some common keywords (`video`, `src`) more than a pair of R_3 and R_1. We presume that the review similarity score of (R_3, R_1) is 0.1 and that of (R_3, R_2) is 0.5. Then, these review similarity scores are propagated to each code-reviewer who has involved in i.e., Code-Reviewer A earns review similarity scores of $0.5 + 0.1$ and Code-Reviewer B earns a review similarity score of 0.1. Finally, the algorithm will produce a list of code-reviewers along with their scores. Since there are many well-known variants of string comparison techniques [32], REVFINDER combines the different lists of code-reviewers into a unified list of code-reviewers. By combining, the truly-relevant code-reviewers are likely to "bubble up" to the top of the combined list, providing code-reviewers with fewer false positive matches to recommend.

Below, we explain the calculation of Code-Reviewers Ranking Algorithm, String Comparison Techniques and the Combination Technique.

B. The Code-Reviewers Ranking Algorithm

The pseudo-code of the Code-Reviewers Ranking Algorithm is shown in Algorithm 1. It takes as input a new review (R_n) and produces a list of code-reviewer candidates (C) with code-reviewer scores. The algorithm begins with retrieving the reviews before R_n as pastReviews and sorts them by their creation date in reverse chronological order (Lines 7 and 8). We note that the pastReviews are previously closed reviews that are marked as *"Merged"* or *"Abandoned"* and must be created before R_n. The algorithm calculates a review similarity score between each of pastReviews and the new review (R_n). Then, the review similarity scores are propagated to code-reviewers who involved in (Lines 9 to 24). For each review (R_p) of the pastReviews, the review similarity score (Score_{R_p}) is an average of file path similarity value of every file path in R_n

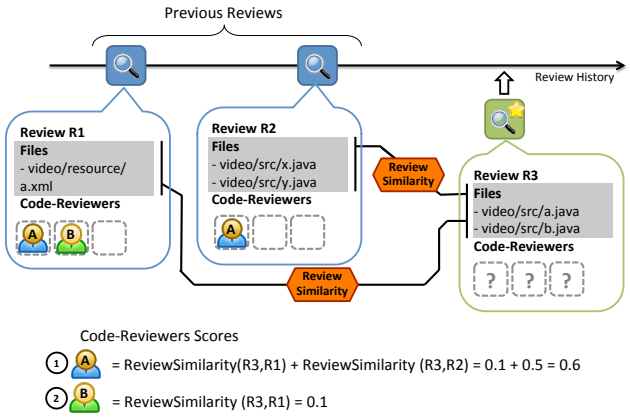

Fig. 3: A calculation example of the Code-Reviewers Ranking Algorithm.

Algorithm 1 The Code-Reviewers Ranking Algorithm

```
1:  Code-ReviewersRankingAlgorithm
2:  Input:
3:  R_n : A new review
4:  Output:
5:  C : A list of code-reviewer candidates
6:  Method:
7:  pastReviews ← A list of previously closed reviews
8:  pastReviews ← order(pastReviews).by(createdDate)
9:  for Review R_p ∈ pastReviews do
10:     Files_n ← getFiles(R_n)
11:     Files_p ← getFiles(R_p)
12:     # Compute review similarity score between R_n and R_p
13:     Score_Rp ← 0
14:     for f_n ∈ Files_n do
15:        for f_p ∈ Files_p do
16:           Score_Rp ← Score_Rp + filePathSimilarity(f_n, f_p)
17:        end for
18:     end for
19:     Score_Rp ← Score_Rp / (length(Files_n) × length(Files_p))
20:     # Propagate review similarity scores to code-reviewers who
        involved in a previous review R_p
21:     for Code-Reviewer r : getCodeReviewers(R_p) do
22:        C[r].score ← C[r].score + Score_Rp
23:     end for
24: end for
25: return C
```

and R_p using $\text{filePathSimilarity}(f_n, f_p)$ function (Lines 13 to 19). After calculating the review similarity score, every code-reviewer in R_p is added to the list of code-reviewer candidates (C) with their review similarity score (Score_{R_p}) (Lines 21 to 23). If a code-reviewer is already in C, the code-reviewer score will be cumulated with the previous code-reviewer score.

To compute file path similarity value between file f_n and file f_p, the $\text{filePathSimilarity}(f_n, f_p)$ function is calculated as follows:

$$\text{filePathSimilarity}(f_n, f_p) = \frac{\text{StringComparison}(f_n, f_p)}{\max(\text{Length}(f_n), \text{Length}(f_p))} \quad (1)$$

We split file path into components using a slash character as a delimiter. The $\text{StringComparison}(f_n, f_p)$ function compares

TABLE III: A description of file path comparison techniques and examples of calculation. The examples are obtained from the review history of Android for LCP, LCSubstr, and LCSubseq techniques; and Qt for LCS techniques. For each technique, the example files were reviewed by the same code-reviewer.

Functions	Description	Example
Longest Common Prefix (LCP)	Longest consecutive path components that appears in the beginning of both file paths.	f_1 ="**src**/**com**/**android**/**settings**/LocationSettings.java" f_2 = "**src**/**com**/**android**/**settings**/Utils.java" LCP(f_1, f_2) = length([src, com, android, settings]) = 4
Longest Common Suffix (LCS)	Longest consecutive path components that appears in the end of both file paths	f_1 = "src/imports/**undo**/**undo.pro**" f_2 = "tests/auto/**undo**/**undo.pro**" LCS(f_1, f_2) = length([undo, undo.pro]) = 2
Longest Common Substring (LCSubstr)	Longest consecutive path components that appears in both file paths	f_1 = "**res**/**layout**/bluetooth_pin_entry.xml" f_2 = "tests/**res**/**layout**/operator_main.xml" LCSubstr(f_1, f_2) = length([res, layout]) = 2
Longest Common Subsequence (LCSubseq)	Longest path components that appear in both file paths in relative order but not necessarily contiguous	f_1 ="apps/CtsVerifier/**src**/com/**android**/**cts**/verifier/sensors/MagnetometerTestActivity.java" f_2 ="tests/tests/hardware/**src**/**android**/hardware/**cts**/SensorTest.java" LCSubstr(f_1, f_2) = length([src, android, cts]) = 3

file path components of f_n and f_p and returns a number of the common components that appear in both files. Then, the value of $\mathrm{filePathSimilarity}(f_n, f_p)$ is normalized by the maximum length of f_n and f_p i.e., the number of file path components. The details of string comparison techniques will be presented in the next subsection.

C. String Comparison Techniques

To compute file path similarity score ($\mathrm{filePathSimilarity}$), we use four state-of-the-art string comparison techniques [32] i.e., Longest Common Prefix (LCP), Longest Common Suffix (LCS), Longest Common Substring (LCSubstr), and Longest Common Subsequence (LCSubseq). Table III presents the definitions and a calculation example for these techniques. We briefly explain the rationale of these techniques below.

Longest Common Prefix. Files under the same directory would have similar or related functionality [33]. LCP calculates the number of common path components that appears in both file paths from the beginning to the last. This is the most simple and efficient way to compare two strings.

Longest Common Suffix. Files having the same name would have the same functionality [32]. LCS calculates the number of common path components that appears from the end of both file paths. This is a simply reverse calculation of LCP.

Longest Common Substring. Since the file path can represent their functionality [34], the related functionality should be under the same directory structure. However, their root directories or filename might not be the same. LCSubstr calculates the number of path components that appears in both file path consecutively. The advantage of this technique is that the common paths can be appeared at any position of file path.

Longest Common Subsequence. Files under similar directory structure would have similar or related functionality [32]. LCSubseq calculates the number of path components that appears in both file paths which is in the same relative order. The advantage of this technique is that the common paths of this technique are not necessary to be contiguous.

D. Combination Technique

A combination of the results of individual techniques has been successfully shown to improve the performance in the data mining and software engineering domains [35, 36]. Since we used variants of string comparison techniques [32], RevFinder combines the different lists of code-reviewers into a unified list of code-reviewers. Therefore, the truly-relevant code-reviewers are likely to "bubble up" to the top of the combined list, providing code-reviewers with fewer false positive matches to recommend. We use the Borda count [37] as a combination technique. It is a voting technique that simply combine the recommendation lists based on the rank. For each code-reviewer candidate c_k, the Borda count method assigns points based on the rank of c_k in each recommendation list. The candidate with the highest rank will get the highest score. For example, if a recommendation list of R_{LCP} votes candidate c_1 as the first rank and the number of total candidates are M, then c_k would get a score of M. The candidate c_{10} (ranked 10^{th}) would get a score of $M - 10$. Given a set of recommendation lists $R \in \{R_{LCP}, R_{LCS}, R_{LCSubstr}, R_{LCSubseq}\}$, a score for a code-reviewer candidate c_k is defined as follow:

$$\mathrm{Combination}(c_k) = \sum_{R_i \in R} M_i - \mathrm{rank}(c_k|R_i) \qquad (2)$$

, where M_i is the total number of code-reviewer candidates that received a non-zero score in R_i, and $\mathrm{rank}(c_k|R_i)$ is the rank of code-reviewer candidate c_k in R_i. Finally, the code-reviewer recommendations of RevFinder is a list of code-reviewer candidates that are ranked according to their Borda score.

V. Empirical Evaluation

We perform an empirical study to evaluate the effectiveness of RevFinder. First, we describe the goal and the research questions we addressed. Second, we describe the studied systems that we used to evaluate. Third, we present evaluation metrics. Last, we briefly describe a baseline comparison approach.

A. Goal and Research Questions

The *goal* of our empirical study is to evaluate the effectiveness of REVFINDER in terms of accuracy and ranking of the correct code-reviewers. The results of REVFINDER are then compared with REVIEWBOT [28] as a baseline approach since it is the only existing code-reviewer recommendation approach for MCR.

To achieve our goal, we address the following two research questions:

(RQ2) Does REVFINDER accurately recommend code-reviewers?
Motivation: We propose REVFINDER to find appropriate code-reviewers based on a similarity of previously reviewed file path. We aim to evaluate the performance of our approach in terms of accuracy. Better performing approaches are of great value to practitioners since they allow them to take better informed decisions.

(RQ3) Does REVFINDER provide better ranking of recommended code-reviewers?
Motivation: Recommending correct code-reviewers in the top ranks could ease developer as well as avoid interfering unrelated code-reviewers. The higher ranks of the correct code-reviewers that the approach can recommend, the more effective it is [38]. We set out this research question to evaluate the overall performance of our approach in the view of ranking for a recommendation.

B. Studied Systems

To evaluate REVFINDER, we use four open-source software systems, i.e. Android, OpenStack, Qt, and LibreOffice. We choose these systems mainly for three reasons. First, these systems use Gerrit system as a tool-based for their code review process. Second, these systems are large, active, real-world software, which allow us to perform a realistic evaluation of REVFINDER. Third, each systems carefully maintains code review system, which allows us to build our oracle datasets to evaluate REVFINDER. Table I shows a statistical summary of the studied systems. Note that we use the same datasets as in RQ1.

Android Open Source Project (AOSP)[7] is a mobile operating system developed by Google. Qt[8] is a cross-platform application and UI framework developed by Digia Plc. OpenStack[9] is a free and open-source software cloud computing software platform supported by many well-known companies e.g., IBM, VMware, and NEC. LibreOffice[10] is a free and open source office suite.

C. Evaluation Metrics

To evaluate our approach, we use the top-k accuracy and the Mean Reciprocal Rank (MRR). These metrics are commonly used in recommendation systems for software engineering [28, 39, 40]. Since most of reviews have only one code-reviewer (cf. Table I), other evaluation metrics (e.g. Mean Average Precision) that consider all of the correct answer might not be appropriate for this evaluation.

Top-k accuracy calculates the percentage of reviews that an approach can correctly recommend code-reviewers and the total number of reviews. Given a set of reviews R, the top-k accuracy can be calculated using Equation 3. The $\text{isCorrect}(r, \text{Top-}k)$ function returns value of 1 if at least one of top-k code-reviewers actually approve the review r; and returns value of 0 for otherwise. For example, a top-10 accuracy value of 75% indicates that for 75% of the reviews, at least one correct code-reviewer was returned in the top 10 results. Inspired by the previous studies [28, 39, 40], we choose the k value to be 1, 3, 5, and 10.

$$\text{Top-}k \text{ accuracy}(R) = \frac{\sum_{r \in R} \text{isCorrect}(r, \text{Top-}k)}{|R|} \times 100\% \quad (3)$$

Mean Reciprocal Rank (MRR) calculates an average of reciprocal ranks of correct code-reviewers in a recommendation list. Given a set of reviews R, MRR can be calculated using Equation 3. The $\text{rank}(\text{candidates}(r))$ returns value of the first rank of actual code-reviewers in the recommendation list $\text{candidates}(r)$. If there is no actual code-reviewers in the recommendation list, the value of $\frac{1}{\text{rank}(\text{candidates}(r))}$ will be 0. Ideally, an approach that can provide a perfect ranking should achieve a MRR value of 1.

$$\text{MRR} = \frac{1}{|R|} \sum_{r \in R} \frac{1}{\text{rank}(\text{candidates}(r))} \quad (4)$$

D. REVIEWBOT: A Baseline Approach

We re-implement REVIEWBOT [28] as our baseline. REVIEWBOT is a code-reviewer recommendation approach based on the assumption that *"the most appropriate reviewers for a code review are those who previously modified or previously reviewed the sections of code which are included in the current review"* [28, p.932]. Thus, REVIEWBOT finds code-reviewers using a modification history in line-by-line of source code. The calculation of REVIEWBOT can summarized as follows: Given a new review, 1) it computes line change history, a list of past reviews that relate to the same changed lines in the new review. 2) The code-reviewers in line change history will be code-reviewer candidates for the new review. Each candidate receives a point based on her frequency of reviews in line change history. 3) The candidates who recent reviewed and have the highest scores will be recommended as appropriate code-reviewers. To conserve space, a full description of REVIEWBOT is provided in [28].

VI. RESULTS

In this section, we present the results of our empirical evaluation with respect to our two research questions. For each research question, we present its approach, and results.

[7] https://source.android.com/
[8] http://qt-project.org/
[9] http://www.OpenStack.org/
[10] http://www.libreoffice.org/

TABLE IV: The results of top-k accuracy of our approach RevFinder and a baseline ReviewBot for each studied system. The results show that RevFinder outperforms ReviewBot.

System	RevFinder				ReviewBot			
	Top-1	Top-3	Top-5	Top-10	Top-1	Top-3	Top-5	Top-10
Android	46 %	71 %	79 %	**86** %	21 %	29 %	29 %	29 %
OpenStack	38 %	66 %	77 %	**87** %	23 %	35 %	39 %	41 %
Qt	20 %	34 %	41 %	**69** %	19 %	26 %	27 %	28 %
LibreOfiice	24 %	47 %	59 %	**74** %	6 %	9 %	9%	10 %

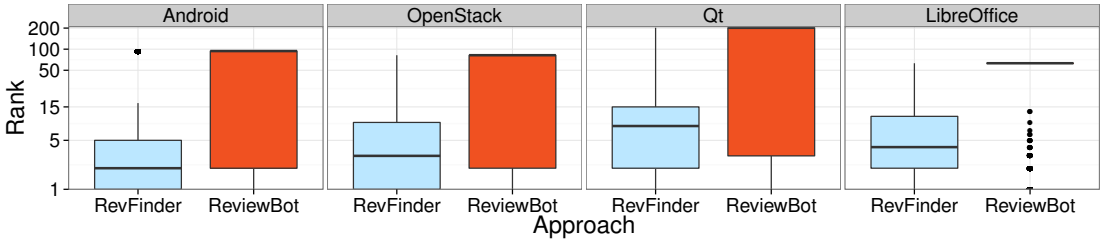

Fig. 4: A rank distribution of the first correct code-reviewers recommended by RevFinder and ReviewBot. The results show that RevFinder provide a better ranking of recommended code-reviewers.

(RQ2) Does* RevFinder *accurately recommend code-reviewers?

Approach. Since RevFinder leverages a review history to recommend code-reviewers, we perform an experiment based on realistic scenario. To address RQ2, for each studied system, we execute RevFinder for every reviews in chronological order to obtain the lists for recommended code-reviewers. To evaluate how accurately the RevFinder can correctly recommended code-reviewers, we compute the top-k accuracy for each studied system. We also compare the results of our approach with ReviewBot.

Result. On average, for 79% of reviews, RevFinder correctly recommended code-reviewers with a top-10 recommendation. Table IV presents the results of top-1, top-3, top-5 and top-10 accuracy of RevFinder and ReviewBot for each studied system. The RevFinder achieves the top-10 accuracy of 86%, 87%, 69%, and 74% for Android, OpenStack, Qt, and LibreOffice, respectively. This indicates that leveraging a similarity of previously reviewed file path can accurately recommend code-reviewers.

On average, RevFinder is 4 times more accurate than ReviewBot. Table IV shows that, for every studied system, RevFinder achieves higher top-k accuracy than ReviewBot. The top-10 accuracy values of RevFinder are 2.9, 2.1, 2.5, and 7.4 times higher than that of ReviewBot for Android, OpenStack, Qt, and LibreOffice, respectively. We also find similar results for other top-k accuracy metrics, indicating that RevFinder considerably outperforms ReviewBot.

> RevFinder *correctly recommended 79% of reviews with a top-10 recommendation.* RevFinder *is 4 times more accurate than* ReviewBot. *This indicates that leveraging a similarity of previously reviewed file path can accurately recommend code-reviewers.*

TABLE V: The results of Mean Reciprocal Rank (MRR) of our approach RevFinder and a baseline ReviewBot. Ideally, a MRR value of 1 indicates a perfect recommendation approach.

Approach	Android	OpenStack	Qt	LibreOffice
RevFinder	**0.60**	**0.55**	**0.31**	**0.40**
ReviewBot	0.25	0.30	0.22	0.07

(RQ3) Does* RevFinder *provide better ranking of recommended code-reviewers?

Approach. To address RQ3, we present the distribution of the ranking of correct code-reviewers. We also use the Mean Reciprocal Rank (MRR) to represent the overall ranking performance of RevFinder. The results are then compared with ReviewBot.

Result. RevFinder recommended the correct code-reviewers with a median rank of 4. Figure 4 shows that the correct ranks of code-reviewers of RevFinder is lower than that of ReviewBot for all studied systems. The median correct ranks of RevFinder are 2, 3, 8, and 4 for Android, OpenStack, Qt, and LibreOffice, respectively. In contrast, the median correct ranks of ReviewBot are 94, 82, 202, 63 for Android, OpenStack, Qt, and LibreOffice, respectively. This indicates that RevFinder provides a higher chance of inviting a correct code-reviewer and a less chance of interfering unrelated code-reviewers.

The overall ranking of RevFinder is 3 times better than that of ReviewBot. Table V shows the MRR values of RevFinder and ReviewBot for each studied system. For Android, OpenStack, Qt, and LibreOffice, the MRR values of RevFinder are 2.4, 1.8, 1.4, and 5.7 times better than that of ReviewBot, respectively. This indicates that RevFinder can correctly recommend the first correct code-reviewers at lower rank than ReviewBot does.

REVFINDER *recommended the correct code-reviewers with a median rank of 4. The code-reviewers ranking of* REVFINDER *is 3 times better than that of* REVIEWBOT*, indicating that* REVFINDER *provides a better ranking of correct code-reviewers.*

VII. DISCUSSION

We discuss the performance and applicability of REVFINDER. We also discuss the threats to validity of our study.

Performance: *Why does* REVFINDER *outperform* REVIEWBOT*?*

The results of our empirical evaluation show that the proposed approach, REVFINDER outperforms the baseline approach, REVIEWBOT. The difference between REVFINDER and REVIEWBOT is the difference in the granularity of code review history. REVFINDER uses the code review history at file path-level, while REVIEWBOT uses the code review history at the line-level of source code. Intuitively, finding code-reviewers who have examined the exact same lines seems to be the best choice for those projects with high frequent changes of source code. However, it is not often that files are frequently change at the same lines [41]. Besides, MCR is relatively new, the performance of REVIEWBOT would be limited due to a small amount of review history. To better understand why does REVFINDER outperform REVIEWBOT, we investigate the frequency of review history at the line level and file level of granularity. We observed that 70% - 90% of lines of code are changed only once, indicating that in a code review system has a lack of the line-level history. Therefore, the performance of REVIEWBOT is limited.

Applicability: *Can* REVFINDER *effectively help developers find code-reviewers?*

In RQ1, the results of our exploratory study show that reviews with code-reviewer assignment problem required more times to integrate a code change. To confirm how effectively REVFINDER help developers, we execute REVFINDER for the reviews with code-reviewer assignment problem of the representative samples. We found that, on average, REVFINDER can correctly recommend code-reviewers for 80% of these reviews with a top 10 recommendation. This result indicates that if a developer cannot find an appropriate code-reviewer for a new change, REVFINDER could accurately recommend appropriate code-reviewers at hand. Therefore, we believe that REVFINDER can help developers find appropriate code-reviewers and speed up the overall code review process.

Threats to Validity: *We discuss potential threats to validity of our work as follows:*

Internal Validity: The reviews classification process in RQ1 involves manual examination. The classification process was conducted by the authors who are not involved in the code review process of the studied systems. The results of manual classification by a domain expert might be different.

External Validity: Our empirical results are limited to four datasets i.e., Android, OpenStack, Qt, and LibreOffice. However, we cannot claim that the same results would be achieved with other systems. Our future work will focus on an evaluation in other studied systems with larger number of code-reviewers to better generalize the results of our approach.

Construct Validity: The first threat involves a lack of code-reviewer retirement information. It is possible that code-reviewers are retired or no longer involve the code review system. Therefore, the performance of our approach might be affected by retired code-reviewers activities. Another threat involves the workload of code-reviewers. It is possible that code-reviewers would be burdened with a huge number of assigned reviews. Therefore, considering workload balancing would reduce tasks of these potential code-reviewers and the number of awaiting reviews.

VIII. CONCLUSION AND FUTURE WORK

In this paper, we empirically investigate the impact of reviews with code-reviewer assignment problem has on reviewing time as compared to the reviews without code-reviewer assignment problem. From our manual examination, we find that 4%-30% of reviews have code-reviewer assignment problem. These reviews significantly take 12 days longer to approve a code change. A code-reviewer recommendation tool is necessary in distributed software development to speed up a code review process.

To help developers find appropriate code-reviewers, we propose REVFINDER, a file location-based code-reviewer recommendation approach. In order to evaluate REVFINDER, we perform a case study on 42,045 reviews of four open-source software systems i.e., Android Open Source Project (AOSP), OpenStack, Qt and LibreOffice. The results show that REVFINDER correctly recommended 79% of reviews with a top 10 recommendation. REVFINDER is 4 times more accurate than REVIEWBOT. This indicates that leveraging a similarity of previously reviewed file path can accurately recommend code-reviewers. REVFINDER recommended the correct code-reviewers with a median rank of 4. The overall ranking of REVFINDER is 3 times better than that of a baseline approach, indicating that REVFINDER provides a better ranking of correctly recommended code-reviewers. Therefore, we believe that REVFINDER can help developers find appropriate code-reviewers and speed up the overall code review process.

In our future work, we will deploy REVFINDER in a development environment and perform experiments with developers to analyze how effectively and practically REVFINDER can help developers in recommending code-reviewers.

REFERENCES

[1] M. Fagan, "Design and code inspections to reduce errors in program development," *IBM Systems Journal*, vol. 15, no. 3, pp. 182–211, 1976.

[2] A. F. Ackerman, P. J. Fowler, and R. G. Ebenau, "Software Inspections and the Industrial Production of Software," in *Proceedings of a Symposium on Software Validation: Inspection-testing-verification-alternatives*, 1984, pp. 13–40.

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

[4] A. Aurum, H. Petersson, and C. Wohlin, "State-of-the-art: software inspections after 25 years," *Software Testing, Verification and Reliability*, vol. 12, no. 3, pp. 133–154, Sep. 2002.

[5] L. G. Votta, "Does Every Inspection Need a Meeting?" in *SIGSOFT'93*, 1993, pp. 107–114.

[6] A. Bacchelli and C. Bird, "Expectations, Outcomes, and Challenges of Modern Code Review," in *ICSE '13*, 2013, pp. 712–721.

[7] P. C. Rigby and M.-A. Storey, "Understanding broadcast based peer review on open source software projects," in *ICSE'11*, 2011, pp. 541–550.

[8] V. Mashayekhi, J. Drake, W.-T. Tsai, and J. Riedl, "Distributed, collaborative software inspection," *IEEE Software*, vol. 10, no. 5, pp. 66–75, 1993.

[9] M. Beller, A. Bacchelli, and A. Zaidman, "Modern Code Reviews in Open-Source Projects: Which Problems Do They Fix?" in *MSR'14*, 2014, pp. 202–211.

[10] R. G. Kula, C. C. A. Erika, N. Yoshida, K. Hamasaki, K. Fujiwara, X. Yang, and H. Iida, "Using Profiling Metrics to Categorise Peer Review Types in the Android Project," in *ISSRE'12*, 2012, pp. 146–151.

[11] P. C. Rigby and C. Bird, "Convergent Contemporary Software Peer Review Practices," in *ESEC/FSE 2013*, 2013, pp. 202–212.

[12] S. Mcintosh, Y. Kamei, B. Adams, and A. E. Hassan, "The Impact of Code Review Coverage and Code Review Participation on Software Quality," in *MSR'14*, 2014, pp. 192–201.

[13] P. Thongtanunam, X. Yang, N. Yoshida, R. G. Kula, C. C. Ana Erika, K. Fujiwara, and H. Iida, "ReDA : A Web-based Visualization Tool for Analyzing Modern Code Review Dataset," in *ICSME'14*, 2014, pp. 606–609.

[14] P. C. Rigby, D. M. German, and M.-A. Storey, "Open Source Software Peer Review Practices : A Case Study of the Apache Server," in *ICSE'08*, 2008, pp. 541–550.

[15] P. Weiß gerber, D. Neu, and S. Diehl, "Small Patches Get In !" in *MSR'08*, 2008, pp. 67–75.

[16] J. Tsay, L. Dabbish, and J. Herbsleb, "Let's Talk About It: Evaluating Contributions through Discussion in GitHub," in *FSE'14*, 2014, pp. 144–154.

[17] Y. Jiang, B. Adams, and D. M. German, "Will My Patch Make It ? And How Fast ? Case Study on the Linux Kernel," in *MSR'13*, 2013, pp. 101–110.

[18] A. Bosu and J. C. Carver, "Impact of Developer Reputation on Code Review Outcomes in OSS Projects : An Empirical Investigation," in *ESEM'14*, 2014, pp. 33–42.

[19] G. Gousios, M. Pinzger, and A. van Deursen, "An Exploratory Study of the Pull-based Software Development Model," in *ICSE'14*, 2014, pp. 345–355.

[20] A. Mockus and J. D. Herbsleb, "Expertise Browser: a quantitative approach to identifying expertise," in *ICSE'02*, 2002, pp. 503–512.

[21] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in *ICSE'06*, 2006, pp. 361–370.

[22] R. Shokripour, J. Anvik, Z. M. Kasirun, and S. Zamani, "Why So Complicated ? Simple Term Filtering and Weighting for Location-Based Bug Report Assignment Recommendation," in *MSR'13*, 2013, pp. 2–11.

[23] X. Xia, D. Lo, X. Wang, B. Zhou, M. Kersten, F. Heidrich, O. Thomann, and G. Gayed, "Accurate Developer Recommendation for Bug Resolution," in *WCRE'13*, 2013, pp. 72–81.

[24] D. Surian, N. Liu, D. Lo, H. Tong, E.-P. Lim, and C. Faloutsos, "Recommending People in Developers' Collaboration Network," in *WCRE'11*, 2011, pp. 379–388.

[25] Y. Tian, P. S. Kochhar, E.-p. Lim, F. Zhu, and D. Lo, "Predicting Best Answerers for New Questions : An Approach Leveraging Topic Modeling and Collaborative Voting," in *SocInfo'14*, 2014, pp. 55–68.

[26] G. Jeong, S. Kim, T. Zimmermann, and K. Yi, "Improving code review by predicting reviewers and acceptance of patches," in *ROSAEC-2009-006*, 2009.

[27] Y. Yu, H. Wang, G. Yin, and C. X. Ling, "Reviewer Recommender of Pull-Requests in GitHub," in *ICSME'14*, 2014, pp. 610–613.

[28] V. Balachandran, "Reducing Human Effort and Improving Quality in Peer Code Reviews using Automatic Static Analysis and Reviewer Recommendation," in *ICSE '13*, 2013, pp. 931–940.

[29] P. Thongtanunam, R. G. Kula, A. E. C. Cruz, N. Yoshida, and H. Iida, "Improving Code Review Effectiveness through Reviewer Recommendations," in *CHASE'14*, 2014, pp. 119–122.

[30] K. Hamasaki, R. G. Kula, N. Yoshida, C. C. A. Erika, K. Fujiwara, and H. Iida, "Who does what during a Code Review ? An extraction of an OSS Peer Review Repository," in *MSR'13*, 2013, pp. 49–52.

[31] P. Kampstra, "Beanplot: A boxplot alternative for visual comparison of distributions," *Journal of Statistical Software*, vol. 28, no. 1, pp. 1–9, 2008.

[32] D. Gusfield, *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*, 1997.

[33] E. T. Barr, C. Bird, P. C. Rigby, A. Hindle, D. M. German, and P. Devanbu, "Cohesive and Isolated Development with Branches," in *FASE '12*, 2012, pp. 316–331.

[34] I. T. Bowman, R. C. Holt, and N. V. Brewster, "Linux as a case study: Its extracted software architecture," in *ICSE '99*, 1999, pp. 555–563.

[35] J. Kittler, I. C. Society, M. Hatef, R. P. W. Duin, and J. Matas, "On Combining Classifiers," *IEEE TPAMI*, vol. 20, no. 3, pp. 226–239, 1998.

[36] T. K. Ho, J. Hull, S. N. Srihari, and S. Member, "Decision Combination in Multiple Classifier Systems," *IEEE TPAMI*, vol. 16, no. 1, pp. 66–75, 1994.

[37] R. Ranawana and V. Palade, "Multi-Classifier Systems - Review and a Roadmap for Developers," *IJHIS*, vol. 3, no. 1, pp. 1–41, 2006.

[38] Z. Guan and E. Cutrell, "An Eye Tracking Study of the Effect of Target Rank on Web Search," in *CHI'07*, 2007, pp. 417–420.

[39] C. Tantithamthavorn, R. Teekavanich, A. Ihara, and K.-i. Matsumoto, "Mining A Change History to Quickly Identify Bug Locations : A Case Study of the Eclipse Project," in *ISSREW'13*, 2013, pp. 108–113.

[40] C. Tantithamthavorn, A. Ihara, and K.-I. Matsumoto, "Using Co-change Histories to Improve Bug Localization Performance," in *SNPD'13*, Jul. 2013, pp. 543–548.

[41] D. Ma, D. Schuler, T. Zimmermann, and J. Sillito, "Expert Recommendation with Usage Expertise," in *ICSM'09*, 2009, pp. 535–538.