Improving Code Review Effectiveness through Reviewer Recommendations

Patanamon Thongtanunam*, Raula Gaikovina Kula†, Ana Erika Camargo Cruz*,
Norihiro Yoshida*, Hajimu Iida*

* Nara Institute of Science and Technology, Japan
{patanamon-t,camago,yoshida}@is.naist.jp,
iida@itc.naist.jp

†Osaka University, Japan
raula-k@ist.osaka-u.ac.jp

ABSTRACT

Effectively performing code review increases the quality of software and reduces occurrence of defects. However, this requires reviewers with experiences and deep understandings of system code. Manual selection of such reviewers can be a costly and time-consuming task. To reduce this cost, we propose a reviewer recommendation algorithm determining file path similarity called FPS algorithm. Using three OSS projects as case studies, FPS algorithm was accurate up to 77.97%, which significantly outperformed the previous approach.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms

Algorithm

Keywords

Peer Code Review; Recommendation System; Open Source Software; Software Quality;

1. INTRODUCTION

Distributed software development, especially with global collaboration through online tools, is increasingly used in software firms [8]. This has led to the birth of a modern peer code review process [2]. Modern peer code review is a tool-supported review that currently used in both industrial and Open Source Software (OSS) projects. This is an effective technique for defect detection, which requires reviewers with experience and a deep understanding of the related system code [1]. However, in a much larger community and relatively impersonal landscape of distributed software development, manual selection of an appropriate reviewer can be a labored and time-consuming task [6]. Thus, there is a growing need of a support tool [7] to accommodate a wide range of knowledge of developers in large scale projects.

Recently, a reviewer recommendation tool (called Review Bot) has been proposed by Balachandran [3]. This tool is developed to reduce human effort for modern peer code review in the industrial setting of VMware. The recommendation algorithm of this tool uses a source code change history, which considers line-by-line code modification history. Appropriate reviewers are distinguished as having recently examined the same line in the history. Also, time prioritization was used to improve the performance.

The Review Bot's algorithm seems best suited to those projects with frequent changes of source code. However, we believe that this granularity of code change history can be limited. Since modern peer code review is relatively new, most projects that adopted this technique would already be in the maintenance phase. This makes most of code change history unavailable. Additionally attributes such as its large-scale codebase as well as the many diverse global development teams may be influencing factors. Therefore, it is challenging to find appropriate reviewers for those projects with lacking of code changes history.

In this paper, we propose a novel recommendation algorithm, called File Path Similarity (FPS) algorithm. The algorithm determines similarity of reviews based on the location of changed file (file path). Our key assumption is that files that are located in similar file paths would be managed and reviewed by similar experienced expert code reviewers. The motivation behind this is that in most large systems, like the Linux kernel, the directory structure loosely mirrors the system architecture and files with similar functions are usually located in the same or near directories [4].

The performance of FPS algorithm was evaluated using three distributed Open Source Software (OSS) projects: Android Open Source Project (AOSP), OpenStack, and Qt. In particular, we addressed the following research questions as a guideline: ***RQ1.)*** *Does FPS outperform the Review Bot's algorithm?*, and ***RQ2.)*** *Does a recent modification history also influence the accuracy of the algorithms?*.

We measured the accuracy of the algorithms for the top-1, top-3 and top-5 recommendations at different time priorities. Our findings confirmed that our FPS algorithm (accurate up to 77.97%) outperformed Review Bot (accurate up to 38.9%). The results suggest that in an distributed environment, the analysis at the directory structure level is more effective than at source code level. We believe that this study opens new in-

CHASE'14, June 2 – June 3, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2860-9/14/06...$15.00
http://dx.doi.org/10.1145/2593702.2593705

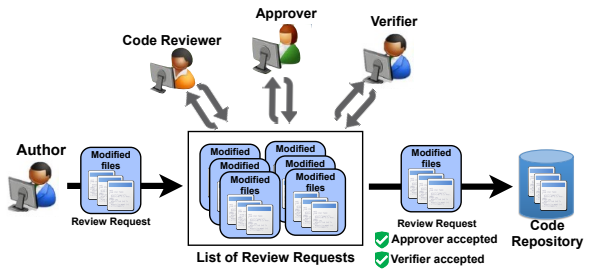


Figure 1: Code Review Process of Gerrit (Simplified Version)

sights into reviewer recommendation systems for distributed software projects.

2. MODERN PEER CODE REVIEW

A modern peer code review refers to a software inspection method that is 1) informal (contrast to traditional method), 2) tool-based, and 3) currently used in both industrial and OSS development [2]. Since practitioners of this method can coordinate asynchronously through a review tool, the adoption to distributed software development would be effortless.

The tool that our case study projects use is Gerrit[1] The process of this tool is shown Fig. 1. It begins with an author creating a patch and submitting a set of new or modified files as a review request to the system. This request is then added to a awaiting list to be reviewed. An author can manually assign reviewers to reduce the waiting time. However, the assigned reviewers also can ultimately decide whether to review the request, based on their experience and interests [9]. In Gerrit, the reviewers (i.e. Code reviewers, Approvers and Verifiers) can examine the patch of a review request with different permissions. Code reviewers can give opinion and comments. Approvers mainly determine the quality and impact of the changes. Verifiers are responsible for integration testing, which is normally automated by Gerrit. Code reviewers can be anyone in the project while approvers and verifiers are experienced reviewers designated by project leads. In the final step of the review process, a patch can be merged into the projects only when the review request is accepted by at least one approver and one verifier.

A code review will be effective if an author requests appropriate reviewers. However, without information about available reviewers in Gerrit, it is difficult for the authors to select those reviewers. Therefore, an automatic reviewer recommendation system should allow code reviews to be performed with less time and effort.

3. REVIEWER RECOMMENDATION

Based on the assumption, FPS algorithm selects candidates from reviewers who had examined files with similar directory paths. Furthermore, this algorithm also uses time prioritization in the same way as the Review Bot's algorithm. A detail of FPS algorithm is described in Algorithm 1. This algorithm takes two inputs: a new review request (R_n) and the number of top candidates (k) to be recommended. The algorithm returns a list of the top-k candidates ordered by their file path similarity scores.

Algorithm 1 *RecommendReviewers*(R_n, k)

```
1: candidates ← list()
2: pastReviewList ← getPastReviews(R_n)
3: m ← 0
4: for Review R_p : pastReviewList do
5:    score ← FPS(R_n, R_p, m)
6:    for Reviewer r : getReviewers(R_p) do
7:       candidates[r] ← candidates[r] + score
8:    end for
9:    m ← m + 1
10: end for
11: candidates.sort()
12: return candidates[0 : k]
```

In line 2, a list of past reviews of R_n is retrieved. The past reviews in this list are the reviews that closed by either being accepted or rejected, before the creation date of R_n. This list is sorted by the creation date of the past reviews in reverse chronological order. In line 3, the value of m is initialized to zero. This variable help us to score past reviews when time prioritization is considered; so that as its value increases, the past review being scored is older and a minor score is given. In lines 4-10, the FPS function is iterated for every past review. In lines 6-8, for each past review, its reviewers are retrieved and assigned as candidates. Their scores are increased by the FPS score of this past review. In line 11, the list of candidates is sorted in descending order based on their scores. Line 12 returns the top k candidates with the highest score from the sorted list.

The calculation of FPS function is described in Equation 1. This calculates a score of a past review (R_p) from an average of similarity of every file in R_p (f_p) comparing with every file in R_n (f_n). The Files function returns a set of file paths of the input review. The Similarity(f_n, f_p) function measures the similarity between f_p and f_n, using Equation 2. The averaged similarity score is prioritized by m and δ value. Same as time prioritization of the Review Bot's algorithm, the δ parameter is a time prioritization factor ranging $(0, 1]$. When $\delta = 1$, the time prioritization is not considered.

$$\text{FPS}(R_n, R_p, m) = \frac{\sum_{\substack{f_n \in \text{Files}(R_n),\\ f_p \in \text{Files}(R_p)}} \text{Similarity}(f_n, f_p)}{|\text{Files}(R_n)| \times |\text{Files}(R_p)|} \times \delta^m \quad (1)$$

$$\text{Similarity}(f_n, f_p) = \frac{\text{commonPath}(f_n, f_p)}{\max(\text{Length}(f_n), \text{Length}(f_p))} \quad (2)$$

In Equation 2, the commonPath(f_n, f_p) function counts the number of common directory and/or file name that appear in both file path from the beginning. This count is based on the assumption that files, which are under the same directory, would have the similar function. Thus, the first directory of file paths is compared firstly. Then, the other components of file path are compared respectively. For example, suppose f_n is `/src/camera/video/a.java` and f_p is `/src/camera/photo/a.java`. The common path will be `/src/camera` and the commonPath(f_n, f_p) returns 2. The count of commonPath(f_n, f_p) can be formularized as Equa-

[1]AOSP: https://android-review.googlesource.com, and OpenStack: https://review.openstack.org, and Qt: https://codereview.qt-project.org/

tion 3, where the values of i and j are initial from 0.

$$\text{commonPath}(f_n, f_p, i, j) = \begin{cases} \text{commonPath}(f_n, f_p, i+1, \\ \quad j+1)+1, \text{if } f_n[i] = f_p[j] \\ \\ 0 \qquad\qquad\qquad \text{otherwise} \end{cases} \tag{3}$$

Continuing to the previous example, the Similarity(f_n, f_p) returns the value of common path number normalized by the maximum length of these two file path, i.e. $\frac{2}{\max(4,4)} = 0.5$

4. EXPERIMENT

The effectiveness of FPS algorithm and Review Bot algorithm were measured using the same method as as the Review Bot research [3], i.e. Top-k accuracy. For N review requests, the Top-k accuracy can be calculated using Equation 4. A recommendation is correct when at least one of the top k recommended reviewers actually examined a review request.

$$\text{Top-}k \text{ Accuracy} = \frac{\#\text{correct top-}k \text{ recommendations}}{N} \times 100\% \tag{4}$$

The review histories of the case study projects were obtained from Hamasaki et. al [5]. For the experiment, we selected the reviews that *a*) are already closed ; *b*) have at least one approver; and *c*) have at least one other file besides the commit message file. Table 1 summarizes the data sets used in our study.

Table 1: Summary of Data Sets

	AOSP	OpenStack	Qt
Study Period	10/08 - 01/12	07/11 - 05/12	05/11 - 05/12
#Reviews	5,126	6,586	23,810
#Approvers	51	64	128
#Files	26,840	12,275	78,400

We evaluated the recommendation algorithms by performing recommendations for every review in chronological order. In this study, both algorithms recommended candidates for a review request by selecting only approvers who had examined a review request in the past. This is because this kind of reviewers is the most important reviewers in Gerrit system. We measured the Top-1, Top-3, and Top-5 accuracy of the algorithms with different δ values of time prioritization factor. Table 2 lists an accuracy of both algorithms using the time prioritization factors, $\delta = \{1, 0.8, 0.6, 0.4, 0.2\}$.

4.1 Experimental Results

For ***RQ1: Does FPS outperform the Review Bot's algorithm?***, the results indicate that FPS algorithm works more effectively than the Review Bot's algorithm. For AOSP, FPS algorithm had the highest recommendation accuracy of 77.12 % while the Review Bot's algorithm had the highest recommendation accuracy of 29.30 %. For OpenStack, FPS algorithm had the highest recommendation accuracy of 77.97 % while the Review Bot's algorithm had the highest recommendation accuracy of 38.90 %. However, for the relatively larger project of Qt, neither of the two algorithms accurately recommended reviewers. They only had the recommendation accuracies of about 27-36%.

We used statistical test to determine if the recommendation accuracy of FPS algorithm is higher than the recommendation accuracy of the Review Bot's algorithm for all recommendations. For the test, we determined Top-5 accuracy of both algorithms where $\delta = 1$ for every number of reviews. The recommendation accuracy of both algorithms were plotted in Fig. 2. In these charts, the x-axis represents the number of reviews N that the algorithms had recommended. The y-axis represents the accuracy calculated corresponding to that number of reviews. We took a one-side statistical test with null hypothesis $\mathbf{H}_0$: *No statistical difference in the recommendation accuracy percentages of both algorithms* and the greater case for alternative hypothesis. We used a paired-wise Mann-Whitney significance test since the data did not follow a normal distribution. From the tests against all three projects, $\mathbf{H}_0$ can be rejected as their all p-values are less than significant value, i.e. $\alpha < 0.05$.

For ***RQ2: Does a recent modification history also influence the accuracy of the algorithms?***, the results in Table 2 interestingly show that FPS algorithm without time prioritization factor ($\delta = 1$) achieved the highest recommendation accuracy. This was similar for the Review Bot's algorithm. Neither of time prioritization factors can significantly increase the accuracy. This is surprising as previous work did suggest a time prioritization ($\delta = 0.8$) yielded the best results. This finding indicates that in the histories of these projects, an approver who would examine the next review request was not necessary to be the recent one. The reasons behind this phenomena could be interesting future avenues of research. Possible future avenues could be studying a review selection of reviewers along with the code evolution as well as tangled code changes investigation to improve our results.

According to the preliminary results, we were able to answer our research questions as follows: For RQ1, we statistically showed that our FPS algorithm outperformed the Bot Review algorithm. For RQ2, we found that, in distributed projects environment, the time prioritization factor did not improve results.

4.2 Discussion

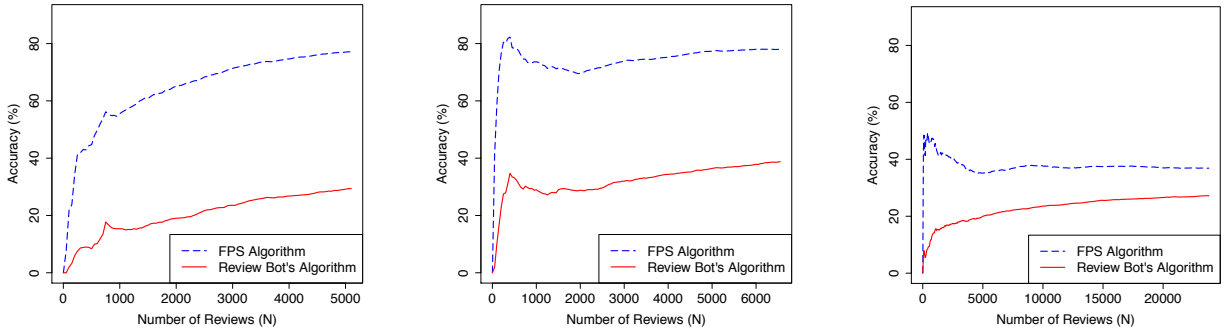
Limitation of Review Bot's algorithm: As we conjectured, the Review Bot's algorithm had a poor performance due to the lack of change lines histories. In examining the data sets, we found that the line change histories of these projects were relatively limited. For example, 80.62% of the files in AOSP were modified only once. This was similar for both OpenStack and Qt.

Limitation of FPS algorithm: For the poor performance of our FPS algorithm in Qt, the cause can be from the file path comparison of function commonPath(f_n, f_p, i, j) in Equation 3. We addressed this point because this calculation was based on our assumption which possibly did not correspond to directory structure of Qt. We plan to explore this to improve our algorithm in our future work. Furthermore, other unknown factors also can be involved for larger scale project. Closer case by case analysis as well as peer review interviews will be needed to better understand and improve our method. We also need to investigation other large scale projects for confirmation.

Repeating Recommended Reviewers: From the observation, we found that some reviewers had examined a lot of review requests. Since both algorithms recommend reviewer using historical data, it is possible that those reviewers would be frequently recommended. Consequently, they would be burdened with a huge number of assigned review requests.

Table 2: Top-k accuracy of FPS algorithm and Review Bot's algorithm with different time prioritization factors (δ)

Project	Top-k	FPS Algorithm					Review Bot's Algorithm				
		$\delta = 1$	$\delta = 0.8$	$\delta = 0.6$	$\delta = 0.4$	$\delta = 0.2$	$\delta = 1$	$\delta = 0.8$	$\delta = 0.6$	$\delta = 0.4$	$\delta = 0.2$
AOSP	Top-1	41.81 %	40.50 %	39.21 %	38.29 %	21.30 %	21.38 %	21.63 %	21.71 %	21.77 %	44.50 %
	Top-3	69.84 %	67.66 %	66.56 %	64.83 %	63.38 %	29.15 %	29.17 %	29.17 %	29.20 %	29.20 %
	Top-5	77.12 %	75.30 %	74.17 %	72.36 %	70.80 %	29.30 %	29.30 %	29.30 %	29.30 %	29.30 %
OpenStack	Top-1	38.32 %	36.47 %	35.18 %	34.73 %	34.04 %	22.94 %	23.23 %	23.19 %	23.20 %	23.19 %
	Top-3	67.57 %	63.09 %	62.45 %	62.10 %	61.66 %	35.76 %	35.76 %	35.68 %	35.55 %	35.53 %
	Top-5	77.97 %	73.28 %	72.85 %	72.62 %	71.71 %	38.78 %	38.90 %	38.82 %	38.89 %	38.89 %
Qt	Top-1	13.02 %	11.64 %	10.21 %	9.45 %	8.88 %	18.64 %	18.70 %	18.72 %	18.71 %	18.75 %
	Top-3	28.82 %	21.39 %	20.15 %	19.27 %	18.46 %	26.18 %	26.19 %	26.16 %	26.16 %	26.16 %
	Top-5	36.86 %	27.25 %	26.07 %	25.34 %	24.36 %	27.18 %	27.18 %	27.17 %	27.19 %	27.19 %



(a) AOSP (b) OpenStack (c) Qt

Figure 2: Top-5 accuracy of FPS algorithm and Review Bot's algorithm with time prioritization factor ($\delta = 1$)

Thus, considering workload balancing would reduce tasks of these potential reviewers as well as the number of awaiting reviews.

5. CONCLUSION & FUTURE WORKS

In this study, we have proposed a recommendation algorithm using file path similarity for the modern peer code review process. The results indicate that our proposed FPS algorithm effectively recommend reviewers in two of the three OSS projects. This algorithm also significantly outperform the existing algorithm (i.e. Review Bot's algorithm) in these projects. Additionally, we found that the used of time prioritization was surprisingly not appropriate for the recommendation algorithms in distributed projects environment.

Our future work will concentrate on explore more insight into projects, especially large scale projects to improve the algorithm. Other evaluations will also be considered to measure the performance of the recommendation algorithms in other aspect besides the accuracy. At the same time, we will consider ways to balance the workload of reviewers to help reviewers and reduce the number of awaiting reviews.

ACKNOWLEDGMENTS

We are thankful to Dr. Mike Barker from NAIST for his valuable suggestions and discussions.

6. REFERENCES

[1] A. Aurum, H. Petersson, and C. Wohlin. State-of-the-art: software inspections after 25 years. *Software Testing, Verification and Reliability*, 12(3):133–154, Sept. 2002.

[2] A. Bacchelli and C. Bird. Expectations, outcomes, and challenges of modern code review. In *Proc. ICSE '13*, pages 712–721, 2013.

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

[4] E. T. Barr, C. Bird, P. C. Rigby, A. Hindle, D. M. German, and P. Devanbu. Cohesive and Isolated Development with Branches. In *Proc. FASE '12*, pages 316–331, 2012.

[5] 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 *Proc. MSR' 13*, pages 49–52, 2013.

[6] E. Kocaguneli, T. Zimmermann, C. Bird, N. Nagappan, and T. Menzies. Distributed development considered harmful? In *Proc. ICSE '13*, pages 882–890, 2013.

[7] A. Mockus and J. Herbsleb. Expertise Browser: a quantitative approach to identifying expertise. In *Proc. ICSE '02*, pages 503–512, 2002.

[8] N. Ramasubbu and R. K. Balan. Globally Distributed Software Development Project Performance : An Empirical Analysis. In *Proc. ESEC/FSE '07*, pages 125–134, 2007.

[9] P. C. Rigby and M.-A. Storey. Understanding broadcast based peer review on open source software projects. In *Proc. ICSE '11*, pages 541–550, 2011.