e-Informatica Software Engineering Journal, Volume 18, Issue 1, 2024, pages: 240108, DOI: 10.37190/e-Inf240108

Automated Code Reviewer Recommendation for Pull Requests

Mina-Sadat Moosareza[*], Abbas Heydarnoori[**]
[*] *Independent Researcher*
[**] *Department of Computer Science, Bowling Green State University*
m.s.moosareza@gmail.com, aheydar@bgsu.edu

Abstract

minu.15em **Background:** With the advent of distributed software development based on pull requests, it is possible to review code changes by a third party before integrating them into the master program in an informal and tool-based process called Modern Code Review (MCR). Effectively performing MCR can facilitate the software evolution phase by reducing post-release defects. MCR allows developers to invite appropriate reviewers to inspect their code once a pull request has been submitted. In many projects, selecting the right reviewer is time-consuming and challenging due to the high requests volume and potential reviewers. Various recommender systems have been proposed in the past that use heuristics, machine learning, or social networks to automatically suggest reviewers. Many previous approaches focus on a narrow set of features of candidate reviewers, including their reviewing expertise, and some have been evaluated on small datasets that do not provide generalizability. Additionally, it is common for them not to meet the desired accuracy, precision, or recall standards.
Aim: Our aim is to increase the accuracy of code reviewer recommendations by calculating scores relatively and considering the importance of the recency of activities in an optimal way.
Method: Our work presents a heuristic approach that takes into account both candidate reviewers' expertise in reviewing and committing, as well as their social relations to automatically recommend code reviewers. During the development of the approach, we will examine how each of the reviewers' features contributes to their suitability to review the new request.
Results: We evaluated our algorithm on five open-source projects from GitHub. Results indicate that, based on top-1 accuracy, top-3 accuracy, and mean reciprocal rank, our proposed approach achieves 46%, 75%, and 62% values respectively, outperforming previous related works.
Conclusion: These results indicate that combining different features of reviewers, including their expertise level and previous collaboration history, can lead to better code reviewer recommendations, as demonstrated by the achieved improvements over previous related works.

Keywords: Automated code reviewer recommendation, Modern code review, Heuristic algorithms.

Submitted: 20 Dec. 2023; Revised: 24 Mar. 2024; Accepted: 05 May 2024; Available online: 31 Jul. 2024

1. Introduction

The process of code review previously involved third parties inspecting code in face-to-face meetings before integrating code into the master branch to find and fix any problems. Due to its potential to significantly impact post-release quality, it has drawn considerable attention in the software industry. It is becoming more common to observe software systems, especially open-source ones, being developed by programmers from different geographic locations. There are powerful tools, such as GitHub, supporting this process by the means of pull requests. A pull request is an event where a contributor develops a code change and requests owners to merge it with the main program. This new program development method requires a different style of code review, called Modern Code Review (MCR). The process of MCR is tool-based and informal, where the developer invites the appropriate reviewer to inspect the code for integration after sending a pull request.

The code review process is costly because the reviewer must read, understand, and critique the code. For this reason, the author is recommended to select programmers knowledgeable about and capable of analyzing the modified sections. This is difficult and time-consuming as the pull requests volume of many projects is high. Furthermore, determining who is the most appropriate reviewer for a new pull request is more challenging in MCR since project participants' capabilities are unknown. Study [1] found that 4% to 30% of code reviews have problems assigning reviewers, in which case it takes 12 additional days for changes to be approved. Meanwhile, [2] suggests that reviewing new code changes can reduce post-release defects of software and thus improve its quality. Therefore, assigning code reviewers is a major problem in software engineering, and automated reviewer recommendations could be very useful.

There have been a variety of approaches (e.g., [2–4]) proposed to automatically suggest appropriate reviewers in order to resolve the above issue. The following factors are primarily considered for this purpose: reviewer *expertise* in terms of her previous reviews or commits on changed files (e.g., [3, 5]), number of reviewer collaborations with the new pull request author in the project (e.g., [6, 7]), reviewer *activity* based on the number of previous reviews and commits the reviewer has made on all project files (e.g., [8]), and *workload* based on the number of open reviews assigned to her (e.g., [9]). The majority of previous studies have focused on just one or two of the features mentioned above. Our experimental evaluations did indicate, however, that combining these features would allow us to recommend more appropriate reviewers.

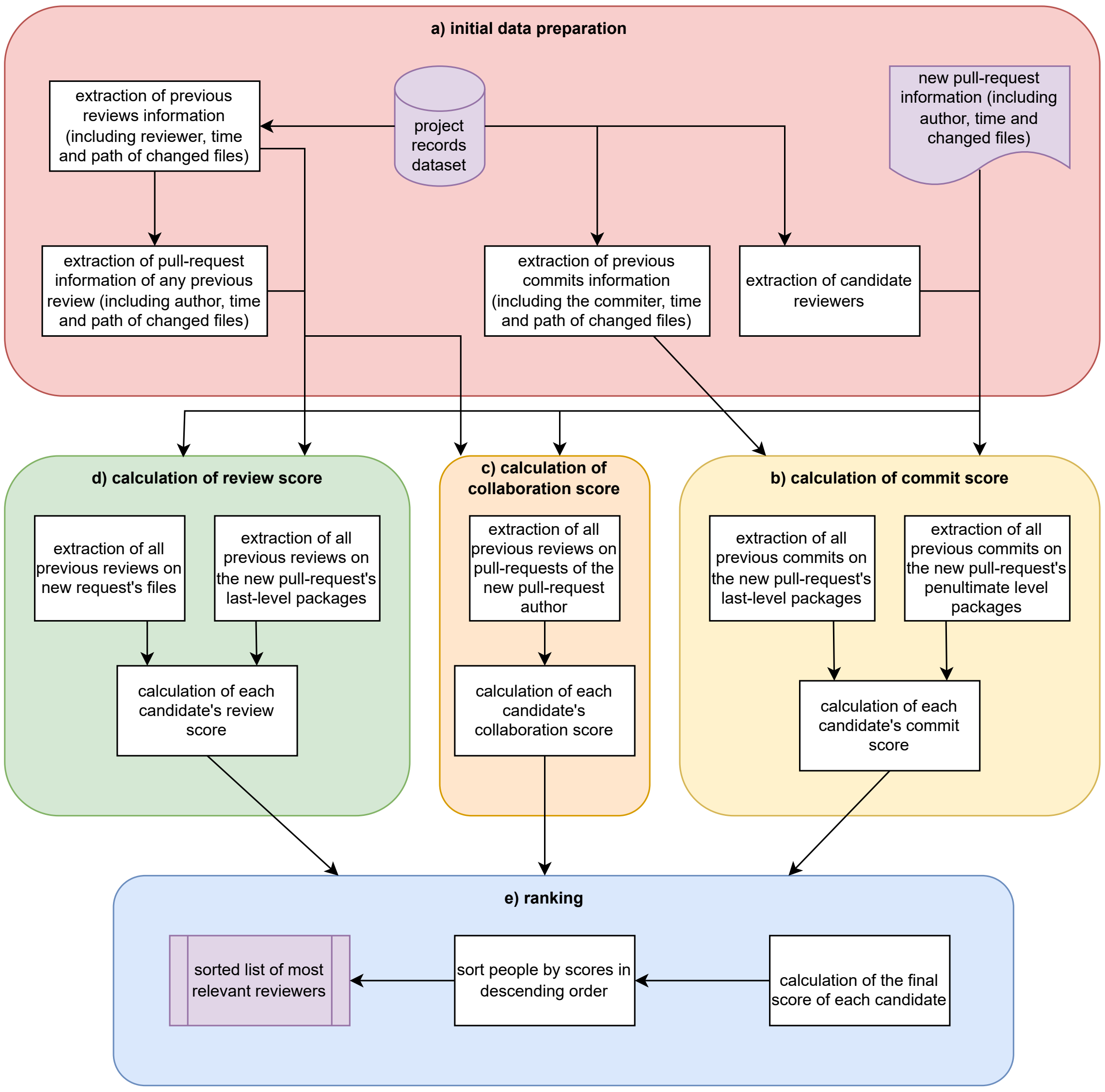
This observation led us to develop a heuristic-based approach to recommending reviewers. Our algorithm calculates three different scores for each candidate reviewer: *review expertise*, *commit expertise*, and *collaboration history*. Then, we combine them using a weighted sum. Our experiments determine the relative importance of each feature in selecting an appropriate reviewer based on these weights. Previous approaches also sometimes are evaluated on small datasets, which negatively impacts the generalizability of the results. To mitigate this issue, we evaluate our proposed algorithm on a large dataset of five GitHub projects [10]. We show that our proposed approach achieves 46%, 85%, and 92% in terms of top-1 accuracy, top-3 accuracy, and mean reciprocal rank, respectively. In summary, this paper has the following contributions:
- Introducing a new heuristics-based approach to automate reviewer recommendations for pull requests.
- Taking into account all three feature categories of review expertise, commit expertise, and collaboration.

– Quantitatively evaluating the proposed approach and indicating that it outperforms existing approaches.

This paper is organized as follows. In Section 2, we explain our new algorithm for code reviewer recommendation. Section 3 shows the evaluation setup, qualitative and quantitative results, and threats to validity. In Section 4, we discuss the pros and cons of our proposed approach. Section 5 reviews related work, and a conclusion and upcoming direction are presented in Section 6.

2. Proposed approach

Figure 1 illustrates the overall process of our heuristic solution. The previous review recommenders considered a variety of features when selecting candidates. Our approach takes expertise and collaboration into account, and we will report that the activity feature



a) initial data preparation
extraction of previous reviews information (including reviewer, time and path of changed files)
project records dataset
new pull-request information (including author, time and changed files)
extraction of pull-request information of any previous review (including author, time and path of changed files)
extraction of previous commits information (including the commiter, time and path of changed files)
extraction of candidate reviewers
d) calculation of review score
extraction of all previous reviews on new request's files
extraction of all previous reviews on the new pull-request's last-level packages
calculation of each candidate's review score
c) calculation of collaboration score
extraction of all previous reviews on pull-requests of the new pull-request author
calculation of each candidate's collaboration score
b) calculation of commit score
extraction of all previous commits on the new pull-request's last-level packages
extraction of all previous commits on the new pull-request's penultimate level packages
calculation of each candidate's commit score
e) ranking
sorted list of most relevant reviewers
sort people by scores in descending order
calculation of the final score of each candidate

Figure 1. Overview of the proposed approach

was excluded from our algorithm as it had not provided any significant improvements. Initially, the primitive data needed for subsequent steps are extracted from the existing dataset as shown in section (a). Following this, each person's score is calculated in three parallel processes (b), (c), and (d) considering their experience of commitment, review, and collaboration. The final score of each candidate is determined by combining all these scores in section (e). Afterward, the output will be a list of candidates sorted by their final scores.

We had to examine the effect of each change on the original formula during the development of our heuristic algorithm. These intermediate evaluations were conducted using the dataset provided by [10], which includes five GitHub projects. Details of this dataset are provided in Section 3.

2.1. Scores definition

This study focuses on three categories of features; collaboration, activity, and expertise, including experience of review and commitment, which will be briefly described below:

– **Review score.** Most reviewer recommendation methods take into account the reviewer's experience, and its importance has always been emphasized. In [11], the authors provide a review ownership criterion. They demonstrated that if developers participate actively in reviews, they could specialize in related code snippets. Several reviewer recommendation systems, such as those presented in [3, 5, 12], are based on review experience, and its importance has been highlighted in articles [13–15].
– **Commit score.** The authors of [16] indicated that developers who make plenty of changes to portions of a program code should be considered owners of those parts. The findings of [17] suggest that individuals are more likely to review changes they have experience with. As part of their proposed approaches, some previous automated reviewer recommendation systems, such as [5, 8, 9], have also considered commit experience. Also, in [15] it is mentioned that the level of the reviewer's commit experience impacts the usefulness of the review.
– **Collaboration score.** Collaboration score refers to a candidate reviewer's involvement with previous pull requests made by the new request's author. In some articles in the reviewer's recommendation area, including [17, 18], the importance of relationships between the candidate reviewer and the pull request author is emphasized in addition to expertise. Some of the previous automated reviewer recommendation approaches, such as the articles [6, 7, 9] approaches, have also benefited from collaboration score in addition to expertise. As a result, some previously automated reviewer recommendation approaches, such as [6, 7, 9], included collaboration scores along with expertise scores.
– **Activity score.** Lastly, some articles, such as [8] which proposed a way to recommend code reviewers, introduced a score that can be called activity. Level of activity indicates how actively a candidate participated in review processes throughout the project.

2.2. Initial formulation of scores

The first step was to calculate each score based on a relation introduced in previous articles. We then made appropriate changes to these relationships to improve them gradually during the steps described in the following subsections. In this research, the relationships presented in [5] are used to calculate the initial score of review and commit based on (1) and (2), respectively:

$$\text{ReviewScore}(r) = \sum_{f\in F} n_{\text{review}}(r,f)\frac{1}{t_{\text{review}}(r,f)} + \sum_{d\in D} n_{\text{review}}(r,d)\frac{1}{t_{\text{review}}(r,d)} \tag{1}$$

$$\text{CommitScore}(r) = \sum_{f\in F} n_{\text{change}}(r,f)\frac{1}{t_{\text{change}}(r,f)} + \sum_{d\in D} n_{\text{change}}(r,d)\frac{1}{t_{\text{change}}(r,d)} \tag{2}$$

assuming F is the set of files changed in the new pull request, while D represents the set of last-level parent directories. $n_{\text{review}}(r,f)$ and $n_{\text{change}}(r,f)$ indicate how many reviews and commits were done by reviewer r for file f, whereas $n_{\text{review}}(r,d)$ and $n_{\text{change}}(r,d)$ refer to the number of previous reviews and commits that r performed on the files in directory d, respectively. Furthermore, $t_{\text{review}}(r,f)$ and $t_{\text{change}}(r,f)$ reflect the time elapsed since reviewer r last reviewed and committed file f, while $t_{\text{review}}(r,d)$ and $t_{\text{change}}(r,d)$ represent how long has passed since the last review and commit on directory d by reviewer r, respectively. In the fractions of the final score, these time coefficients appear at the denominators, so that as time passes since a review or commit, its effect decreases.

To calculate the initial score of collaboration, we used the relation presented in [6], which is calculated according to the (3):

$$\text{CollaborationScore}(r) = n_{\text{collaboration}}(r,a) \tag{3}$$

where a refers to the author of the new pull request, and $n_{\text{collaboration}}(r,a)$ is the number of collaborations between a and reviewer r, i.e., the number of pull requests authored by a, and reviewed by r.

To calculate the initial activity score, we used the relation presented in [8], which is calculated according to the (4):

$$\text{ActivityScore}(r) = \sum_{t<360} n_{\text{review}}(r,t) \tag{4}$$

where $\sum\limits_{t<360} n_{\text{review}}(r,t)$ means the number of times which r has reviewed a pull request of the whole project in the past year.

2.3. Relative formulation of reviews and commit scores

Assume there are two cases in which reviewer r reviewed file f twice and last reviewed it one day ago, but in one case, the file has already been reviewed twice in total, while in the other, it has been reviewed a hundred times. In both cases the value of $\dfrac{n_{\text{review}}(r,f)}{t_{\text{review}}(r,f)}$ will be equal to 2. However, we know that the level of expertise of the reviewer r on the file f relative to all project reviewers is much higher in the first case than in the second one. Because in the first case, 100% of the review history of f belongs to r, but in the second, this value is only two percent, and there may be people with more experience on f in the project. Therefore, like [3, 8], we decided to calculate the review and commit experience score in relative terms as (5) and (6), respectively:

$$\text{ReviewScore}(r) = \sum_{f\in F} \frac{n_{\text{review}}(r,f)}{\sigma_{r'\in R} n_{\text{review}}(r',f)} \cdot \frac{1}{t_{\text{review}}(r,f)} + \\ + \sum_{d\in D} \frac{n_{\text{review}}(r,d)}{\sigma_{r'\in R} n_{\text{review}}(r',d)} \cdot \frac{1}{t_{\text{review}}(r,d)} \tag{5}$$

$$CommitScore(r) = \sum_{f\in F} \frac{n_{\text{change}}(r,f)}{\sigma_{r'\in R} n_{\text{change}}(r',f)} \cdot \frac{1}{t_{\text{change}}(r,f)} + \\ + \sum_{d\in D} \frac{n_{\text{change}}(r,d)}{\sigma_{r'\in R} n_{\text{change}}(r',d)} \cdot \frac{1}{t_{\text{change}}(r,d)} \tag{6}$$

where R is the list of all people who have provided reviews for the project, and $\sigma_{r'\in R} n_{\text{review}}(r',f)$ and $\sigma_{r'\in R} n_{\text{review}}(r',d)$, respectively, denote all reviews on file f and directory d. Also, $\sigma_{r'\in R} n_{\text{change}}(r',f)$ and $\sigma_{r'\in R} n_{\text{change}}(r',d)$ indicate how many commits are done on file f and directory d, respectively. We examined both absolute and relative formulas separately on our dataset to test the validity of our hypothesis regarding the importance of the relative review experience of candidates. The observations showed an improvement in the results of the review and commit scores in the relative state. Relative calculation of collaboration and activity scores is meaningless because they are calculated regardless of the files under review.

2.4. Applying the impact of time factor

We have already seen that the time factor is at the denominator of the fractions in the initial review score formula, to reduce the impact of older reviews. However, the relationship between the time elapsed since candidates' last review and their expertise is not always linear. After reviewing a file, at first, a person's level of expertise in the file decreases dramatically with each passing day, as a result of him slowly forgetting the contents of the file, and on the other hand, the content of the file altered by others over time. However, if a person has not reviewed a file for a long time, such as a year, the daily passage of time does not have a significant impact on his or her level of expertise because he or she has probably forgotten most of its contents and the file has changed more frequently.

Therefore, instead of using a linear relationship to calculate the effect of time in the denominator of fractions, we can use a relationship whose slope is initially high and then gradually decreases. This can be accomplished by considering relations like logarithms or second or higher roots for the time at the denominator. This explanation holds for the scores for collaboration, commitment, and activity, as well. We used our dataset to test the scoring formula by substituting different time coefficients since it has only been considered linearly in previous articles. For all scores, we found that the presence of the third root of the time in the denominator provided the best results. There is an average accuracy deduction with higher roots. Therefore, the formulas for calculating review, commit, collaboration, and activity scores are modified as (7), (8), (9) and (10), respectively:

$$\text{ReviewScore}(r) = \sum_{f\in F} \frac{n_{\text{review}}(r,f)}{\sigma_{r'\in R} n_{\text{review}}(r',f)} \cdot \frac{1}{\sqrt[3]{t_{\text{review}}(r,f)}} +$$

$$+\sum_{d\in D}\frac{n_{\text{review}}(r,d)}{\sigma_{r'\in R}n_{\text{review}}(r',d)}\cdot\frac{1}{\sqrt[3]{t_{\text{review}}(r,d)}} \tag{7}$$

$$\text{CommitScore}(r)=\sum_{f\in F}\frac{n_{\text{change}}(r,f)}{\sigma_{r'\in R}n_{\text{change}}(r',f)}\cdot\frac{1}{\sqrt[3]{t_{\text{change}}(r,f)}}+$$

$$+\sum_{d\in D}\frac{n_{\text{change}}(r,d)}{\sigma_{r'\in R}n_{\text{change}}(r',d)}\cdot\frac{1}{\sqrt[3]{t_{\text{change}}(r,d)}} \tag{8}$$

$$\text{CollaborationScore}(r)=n_{\text{collaboration}}(r,a)\cdot\frac{1}{\sqrt[3]{t_{\text{collaboration}}(r,a)}} \tag{9}$$

$$\text{ActivityScore}(r)=(\sum_{t<360}n_{\text{review}}(r,t)\cdot\frac{1}{\sqrt[3]{t_{\text{review}}(r)}} \tag{10}$$

In which $t_{\text{collaboration}}(r,a)$ and $t_{\text{review}}(r)$ refer to the time elapsed since the last review of r on the pull requests of the author of the new request and the project as a whole, respectively.

2.5. Examining reviews and commit scores for directories of previous levels

Using the track record of candidates committing and reviewing on the last-level directories, [5] encouraged us to test the effect on earlier-level directories as well. Considering review scores for directories at earlier levels did not improve the results, according to our experimental findings. On the other hand, the penultimate and last-level directories yielded the best results for the commit score.

The following may be the reason for the better results of commit experience on penultimate level directories over files. In a development environment, it is common for developers to repeatedly make changes to the same file once they have committed to it and familiarized themselves with its content. So, in many cases, the person who has the most experience committing on the file is the modifier of the pull request and should not be chosen as a reviewer. Additionally, files in the same path as a file usually contain similar content. The committers of these files may therefore have become acquainted with the file content during their committing. So, we changed the calculation formula for the commit score to be as (11):

$$\text{CommitScore}(r)=\sum_{d_1\in D_1}\frac{n_{\text{change}}(r,d_1)}{\sigma_{r'\in R}n_{\text{change}}(r',d_1)}\cdot\frac{1}{\sqrt[3]{t_{\text{change}}(r,d_1)}}+$$

$$+\sum_{d_2\in D_2}\frac{n_{\text{change}}(r,d_2)}{\sigma_{r'\in R}n_{\text{change}}(r',d_2)}\cdot\frac{1}{\sqrt[3]{t_{\text{change}}(r,d_2)}} \tag{11}$$

where D_1 refers to the last-level directories of files that changed in the new pull request. As well, D_2 means the set of changed files' parent directories in the level preceding the last level.

2.6. Determining Optimal Coefficients

The work presented in [5] and other articles which combine different scores by adding values, such as [3, 9], assume the same coefficients for all sentences. However, this assumption is not necessarily correct. Thus, we performed multiple experiments to obtain relative coefficients for each of the scores while combining them to optimize the results. We derived the coefficients through iterative experimentation, selecting those that yielded the most favorable experimental outcomes. AT first, we assigned a coefficient of one to the commit score, as among the various scores calculated, it exhibited the weakest correlation with reviewer suitability for new request reviews. Subsequently, we explored different coefficients for the review score, evaluating the list of recommended reviewers and their Mean Reciprocal Rank (MRR) in each instance. Accordingly, we calculated the appropriate coefficients for the two-sentence scores of review and commit, resulting in (12) and (13), respectively.

$$\begin{aligned}\text{ReviewScore}(r) = 1.25 \cdot \sum_{f\in F} \frac{n_{\text{review}}(r,f)}{\sigma_{r'\in R} n_{\text{review}}(r',f)} \cdot \frac{1}{\sqrt[3]{t_{\text{review}}(r,f)}} + \\ + \sum_{d\in D} \frac{n_{\text{review}}(r,d)}{\sigma_{r'\in R} n_{\text{review}}(r',d)} \cdot \frac{1}{\sqrt[3]{t_{review}(r,d)}}\end{aligned} \quad (12)$$

$$\begin{aligned}\text{CommitScore}(r) = \sum_{d_1\in D_1} \frac{n_{\text{change}}(r,d_1)}{\sigma_{r'\in R} n_{\text{change}}(r',d_1)} \cdot \frac{1}{\sqrt[3]{t_{\text{change}}(r,d_1)}} + \\ + 1.25 \cdot \sum_{d_2\in D_2} \frac{n_{\text{change}}(r,d_2)}{\sigma_{r'\in R} n_{\text{change}}(r',d_2)} \cdot \frac{1}{\sqrt[3]{t_{\text{change}}(r,d_2)}}\end{aligned} \quad (13)$$

Next, we employed a similar approach to determine the coefficient for the collaboration score. We combined the review and commit scores, then combined them with the scores of collaboration and activity, determining the optimal coefficient for each. As for the activity score, we found that the best result was achieved with a coefficient of about 0.002. Therefore, the impact of the activity score is very small compared with the combination of the commit, review, and collaboration scores. Additionally, adding the activity score improved the results in only three out of five projects. With these observations, we can discard activity score when formulating heuristic relation and rely on the scores of expertise, including review and commitment, as well as collaboration. This is especially true when we observe that the average improvement of the result is only about 0.001, which is very small. Finally, the following relation (14) is our proposed heuristic relation:

$$\begin{aligned}\text{ReviewerScore}(r) = 7.7344 \cdot \sum_{f\in F} \frac{n_{\text{review}}(r,f)}{\sigma_{r'\in R} n_{\text{review}}(r',f)} \cdot \frac{1}{\sqrt[3]{t_{\text{review}}(r,f)}} \\ + 6.1875 \cdot \sum_{d_1\in D_1} \frac{n_{\text{review}}(r,d_1)}{\sigma_{r'\in R} n_{\text{review}}(r',d_1)} \cdot \frac{1}{\sqrt[3]{t_{\text{review}}(r,d_1)}} \\ + 2.25 \cdot \textstyle\sum_{d_1\in D_1} \displaystyle\frac{n_{\text{change}}(r,d_1)}{\sigma_{r'\in R} n_{\text{change}}(r',d_1)} \cdot \frac{1}{\sqrt[3]{t_{\text{change}}(r,d_1)}}\end{aligned}$$

$$+ 2.8125 \cdot \sum_{d_2 \in D_2} \frac{n_{\text{change}}(r, d_2)}{\sigma_{r' \in R} n_{\text{change}}(r', d_2)} \cdot \frac{1}{\sqrt[3]{t_{\text{change}}(r, d_2)}}$$

$$+ n_{\text{collaboration}}(r, a) \cdot \frac{1}{\sqrt[3]{t_{\text{collaboration}}(r, a)}} \tag{14}$$

3. Evaluations

In this section, we present our approach evaluation setup and report the results. We then compare it to previous related works and find that the proposed approach performs better.

3.1. Evaluations questions

We evaluated our approach to answer the following research questions:
1. What is the accuracy, precision, and recall of our code reviewer's recommendations?
2. Regarding the MRR of our approach, how is the quality of code reviewers ranking?
3. Is our algorithm superior to previous related works in terms of functionality and results?

3.2. Evaluations setup

The first step toward evaluating our approach is to submit a new pull request to our system as input. Having calculated the scores, the system produces the ranked list of reviewers as the output. Afterward, we evaluate the results using a set of criteria derived from the literature. The more actual reviewers there are on our suggested list, the better the algorithm functions.

We have written a code that completely automates our evaluations. Therefore, the personal opinions of evaluators do not affect the evaluation results. Furthermore, we compare our heuristic algorithm's results with those of previous related work to better judge its effectiveness. We will present the dataset, the evaluation criteria, and the previous work we chose to compare with our method results in the following subsections.

3.2.1. Dataset

Our analysis and evaluation of the proposed solution is based on the publicly available data presented in [10]. Due to its comprehensiveness and the fact that it covers a long period, the dataset is reliable and usable. The entire dataset is derived from nine projects on GitHub and Gerrit. The dataset we used for our research is only from its GitHub projects. There are records of five projects starting with their first pull request up until early 2020 in our dataset. As a result of the different sizes and languages used in these projects, the dataset has a favorable generality. There are only about 1 200 pull requests in the Zookeeper project, for instance. Meanwhile, there are more than 27 000 pull requests in the Spark project. In addition, on average, selected projects on GitHub have around sixteen thousand stars and more than ten thousand forks, which indicates high credibility and popularity.

This dataset contains no review date. Using the GitHub Rest API, we compiled the review dates for all five projects' pull requests. Our final dataset includes the author name,

Table 1. Specification of dataset used for evaluation

Project name	Number of total pull requests	Number of test pull requests	Number of candidate reviewers
Zookeeper	780	156	194
Kafka	5 492	549	877
Beam	6 966	697	761
Flink	5 928	593	925
Spark	16 977	1 698	2 369

Table 2. Minimum, maximum, mean and median number of reviewers for pull requests

Project name	Minimum number of reviewers	Maximum number of reviewers	Mean number of reviewers	Median number of reviewers
Zookeeper	1	6	1.98	2
Kafka	1	17	1.84	2
Beam	1	11	1.37	1
Flink	1	9	1.44	1
Spark	1	14	1.98	2

creation date, and path of modified files in each pull request, the author name, the modified files, and the commit date for each commit, as well as the reviewer name, review date, and pull request number for each review. All individuals who reviewed or committed to the project before the new pull request was made, were considered candidates for review.

First, we set aside pull requests that have not been reviewed; because a recommended reviewer must have actually reviewed the pull request for the recommendation to be correct. Therefore, we cannot use these requests in the evaluation process since their correct answer is unknown. As in previous articles, such as [1, 3], we selected some late pull requests for each project and implemented our proposed solution to anticipate who would be the best reviewers for these requests. Data from previous pull requests were also used to calculate candidates' records. In Table 1, we present the number of pull requests selected, and the number of candidate reviewers. For more information, in Table 2, we provide the values for the minimum, maximum, mean and median number of reviewers for pull requests used in experiments.

3.2.2. Evaluation criteria

As defined in previous works, such as [1, 5, 6, 10] the proposed reviewer of a recommendation system is valid when that reviewer has actually reviewed the new pull request. We leveraged the criteria used in the previous automated reviewer recommendation studies as correct and valid criteria that measure the accuracy of the approaches. Many previous works, such as [3, 5–7], have used precision@m, recall@m, and f_score@m to evaluate their solutions. In precision@m, we can see what proportion of the candidates we proposed in the final list of reviewers have actually reviewed the pull request. Recall@m shows what proportion of real reviewers is on our suggested list. The f_score@m also combines precision and recall and assigns a final score to the list of recommended reviewers for each pull request. The methods of calculating these criteria are given in the (15), (16), and (17), respectively.

$$\text{precision@m} = \frac{|RR(p) \cap AR(p)|}{|RR(p)|} \tag{15}$$

$$\text{recall@m} = \frac{|RR(p) \cap AR(p)|}{|AR(p)|} \tag{16}$$

$$\text{f\_score@m} = 2 \cdot \frac{\text{precision@m} \cdot \text{recall@m}}{\text{precision@m} + \text{recall@m}} \tag{17}$$

where p, $RR(p)$, and $AR(p)$ refer to the new pull request, the recommended reviewers set for it and its actual reviewers set, respectively. m also indicates the number of reviewers at the beginning of the list for whom we computed the criterion. As an example, if $m = 3$, the values of these criteria will be calculated for the first three individuals on the list. To match all previous work, such as [3, 6, 7], we set the value of m in our evaluations equal to 1, 2, 3, 5, and 10. Articles of the other group, such as [1, 10, 19], have used the top-k accuracy criterion to measure their solutions. Top-k accuracy indicates for what proportion of the pull requests the first k reviewers in the proposed list included at least one actual reviewer. This criterion is calculated according to (18):

$$\text{top-}k\text{ accuracy}(P) = \frac{\sum\limits_{p \in P} \text{isCorrect}(p, \text{top-}k)}{|P|} \tag{18}$$

where P is the set of pull requests and k is the number of proposed reviewers we have considered in the calculation. Following previous works, such as [1, 10, 19], we considered the values of k in our evaluations to be 1, 3, 5, and 10, respectively. The Mean Reciprocal Rank (*MRR*) is also widely used in this area. The mean reviewer rank is calculated as the average of the inverse of the first rank of the reviewers who actually reviewed the new pull request. Based on this criterion, we determine how much we have to scroll down the list of recommended reviewers to locate the first reviewer who is suitable. It is calculated as (19):

$$MRR = \frac{1}{|n|} \sum_{i=1}^{|n|} \frac{1}{\text{rank}_i} \tag{19}$$

where n is the total number of pull requests and $rank_i$ is the rank of the first correct answer in the proposed reviewers list for the i-th pull request. We have used these criteria to evaluate our approach.

3.2.3. Comparison with Related Works

Previous reviewer recommenders are not easy to judge and it is not easy to conclude that one method always achieves the highest accuracy. According to the experimental studies of [20], each of these methods is most effective for a particular dataset. To evaluate our approach, we selected the WhoDo reviewer recommender to compare our approach results with. WhoDo is a more recent approach with better results compared to more popular rivals such as Revfinder and CHRev, and the most similar heuristic method to ours. It is published in the proceedings of a well-known, high-rank conference. WhoDo has two versions: The first version only considers the expertise score of the candidates based on their commitment and review records, whereas the second version also takes into account the workload score to balance the different reviewers' workloads. It is pointed out in this work that the system may recommend reviewers who have not reviewed the pull request if the workload is taken into account. Developers tend to choose reviewers based primarily on their level of expertise since they do not know the workload of candidates. Therefore, naturally, the precision and recall of the second version of WhoDo are less than those of its first version. Hence, to demonstrate its functionality fairly, we need to compare our algorithm with WhoDo's first version since we did not consider workload. To do so, we reimplemented WhoDo first version heuristic approach according to authors explanations and using provided formulas.

The WhoDo developers showed that the second version is comparable to CHRev's introduced in [3] in terms of precision and recall. Furthermore, the authors of [3] found their system to be superior to previous methods in evaluations. Hence, by demonstrating the superiority of our approach over WhoDo's first version, we are showing that our approach is better than most previous efforts. Moreover, WhoDo was introduced in 2019 and is thus newer than many other approaches. These were the reasons why we chose WhoDo to compare our approach with.

3.3. Evaluations results

3.3.1. Quantitative results

Following the reimplementing of WhoDo, we calculated the precision@m, recall@m, and f_score@m introduced in the previous section for our algorithm and the first version of WhoDo, the results of which can be found in Table 3. Due to the fact that we utilize the Review, Commit, and Collaboration scores to find the appropriate reviewer, RCC-Finder is the name we gave our approach. RCC-Finder's precision and recall are shown by Table 3, answering the research question RQ1.

Here it is appropriate to see the raw values of Table 3 in a visual graph to see the resulting improvements more clearly. This was done by calculating precision@m, recall@m, and f_score@m for WhoDo and RCC-Finder across all five projects for integers ranging

Table 3. Comparison of Precision@m, Recall@m and F_score@m for Whodo and RCC-Finder

Project name	Zookeeper		Kafka		Beam		Flink		Spark		Average	
Approach	Whodo	RCC-Finder	Whodo	RCC-Finder	Whodo	RCC-Finder	Whodo	RCC-Finder	Whodo	RCC-Finder	Whodo	RCC-Finder
Precision@1	0.44	0.56	0.35	0.41	0.27	0.39	0.41	0.56	0.47	0.51	0.37	0.46
Recall@1	0.24	0.29	0.20	0.24	0.19	0.31	0.24	0.29	0.24	0.27	0.22	0.28
F_score@1	0.30	0.37	0.23	0.29	0.22	0.33	0.29	0.37	0.30	0.33	0.26	0.33
Precision@2	0.40	0.50	0.31	0.37	0.24	0.29	0.37	0.50	0.39	0.42	0.33	0.39
Recall@2	0.42	0.53	0.35	0.43	0.36	0.44	0.43	0.53	0.39	0.43	0.39	0.47
F_score@2	0.39	0.49	0.31	0.37	0.28	0.33	0.37	0.49	0.37	0.40	0.34	0.40
Precision@3	0.35	0.42	0.28	0.31	0.21	0.23	0.31	0.42	0.34	0.37	0.29	0.33
Recall@3	0.55	0.67	0.46	0.53	0.46	0.53	0.53	0.67	0.50	0.55	0.50	0.57
F_score@3	0.41	0.49	0.33	0.37	0.28	0.31	0.37	0.49	0.38	0.42	0.34	0.39
Precision@5	0.29	0.31	0.23	0.25	0.16	0.18	0.25	0.31	0.28	0.30	0.23	0.25
Recall@5	0.75	0.80	0.62	0.68	0.59	0.64	0.68	0.80	0.67	0.71	0.66	0.71
F_score@5	0.40	0.43	0.32	0.34	0.24	0.27	0.34	0.43	0.37	0.39	0.32	0.35
Precision@10	0.18	0.18	0.15	0.15	0.10	0.11	0.15	0.18	0.18	0.18	0.15	0.15
Recall@10	0.91	0.90	0.79	0.82	0.75	0.79	0.82	0.90	0.84	0.85	0.82	0.84
F_score@10	0.29	0.29	0.24	0.24	0.18	0.19	0.24	0.29	0.28	0.29	0.24	0.25

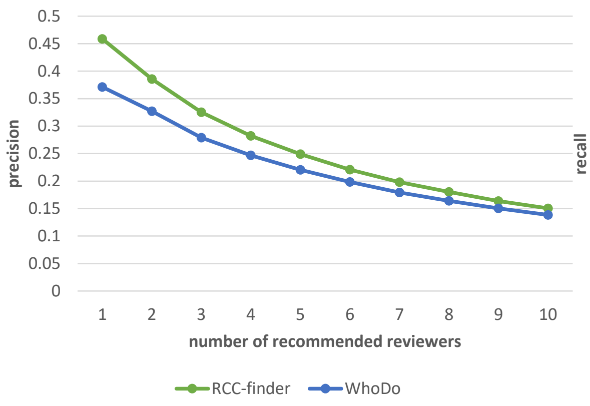


Figure 2. Comparison of the Precision@m of WhoDo with RCC-Finder for integer values of 1 to 10

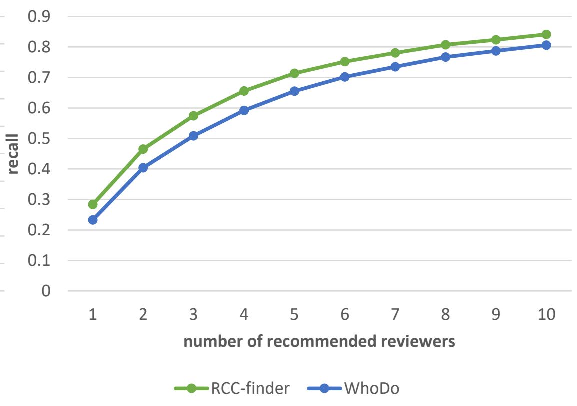


Figure 3. Comparison of the Recall@m of RCC-Finder with WhoDo for integer values of 1 to 10

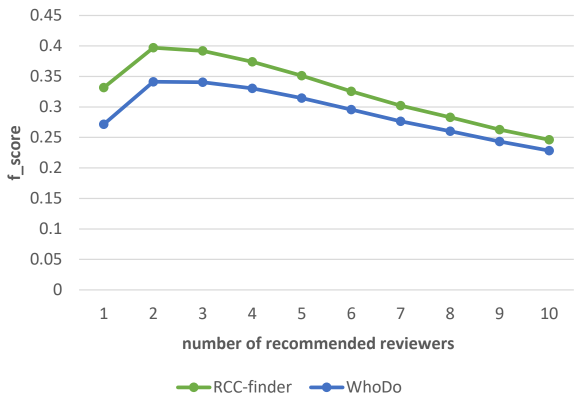


Figure 4. Comparison of the F_score@m of RCC-Finder with WhoDo for integer values of 1 to 10

from 1 to 10. Figure 2 shows the precision@m values, Figure 3 represents the recall@m values, and Figure 4 illustrates the f_score@m values for RCC-Finder compared to WhoDo.

To complete the evaluation, we calculated top-k accuracy and MRR for the two approaches along with the three criteria examined by WhoDo's developers. The results can be seen in Table 4. It shows us the quality of code reviewers' ranking in RCC-Finder in terms of MRR, providing the answer for the research question RQ2.

As visual observation transmits information better and faster, we calculated the top-k accuracy of RCC-Finder and WhoDo for all integers from 1 to 10 and provided the results in Figure 5.

Whether this improvement in the mean state reflects in the results of all projects is another question to consider. In other words, the problem is that our approach works better only in certain types of projects, which improves results in the average state or shows better performance across all projects. It's apparent from Table 4 that each project has improved, but to provide an intuitive comparison, the MRR values of RCC-Finder versus WhoDo for different projects are plotted in Figure 6.

As we know, in the WhoDo algorithm, all sentences have the same coefficient of one. While in our proposed approach, we considered the optimal coefficient for each sentence. Therefore, we have once again compared WhoDo to our approach while equalizing the

Table 4. Comparison of top-k accuracy and MRR for WhoDo and RCC-Finder

Project name	Zookeeper		Kafka		Beam		Flink		Spark		Average	
Approach	Whodo	RCC-Finder	Whodo	RCC-Finder	Whodo	RCC-Finder	Whodo	RCC-Finder	Whodo	RCC-Finder	Whodo	RCC-Finder
top-1	0.44	0.56	0.35	0.41	0.27	0.39	0.34	0.43	0.47	0.51	0.37	0.46
top-3	0.80	0.88	0.65	0.72	0.56	0.63	0.62	0.72	0.75	0.80	0.67	0.75
top-5	0.92	0.94	0.76	0.83	0.69	0.74	0.71	0.82	0.88	0.90	0.79	0.85
top-10	0.98	0.96	0.89	0.91	0.83	0.85	0.82	0.91	0.96	0.96	0.90	0.92
MRR	0.64	0.72	0.52	0.59	0.44	0.53	0.50	0.59	0.64	0.67	0.55	0.62

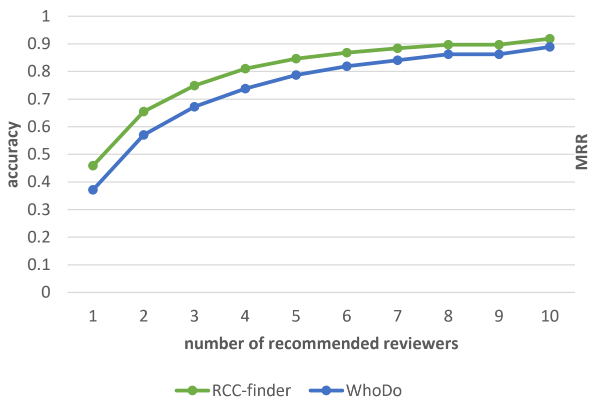


Figure 5. Comparison of the top-k accuracy of RCC-Finder with WhoDo for integer values of 1 to 10

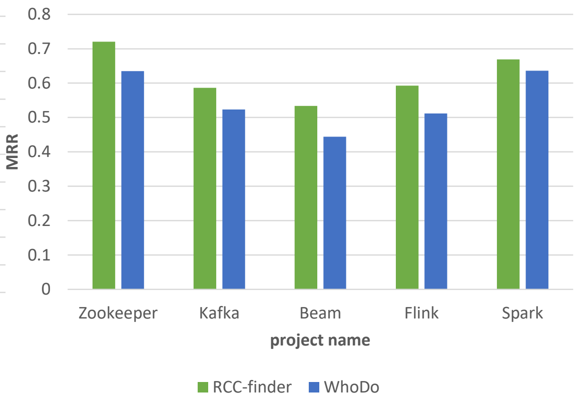


Figure 6. Comparison of the MRR of RCC-Finder with WhoDo by different projects

coefficients of all RCC-Finder sentences, so that we can assure fairness of the comparison. The results of this comparison in terms of the precision@m and recall@m for the integer values of 1 to 10 can be seen in Figure 7 and Figure 8, respectively.

3.3.2. Qualitative results

Now, according to the provided comparative charts, we can answer the research question RQ3. In all five projects, our algorithm has a higher value for almost all metrics than WhoDo. We only have slightly lower recall@10 and top-10 accuracy than WhoDo for the Zookeeper project. Therefore our approach outperforms WhoDo. As expected, our algorithm improves the automated reviewer recommendation for pull requests.

It appears from Figure 2 that the precision for the first recommendation has the highest value and has the most improvement compared to WhoDo, which is about nine percent. This amount of precision improvement is significant. Having more recommendations decreases precision since the precision formula's denominator is the recommendations number. In the numerator of this fraction, there is a subscription of recommended and actual reviewers. It's often the case that the actual reviewers of pull requests are a few, on average about two, and therefore they have little in common with the recommended reviewers. As a result, by increasing the number of reviewers recommended, the fraction denominator will grow much faster than the numerator, which leads to a decrease in precision. The precision does not improve as much in case of more recommendations since WhoDo can identify all

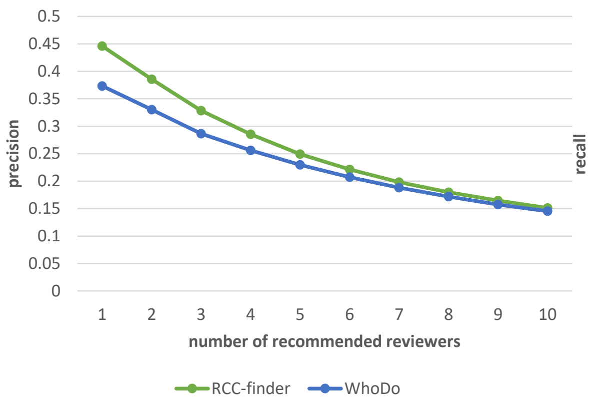


Figure 7. Comparison of the Precision@m of RCC-Finder with WhoDo with equalized coefficients for integer values of 1 to 10

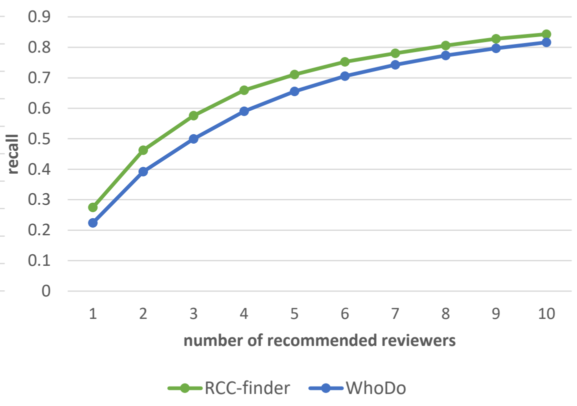


Figure 8. Comparison of the Recall@m of RCC-Finder with WhoDo with equalized coefficients for integer values of 1 to 10

the real reviewers for more pull requests. Therefore, the number of incorrectly answered requests in WhoDo decreases. So, the improvement we can make in these situations is less. The low precision can be partly attributed to developers not correctly selecting the actual reviewers. There are times in practice when candidates with less expertise and appropriateness are selected for review due to greater availability, individual contrasts, or to familiarize newcomers with the project. Moreover, the number of recommended reviewers in the denominator is much greater than the number of actual reviewers and their subscriptions with the recommended reviewers, which is in the numerator.

In the Figure 3, the recall has the highest value for the top ten recommendations, and the top two recommendations have the most recall improvement over WhoDo, which is about eight percent. This amount of enhancement in the recall is desirable. There has been a general trend of increasing recall values and decreasing improvement with the increase in recommendations. For more suggestions, the recall value increased because the denominator of the recall formula is the number of actual reviewers, which remains constant regardless of the number of recommendations. A subscription of actual reviewers and recommended reviewers is in the numerator of this fraction, which increases with more recommendations. Therefore, as the number of recommended reviewers increases, the recall get higher due to the growth in the numerator and stability of the denominator. For more recommendations, for the same reason we have stated about precision, there has been low improvement in the recall. Also, the value of recall for more suggestions in WhoDo is very high and about 90%, so it is less likely to improve.

Figure 5 shows accuracy has the highest value for top-10 recommendations, whereas top-1 recommendations yield the highest accuracy improvement over WhoDo, which is about nine percent. An improvement of this magnitude in accuracy is highly desirable. More recommendations increase the probability of getting at least one correct answer, which is the reason for increased accuracy. There is less improvement in accuracy for more recommendations. This decrease has the same explanation as decreases in recall and precision.

As shown in the Figure 6, the MRR value and consequently the quality of our candidate rankings in all projects is better than WhoDo. A 7% improvement in this criterion in the average condition is significant.

3.4. Threats to validity

The validity of this research is threatened by four types of threats: internal, external, construct, and reliability.

- **Internal validity.** This category of threats relates to the analysis and design of the approach and to factors that affect accuracy. According to [10], there are two internal threats to the data we used. A person's different user accounts are treated as distinct individuals. In addition, a robot may sometimes make automatic changes to the files of a pull request. Here, the robot is regarded as a committer.
- **External validity.** Contains threats that hinder the generalization of results to a variety of datasets. The results of our approach were evaluated using five popular and authoritative GitHub projects. With a wide range of project sizes and programming languages used, the dataset possesses acceptable generality. However, based on the evaluation of such a limited dataset, we cannot claim that our approach will yield good results for all possible projects.
- **Construct validity.** Our approach was evaluated by considering real reviewers as the best candidates, but this assumption is not always accurate. However, as far as we know, all previous works have evaluated their approach with such an assumption. Furthermore, we sought to predict the score of each person's expertise and collaboration using the formulas we presented. Nevertheless, there is no proof that our algorithm works better than all other formulas. Furthermore, we are not certain how efficient our proposed reviewers will be in practice. However, pull request authors can scroll through the list of top ten recommendations and choose the most suitable candidates at their discretion.
- **Reliability.** The reliability of an approach determines the possibility of repeating the evaluation process on the same inputs and receiving the same output. Our algorithm has a fixed formula. Therefore, all results are definite and there are no probabilistic parts in it, so if the evaluation process is repeated with the same data, the results will surely be the same. Furthermore, all the source codes and dataset of RCC-Finder are available online at https://github.com/ISE-Research/RCC-Finder. Therefore, we do not recognize any threats to the reliability of our approach.

4. Discussion

As compared to previous studies, our proposed approach has some strengths and advantages. Our scores are according to a comprehensive set of candidate features, including review experience, commit experience, collaboration records, and activity history. Previously, studies that show a combination of all three categories, expertise, collaboration, and activity were rare. Previous researchers considered a linear coefficient of time when calculating the latency. This is the first study to examine the various nonlinear coefficients of time and to use the best case, which is the third root. Also, by determining the optimal coefficients within the formula, we quantified the relative importance of each score. Based on our algorithm, we conclude, for instance, that individuals' review history is the most effective feature among those we considered. In addition, our algorithm, based on the evaluations, also recommends suitable reviewers with the desired accuracy, outperforming the previous method. Further, our database included five GitHub projects that varied widely in both project size and programming language. This makes our findings relatively generalizable.

On the other hand, this study has some shortcomings and weaknesses. During this study, as in previous ones, the criteria used compare the proposed reviewers of our approach with the actual reviewers of the new pull request. There are times, however, when actual reviewers are not the best people to review. Furthermore, we have good results from our algorithm, yet in practice, most reviews may be carried out by a small group of expert reviewers, creating an unbalanced workload for developers. Moreover, like previous methods, we recommend each pull request suitable reviewers once upon its creation. While it is not uncommon for a request to remain open for months and the level of collaboration and expertise of the project members during this time can greatly vary. As a final note, despite the high generality of our dataset, only five projects from a single review platform cannot demonstrate the algorithm's superiority in all situations.

5. Related work

In this section, we review the related work on automated code reviewer recommendation in four categories: heuristics-based approaches, machine learning-based approaches, social network-based approaches, and hybrid approaches. In general, all the features of code reviewers used in previous works fall into one of the following four categories:
- Expertise: includes features that demonstrate the level of reviewer knowledge about new pull request changed files, such as the number of or delay of reviews or commits on these files.
- Collaboration: contains items about friendship or relationships between the reviewer and the author of the new pull request, such as the number and delay of the previous reviews on author pull requests by the reviewer.
- Activity: includes items that indicate the amount of time and effort the reviewer spends on the project as a whole, such as the total number of reviews and commits on all pull requests of the project.
- Workload: contains features that demonstrate how busy a reviewer is and so how likely it is that he or she reviews the new pull request, such as the number of his or her open review requests.

In heuristics-based approaches, historical data is used to calculate a score for each candidate reviewer through heuristical algorithms, and then the candidates are sorted by their score value to determine the most relevant reviewers. For instance, Thongtanunam et al. [1] introduced RevFinder. Its heuristic is that files in the same paths are similar and can be reviewed by the same expert code reviewers. In another work [3], Bahrami Zanjani et al. proposed an approach called CHRev in which frequency, workdays, and recency of prior code reviews are calculated. Mirsaeedi and Rigby [8] have developed Sofia, which combines CHRev's advanced reviewer recommendation engine with TurnOverRec's learning and retention recommendation engine. Sofia distributes knowledge when files under review are at risk of turnover, but suggests experts otherwise. Asthana et al. [5] proposed WhoDo. They use a heuristic algorithm to recommend reviewers based on the history of commits and reviews, which considers the reviewers' workload to reduce the impact of unbalanced recommendations. Rebai et al. [6] devised a multi-objective search algorithm to find a trade-off between expertise, availability, and collaboration history By analyzing current content and resources. Chouchen et al. [21] introduced WhoReview which finds reviewers with the most experience with code changes under review while taking account of their current workload using an Indicator-Based Evolutionary Algorithm (IBEA). S*ülün* et al. in [22] calculate reviewers' scores based on the fact that an individual's knowledge of an artifact is directly correlated with the number of paths linking the individual with the artifact in a software artifact traceability graph, while inversely is proportional to the length of those paths. Then, in [23], they further developed that work by taking into account links latency. In [9] Al-Zubaidi et al. proposed a multi-objective search-oriented approach that utilized the NSGA-II algorithm to employ five criteria: code ownership, review experience, familiarity with the request author, review participation rate, and review workload. By using the NSGA-II algorithm, Chouchen et al. in [7] attempted to maximize expertise and collaboration while minimizing reviewers' workload. The previous heuristic approaches usually limited their algorithms to features from one or two categories. Especially most of them concentrated on review expertise. Also, when combining different scores through addition, all of them presumed the weight of all features equally, although their influence may differ. In our heuristic algorithm, we tried to consider features from three categories; expertise of review and commitment, collaboration, and activity. Then, we try to identify each feature's relative importance by finding the best coefficient for each sentence in the algorithm's weighted sum.

Implementing machine learning-based approaches involves building a model based on training data. Then, the performance of this model is evaluated in the prediction phase on the test data. For example, Xia et al. [24] proposed an approach called TIE that combines two learning models. One is a text-mining model that has been developed based on the textual content of the description section, the file paths, and the time of upload. In the other model, the similarities between a new review and previous reviews are calculated using a time-aware, file location-based similarity model. Sadman et al. in [25] used natural language processing techniques, a genetic algorithm, and a neural network. A variety of data is measured, including responsiveness (server logins), experience (developers' profiles and reviews previously submitted), and acquaintanceship (developers' associations with modified code blocks). Ye et al. in [26] proposed a multi-instance-based deep neural network model using CNN and LSTM networks. To recommend reviewers for pull requests, they consider three attributes, namely the title of the pull request, the commit message, and the changes made to the code. Based on a socio-technical graph, Zhang et al. in [27] proposed Coral, a new graph-based machine learning model. The graph contains a variety of entities (developers, files, pull requests, etc.) and relationships among them in modern source code management systems. They trained a graph convolutional neural network (GCN) on this graph. Chueshv et al. in [10] introduced a form of collaborative filtering, and more precisely, matrix factorization that enables the recommendation of both regular reviewers and new reviewers. In terms of precision, recall, and accuracy, most previous works in this category have delivered unsatisfactory results. Some others, like TIE, only consider features about expertise. Utilizing a broader range of feature categories, we tried to develop an algorithm that yields acceptable results.

Utilizing multiple social networks, social network-based approaches can identify the relationships between developers and their similarities, which can be used to select reviewers for new pull requests. Yu et al. [28] use cosine similarity to measure semantic similarity between pull requests based on their title and description and predict developers' scores based on how many comments they have written for similar pull requests. Moreover, they calculate collaboration scores by building a comment network between developers. Afterward, using machine learning, data retrieval, and location of files, Yu et al. [19] implemented three common approaches for assigning reviewers to pull requests. In addition, they created a Comment Network and combined it with traditional approaches. They found that hybrid approaches' overall efficiency is more stable than using different approaches separately. As a pull request may have multiple reviewers and potential influence between them, Rong et al. [29] adapted the hypergraph technique to model these high-order relationships and introduced the HGRec. Modeling of more elements is possible with HGRec, thanks to its flexible and natural model architecture. The approaches from this category typically emphasize collaboration and sometimes expertise in reviewing and have a low recall rate. Our approach considering a wide range of feature categories, such as committing expertise, led to a higher recall rate for reviewer recommendations. It means our recommended list covers more correct answers among candidate reviewers.

A hybrid approach uses a combination of different approaches explained previously to determine who should review the new pull requests. Yang et al. [30] combined an expert-based approach with data retrieval methods available in RevFinder. With the addition of a support vector classifier, dividing reviewers into technical and managerial types, they proposed a two-tier code reviewer recommendation model called RevRec. Liao et al. [31] introduced TiRR that automatically generates the topics distribution related to each reviewer, creates a reviewer-request network and a reviewers' interest network, and determines the influence weight of reviewers by analyzing these networks. Then it calculates the probability of the new pull request topic and recommends top-k reviewers. Xia et al. [32] organized the data of previous reviews into a matrix to use collaborative filtering. To capture implicit relations between reviewers and pull requests, the authors used a hybrid approach combining latent factor modeling and neighborhood methods. Pandya et al. [33] proposed CORMS, which calculates the similarity between file paths, projects/subprojects, and author information using similarity analysis. It leverages machine learning-based predictive models for reviewer recommendations based on change topics. Assavakamhaenghan et al. [34] used ChatBots along with recommendation systems to create an interactive experience for suggestions. They suggest how using a ChatBot might improve the solution, to provide more accurate and realistic reviewer recommendations. Kong et al. [35] proposed CAMP to suit the context of proprietary software development. It considers collaboration by creating a network of all participants. Using an identifier splitting algorithm, CAMP extracts common information from pull request text and file paths to account for expertise. Some previous hybrid approaches, like RevRec, evaluated their algorithm with the data of just one or two projects, which is a small dataset, resulting in unreliable results. We tried to use more data in our approach evaluation. Some others have undesirable results in terms of accuracy, precision, or recall, like RevRec, TIRR, and [32]. Another weakness among all previous works and different approaches is their attitude to the influence of latency. Our approach to considering latency in historic records importance optimally, compares different coefficients of elapsed time, while previous works either ignored it or divided scores by a simple factor of elapsed time.

6. Conclusions and future work

The review of code changes by a third party is highly beneficial before they are incorporated into the master program. Today's distributed environments make it challenging to choose the right reviewers for new pull requests. Several studies have previously proposed automated approaches for reviewer recommendation. It is important to note, however, that most of them focus on a limited set of features of candidate reviewers. Combining different features led to better results in our experiments. As a result, we proposed in this research a heuristics-based algorithm that ranks review candidates based on their expertise level and previous collaboration history. This effort was evaluated using the dataset presented in [10], which consists of five GitHub projects with various characteristics. In terms of top-1 accuracy, top-3 accuracy, and mean reciprocal rank, our approach achieved 46%, 75%, and 62% values, respectively.

Our future research plans are to consider other features, such as the candidates' areas of interest and the hours of the day they are usually available to review, to recommend a reviewer. Additionally, instead of recommending reviewers only once upon the creation of a pull request, it may be better to update the list of reviewers whenever a new commit or review related to the pull request files is made.

Acknowledgments

The authors did not receive support from any organizations for the submitted work.

References

[1] P. Thongtanunam, C. Tantithamthavorn, R.G. Kula, N. Yoshida, H. Iida et al., "Who should review my code? A file location-based code-reviewer recommendation approach for modern code review," in *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering*, 2015, pp. 141–150.

[2] S. McIntosh, Y. Kamei, B. Adams, and A.E. Hassan, "An empirical study of the impact of modern code review practices on software quality," *Empirical Software Engineering*, Vol. 5, No. 21, 2016, pp. 2146–2189.

[3] M. Bahrami Zanjani, H. Kagdi, and C. Bird, "Automatically recommending peer reviewers in modern code review," *IEEE Transactions on Software Engineering*, Vol. 42, No. 6, 2016, pp. 530–543.

[4] J. Lipcak and B. Rossi, "A large-scale study on source code reviewer recommendation," in *44th Euromicro Conference on Software Engineering and Advanced Applications*, 2018, pp. 378–387.

[5] S. Asthana, R. Kumar, R. Bhagwan, C. Bird, C. Bansal et al., "WhoDo: automating reviewer suggestions at scale," in *27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 937–945.

[6] S. Rebai, S. Molaei, A. Amich, M. Kessentini, and R. Kazman, "Multi-objective code reviewer recommendations: Balancing expertise, availability and collaborations," *Automated Software Engineering*, Vol. 27, No. 3, 2020, pp. 301–328.

[7] M. Chouchen, A. Ouni, M.W. Mkaouer, R.G. Kula, and K. Inoue, "Recommending peer reviewers in modern code review: A multi-objective search-based approach," in *Genetic and Evolutionary Computation Conference Companion*, 2020, pp. 307–308.

[8] E. Mirsaeedi and P.C. Rigby, "Mitigating turnover with code review recommendation: Balancing expertise, workload, and knowledge distribution," in *42nd ACM/IEEE International Conference on Software Engineering*, 2020, pp. 1183–1195.

[9] W.H.A. Al-Zubaidi, P. Thongtanunam, H.K. Dam, C. Tantithamthavorn, and A. Ghose, "Workload-aware reviewer recommendation using a multi-objective search-based approach," in *16th ACM International Conference on Predictive Models and Data Analytics in Software Engineering*, 2020, pp. 21–30.

[10] A. Chueshev, J. Lawall, R. Bendraou, and T. Ziadi, "Expanding the number of reviewers in open-source projects by recommending appropriate developers," in *IEEE International Conference on Software Maintenance and Evolution*, 2020, pp. 499–510.

[11] P. Thongtanunam, S. McIntosh, A.E. Hassan, and H. Iida, "Revisiting code ownership and its relationship with software quality in thescope of modern code review," in *38th International Conference on Software Engineering*, 2016, pp. 1039–1050.

[12] A. Ouni, R.G. Kula, and K. Inoue, “Search-based peerreviewers recommendation in modern code review,” in *IEEE International Conference on Software Maintenance and Evolution*, 2016, pp. 367–377.

[13] O. Kononenko, O. Baysal, L. Guerrouj, Y. Cao, and M.W. Godfrey, “Investigating code review quality: Do people and participation matter?” in *IEEE International Conference on Software Maintenance and Evolution*, 2015, pp. 111–120.

[14] O. Baysal, O. Kononenko, R. Holmes, and M.W. Godfrey, “The influence of non-technical factors on code review,” in *20th Working Conference on Reverse Engineering*, 2013, pp. 122–131.

[15] M.M. Rahman, C.K. Roy, and R.G. Kula, “Predicting usefulness of code review commentsusing textual features and developer experience,” in *14th IEEE/ACM International Conference on Mining Software Repositories*, 2017, pp. 215–226.

[16] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu, “Don’t touch my code!: Examining the effects ofownership on software quality,” in *19th ACM SIGSOFT symposium and the 13th European conference on Foundations of Software Engineering*, 2011, pp. 4–14.

[17] S. Ruangwan, P. Thongtanunam, A. Ihara, and K. Matsumoto, “The impact of human factors on the participation decision of reviewers in modern code review,” *Empirical Software Engineering*, Vol. 24, No. 2, 2019, pp. 1016–2019.

[18] A. Bosu, J.C. Carver, C. Bird, J. Orbeck, and C. Chockley, “Process aspects and social dynamics of contemporary code review: Insights from Open Source development and industrial practice at Microsoft,” *IEEE Transactions on Software Engineering*, Vol. 43, No. 1, 2016, pp. 56–75.

[19] Y. Yu, H. Wang, G. Yin, and T. Wang, “Reviewer recommendation for pull-requests in GitHub: What can welearn from code review and bug assignment?” *Information and Software Technology*, Vol. 74, 2016, pp. 204–218.

[20] Y. Hu, J. Wang, J. Hou, S. Li, and Q. Wang, “Is there a “golden” rule for code reviewer recommendation? – An experimental evaluation,” in *20th IEEE International Conference on Software Quality, Reliability and Security*, 2020, pp. 497–508.

[21] M. Chouchen, A. Ouni, M.W. Mkaouer, R.G. Kula, and K. Inoue, “WhoReview: A multi-objective search-based approach for code reviewers recommendation in modern code review,” *Applied Soft Computing*, Vol. 100, No. 106908, 2021.

[22] E. Sülün, E. Tüzün, and U. Doğrusöz, “Reviewer recommendation using software artifact traceability graphs,” in *15th International Conference on Predictive Models and Data Analytics in Software Engineering*, 2019, pp. 66–75.

[23] E. Sülün, E. Tüzün, and U. Doğrusöz, “RSTrace+: Reviewer suggestion using software artifact traceability graphs,” *Information and Software Technology*, Vol. 130, 2021.

[24] 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 *IEEE International Conference on Software Maintenance and Evolution*, 2015, pp. 261–270.

[25] N. Sadman, M.M. Ahsan, and M.A.P. Mahmud, “ADCR: An adaptive TOOL to select Appropriate Developer for Code Review based on code context,” in *11th IEEE Annual Ubiquitous Computing, Electronics and Mobile Communication Conference*, 2020, pp. 583–591.

[26] X. Ye, Y. Zheng, W. Aljedaani, and M.W. Mkaouer, “Recommending pull request reviewers based on code changes,” *Soft Computing*, Vol. 25, No. 7, 2021, pp. 5619–5632.

[27] J. Zhang, C. Maddila, R. Bairi, C. Bird, U. Raizada et al., “Using large-scale heterogeneous graph representation learning for code review recommendations at Microsoft,” in *45th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice*, 2023, pp. 162–172.

[28] Y. Yu, H. Wang, G. Yin, and C.X. Ling, “Reviewer recommender of pull-requests in GitHub,” in *IEEE International Conference on Software Maintenance and Evolution*, 2014, pp. 609–612.

[29] G. Rong, Y. Zhang, L. Yang, F. Zhang, H. Kuang et al., “Modeling review history for reviewer recommendation: A hypergraph approach,” in *44th ACM International Conference on Software Engineering*, 2022, pp. 1381–1392.

[30] C. Yang, X.h. Zhang, L.b. Zeng, Q. Fan, T. Wang et al., "RevRec: A two-layer reviewer recommendation algorithm in pull-based development mode," *Central South University*, Vol. 25, No. 5, 2018, pp. 1129–1143.

[31] L. Zhifang, W. Zexuan, W. Jinsong, Z. Yan, L. Junyi et al., "TIRR: A code reviewer recommendation algorithm with topic model and reviewer influence," in *IEEE Global Communications Conference*, 2019, pp. 1–6.

[32] Z. Xia, H. Sun, J. Jiang, X. Wang, and X. Liu, "A hybrid approach to code reviewer recommendation with collaborative filtering," in *6th International Workshop on Software Mining*, 2017, pp. 24–31.

[33] P. Pandya and S. Tiwari, "CORMS: A GitHub and gerrit based hybrid code reviewer recommendation approach for modern code review," in *30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 546–557.

[34] N. Assavakamhaenghan, R. Gaikovina, and K. Matsumoto, "Interactive chatbots for software engineering: A case study of code reviewer recommendation," in *22nd IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, 2021, pp. 262–266.

[35] D. Kong, Q. Chen, L. Bao, C. Sun, X. Xia et al., "Recommending code reviewers for proprietary software projects: A large scale study," in *IEEE International Conference on Software Analysis, Evolution and Reengineering*, 2022, pp. 630–640.