**J. Cent. South Univ.** (2018) 25: 1129−1143
**DOI:** https://doi.org/10.1007/s11771-018-3812-x

Springer

# RevRec: A two-layer reviewer recommendation algorithm in pull-based development model

YANG Cheng(杨程), ZHANG Xun-hui(张迅晖), ZENG Ling-bin(曾令斌), FAN Qiang(范强), WANG Tao(王涛), YU Yue(余跃), YIN Gang(尹刚), WANG Huai-min(王怀民)

National Laboratory for Parallel and Distributed Processing, College of Computer, National University of Defense Technology, Changsha 410073, China



**Abstract:** Code review is an important process to reduce code defects and improve software quality. In social coding communities like GitHub, as everyone can submit Pull-Requests, code review plays a more important role than ever before, and the process is quite time-consuming. Therefore, finding and recommending proper reviewers for the emerging Pull-Requests becomes a vital task. However, most of the current studies mainly focus on recommending reviewers by checking whether they will participate or not without differentiating the participation types. In this paper, we develop a two-layer reviewer recommendation model to recommend reviewers for Pull-Requests (PRs) in GitHub projects from the technical and managerial perspectives. For the first layer, we recommend suitable developers to review the target PRs based on a hybrid recommendation method. For the second layer, after getting the recommendation results from the first layer, we specify whether the target developer will technically or managerially participate in the reviewing process. We conducted experiments on two popular projects in GitHub, and tested the approach using PRs created between February 2016 and February 2017. The results show that the first layer of our recommendation model performs better than the previous work, and the second layer can effectively differentiate the types of participation.

**Key words:** Pull-Request; code reviewer recommendation; GitHub; open source community

**Cite this article as:** YANG Cheng, ZHANG Xun-hui, ZENG Ling-bin, FAN Qiang, WANG Tao, YU Yue, YIN Gang, WANG Huai-min. RevRec: A two-layer reviewer recommendation algorithm in pull-based development model [J]. Journal of Central South University, 2018, 25(5): 1129–1143. DOI: https://doi.org/10.1007/s11771-018-3812-x.

## 1 Introduction

In the past 30 years, code review has been regarded as the best practice of software engineering both in industry and open source communities [1], which can help to improve the quality of source code. Traditionally, code reviewers are third-party developers who can identify the defects in source code before integrating into the system [2], and the code review process should be held in group meetings, which will cost a lot of time. With the development of social coding communities like GitHub, traditional code review is gradually replaced by modern code review [3]. When a code change is submitted to developers for reviewing, the reviewers will cooperate with each other to discuss and give suggestions about the code.

If the change meets the need of the project and is supported by many reviewers, it will be integrated into the project's source code. However,

**Foundation item:** Project(2016-YFB1000805) supported by the National Grand R&D Plan, China; Projects(61502512, 61432020, 61472430, 61532004) supported by the National Natural Science Foundation of China
**Received date:** 2017−07−25; **Accepted date:** 2017−12−27
**Corresponding author:** YANG Cheng, Doctoral Candidate; Tel: +86−17807312257; E-mail: delpiero710@126.com; ORCID: 0000-0002-4782-1645

code review actually costs a lot of time, and the reviewers invited are not always suitable for the job [4]. Therefore, in order to reduce the time cost and improve the effectiveness of code review, it is necessary to recommend suitable reviewers to different code changes.

With the development of distributed software the pull-based model becomes more and more popular because it can lower the barrier of developers, which means that everyone can submit Pull-Requests to the repository. Because the amount of code changes increases, open source projects need more developers to review the PRs and guard the quality of the project. As there are so many developers in open source repositories and the number increases sharply, it becomes more and more important to recommend suitable and qualified reviewers to PRs.

However, there are also different kinds of reviews for PRs. We take Ruby on Rails project in GitHub as an example.

From Figures 1 and 2, we can see that some reviewers will check whether there is any problem in the code change, other people will just managerially comment below the PR description. These two kinds of reviewers form into the reviewing group of PRs in GitHub, and promote the development of open source projects cooperatively. Therefore, it is basically important to tell different kinds of reviewers apart so that we can find reviewers with different responsibility and recommend needed reviewers to different PRs.

Looking at the above factors, we find that for modern code review especially pull-based models, it is necessary to recommend developers to review code changes. There have already been many studies focusing on code reviewer recommendation. THONGTANUNAM et al [2], BALACHANDRAN [5], JEONG et al [6] and YU et al [7] focused on recommending suitable developers for code changes or PRs, either by calculating the similarity of developers' technical focus or generating features from the modified source code. However, they just focus on the precision, recall or accuracy values; the reviewers that they recommend still need to be analyzed. Meanwhile, to the best of our knowledge, there are few studies focusing on recommending fine-grained reviewers for code changes. Therefore, in this paper, we will focus on analyzing the problem of existing code reviewer recommendation methods and improve them by promoting the performance and propose a fine-grained


**Figure 1** Example of technical reviewer

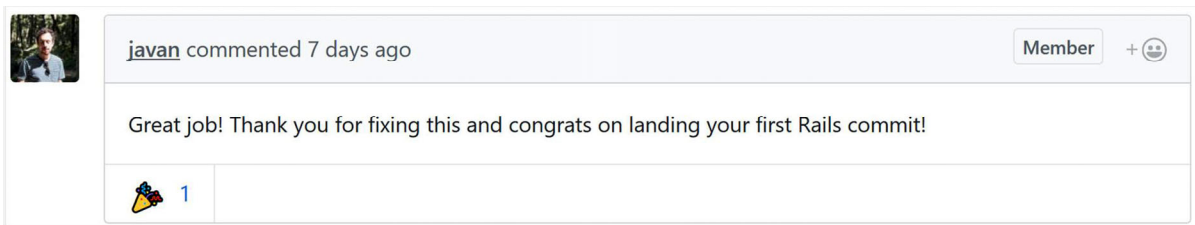


**Figure 2** Example of managerial reviewer

recommendation model.

The contributions of our work are as follows: We statistically analyze the review process in Ruby on Rails, and find that non-active reviewers are important to the reviewing process. What's more, code reviewer recommendation is of great importance to such time-consuming process. We reproduce a popular and effective recommendation algorithm (IR-based recommendation) and verify that the existing recommendation approach is easy to get into overfitting trouble, namely, excessive reviewing tasks tend to be recommended to active reviewers. We propose a hybrid reviewer recommendation approach which combines the expertise based method with existing IR and revfinder methods. We validate our approach on more than 36000 PRs in two popular GitHub projects. The result suggests that the hybrid method outperforms other separate methods regardless of the reviewers' activeness.

We propose a two-layer code reviewer recommendation model by combining the hybrid recommendation method and a SVM classifier which subdivide the reviewers into technical and managerial types. The results suggest the validation of our approach in both layers.

The rest of this paper is organized as follows. Section 2 reviews a few related studies. Section 3 describes the empirical study of PRs in Ruby on Rails. Section 4 presents the recommendation methods for the first layer. Section 5 describes the experiment settings. Section 6 discusses the experiment results. Section 7 elaborates the conclusion and describe the future plans of the study.

## 2 Related work

### 2.1 Modern code review

Code review is widely-agreed as the best software engineering practice in both industrial and open source contexts [1, 3], which helps to improve the quality of source code and reduce the defects in open source repositories. Traditionally, code reviewing is a time-consuming and heavy-weight process, which requires experts and a group meeting most often [8]. In order to speed up the process and improve the effectiveness at the same time, there comes up many modern code review methods. MCINTOSH et al [9] evaluated the impact of code review coverage to the software quality. After that, they extended their work and studied the influence of different types of code reviews through an empirical study [10]. BOSU et al [11] and MORALES et al [12] also found that code review did have impact on security vulnerabilities and design quality respectively. In addition to empirical studies, there are also many studies focusing on the development of code reviewing tools. For example, MUKADAM et al [13] developed Gerrit for reviewing android based source code.

From these studies, we can see that modern code review is becoming a more and more important process in software development, and good code reviews do have significant influence towards the software system.

### 2.2 Pull-Requests

With the development of social coding communities and distributed version control system, Pull-Request becomes a more and more popular mechanism for lowering the barrier of contributors and ensuring the code quality of open source repositories [14]. PHAM et al [15] found that PRs make contributions become more and more casual. Many "freshmen" make contributions to open source projects using PRs. On one hand, it can help to accelerate the code refinement and iteration of software version. On the other hand, more PRs need more code reviewers to review them. Meanwhile, because developers can freely submit their code changes, it becomes more and more necessary to integrate cautiously. In GitHub, there is a well-developed pull-based model, each developer in the whole community can review the code and propose their suggestions and questions about the PRs. Thanks to this mechanism, the speed of code review process becomes faster, and also this mechanism can improve the quality of each PR [16].

In our work, we mainly focus on the PRs in GitHub community. Next, we will discuss about the Pull-Requests mechanism in GitHub.

Almost all the projects in GitHub use Pull-Requests [14] and the number of PRs increases sharply [17] with the development of open source community. Therefore, it is important to deeply analyze the mechanism of PR. Firstly, developers clone the repository by forking in GitHub. Secondly,

change the source code and merge to the forked repository. Then, submit the PR to ask for the integration of the code change. Thereafter, PR reviewers test and discuss about the target PR. Finally, one of the core members of the repository decide whether to merge or close the PR. If the PR is closed, the developer can modify the PR and submit again. Otherwise, the code will be included in the main branch of the project, which can promote the development of the repository.

### 2.3 Developer recommendation

With the increase of developers and repositories in social coding communities, it is becoming more and more important to recommend developers to different tasks so that there can be a better development of open source projects [18].

#### 2.3.1 Bug assignment

There is a large number of bug reports and huge number of developers in popular open source projects (Ruby on Rails has more than 28K issues up to May 2017), which makes it a labor-intensive task to assign bugs to suitable fixers. Therefore, recommending fixers to bug reports is of significant importance. Many previous works used machine learning or information retrieval methods to triage bug reports or assign them to different developers [19–27]. JEONG et al [19] recommended bug fixers by building the tossing graph according to the bug fix history. KAGDI et al [26] matched bug reports with developers by extracting the technical terms of source code committed by contributors and bug report description. CANFORA et al [22] also used the information retrieval method, and index developers through the textual information of bug reports. ANVIK et al [28] used SVM to classify bug reports after filtering unfixed bug reports and non-active developers. SHOKRIPOUR et al [29] improved bug assignment method through using time based metadata and adding weights to technical terms.

Even though bug assignment is not directly related to our work, it is also a kind of task recommendation in software development process. The method used in this field can also be used in code reviewer recommendation.

#### 2.3.2 Contributor recommendation

There are some other works focusing on recommending contributors across different projects in the whole community. ZHANG et al [30] expanded users' activities in social coding communities by matching users from both GitHub and StackOverflow. Then, recommend potential developers who may have interests in the target project. After that, ZHANG [31] also proposed a hybrid method by combing the weighted collaborative filtering algorithm and the text matching algorithm based on the collaborative network in GitHub.

Although there are many works focusing on developer recommendation in social coding communities and many of which are about code reviewer recommendation. They still focus on the coarse-grained situations; however, our work proposed a recommendation model which can solve the fine-grained parts in code review. We not only judge whether a developer can participate a project and we also point out what kind of review may be the developer propose.

#### 2.3.3 Code reviewer recommendation

For the problem that we focus on, JEONG et al [6] predicted the code reviewer using Bayesian Network. They trained the model using features generated from the code files. THONGTANUNAM et al [2, 32] found code reviewers by comparing the file path of the target source code file. BALACHANDRAN [5] found suitable reviewers by checking the change history of source code lines. RAHMAN et al [33] recommended code reviewers across different projects. ZANJANI et al [35] presented a method named *cHRev* based on historical contributions to automatically recommend best suited reviewers for a given review. XIA et al [36] proposed a hybrid approach that combines latent factor models and neighborhood methods to capture implicit relations of code reviewers. YU et al [7, 17, 34] proposed several methods for pull-based code reviewer recommendation based on the social comment network, which could reduce the human effort in reviewing code changes.

Even though modern code review is becoming more and more popular, there are many related works focusing on recommending suitable developers to review code, they just used the information retrieval method or social network based method to find reviewers and few of them adopted machine learning algorithm in this field. In this paper, we propose a ML-based algorithm to recommend code reviewers to Pull-Requests in open source repositories.

## 3 Empirical study

For this part, we will deeply analyze the review process of PRs, and find the influence of active reviewers on code review. Because there are so many PRs and code review is a time-consuming process which is based on the reviewers' ability, it is unpractical and unnecessary for all the developers to review all the PRs. In order to deeply understand the relationship between developers and PRs, and make preparations for the code reviewer recommendation model, we propose two research questions.

RQ1: What is the difference between active and non-active code reviewers? Do PRs need non-active reviewers indeed?

RQ2: Why we say that code review is a time-consuming process?

We do empirical studies on 16049 PRs and 5075 reviewers before May 2017 in Ruby on Rails.

### 3.1 RQ1

In order to answer RQ1, we firstly analyze the reviewer number of each PR to see whether PR reviewing process needs many reviewers. The result is shown in Figure 3.

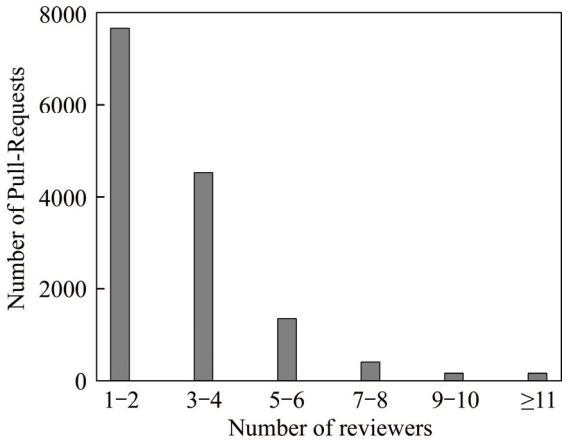


**Figure 3** Number of reviewers for Pull-Requests

From Figure 3, we can see that, the number of reviewers for each PR is very small. And the PRs having 1 or 2 reviewers take up 53.7%. Moreover, the PRs having more than 10 reviewers only take up 1.1%. That is to say, when reviewing a PR, it is not necessary to invite many reviewers.

Now that there is no need to invite so many reviewers to review a PR, will active reviewers undertake all the reviewing work? We calculate the number of reviewed PRs for each developer in Ruby on Rails, and get the changing curve of the top 100 reviewers. The result is shown in Figure 4.

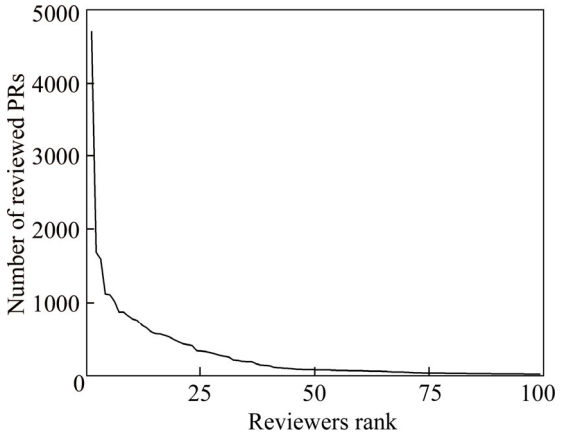


**Figure 4** Number of reviews for top 100 reviewers

Figure 4 shows that for the top 100 active reviewers in Ruby on Rails, even though the number of reviewed PRs decreases sharply from 4695 to 30, the non-active developers also undertake a large amount of reviewing work. That is to say, the difference between active and non-active developers is very big; however, non-active developers also play an important role.

We take *pixeltrix* as an example, whose activeness ranks the 13th among all the code reviewers in Ruby on Rails. He has reviewed 657 PRs till March 2017. From Figure 5, we see that *pixeltrix* also submitted reviews of high quality.

From the above analysis, we conclude that although code review is led by some super active reviewers, other reviewers can also make a lot of contribution in the code reviewing process. Therefore, when a PR comes, we do not need to recommend those developers who are highly involved in the reviewing process. On the contrary, it is important to recommend other reviewers to undertake the code reviewing work in order to reduce the burden of those active reviewers and speed up the code reviewing process.

For traditional code reviewer recommendation algorithms like the very popular IR-based method, there still is the problem for recommending experienced reviewers. We do an experiment to verify the performance of IR-based method when removing those active reviewers. The result can be seen in Figure 6 that the number of reviewers is greater than 4.

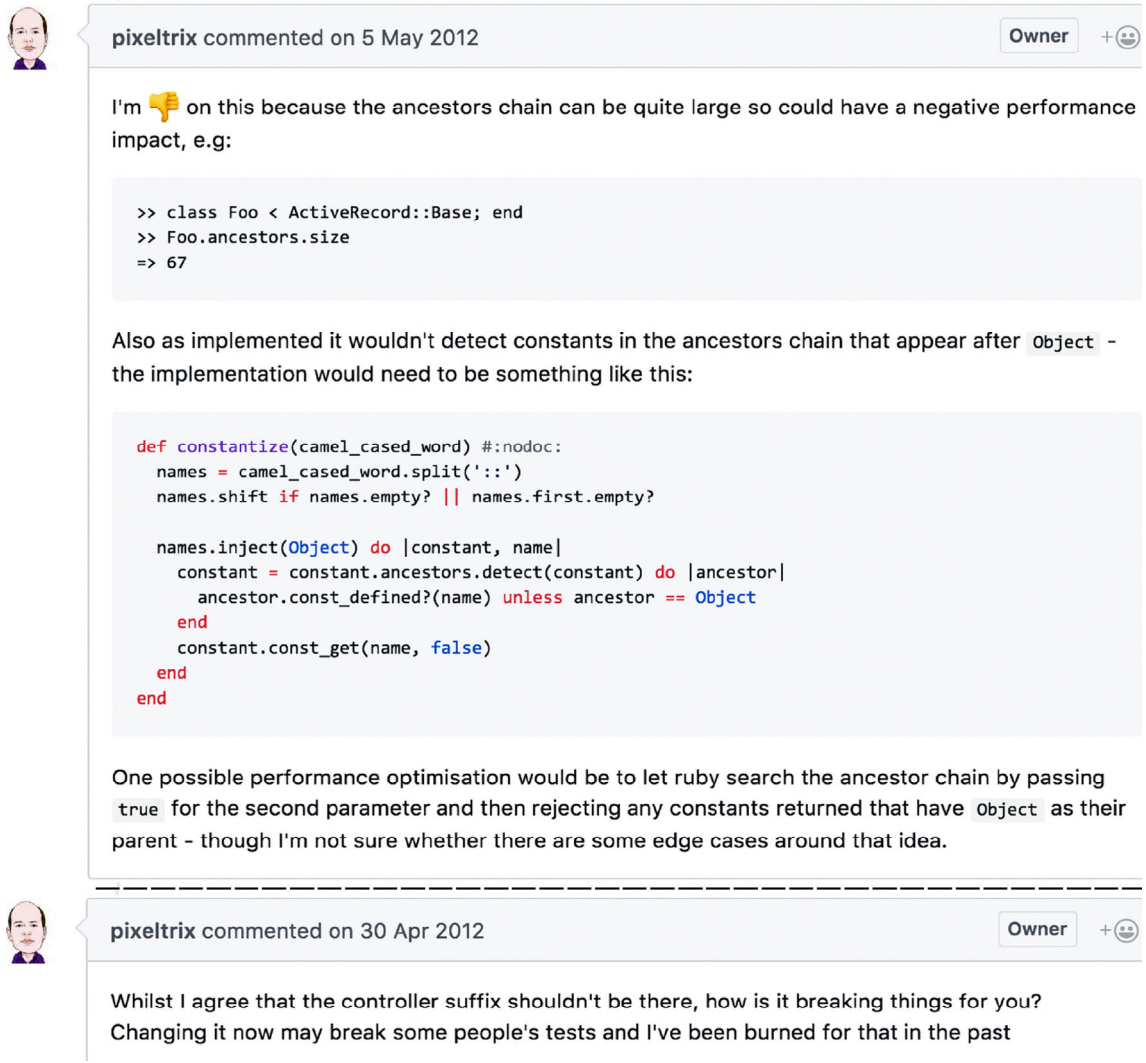


**Figure 5** Reviews by *pixeltrix*

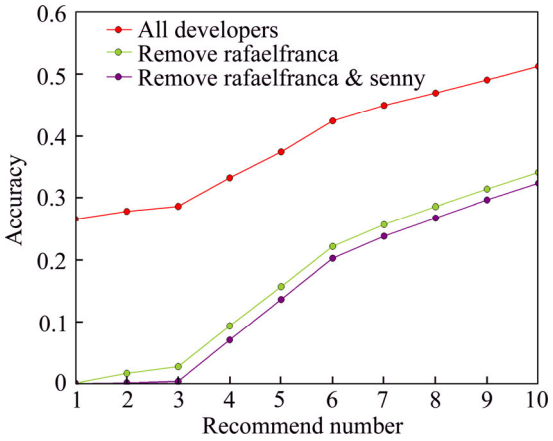


**Figure 6** Accuracy of IR-based recommendation algorithm

From Figure 6, we can see that when recommending 1 to 10 reviewers, the accuracy value of the IR-based algorithm ranges from 26.6% to 51.3%. However, when we remove the most active reviewer called "rafaelfranca", the accuracy drops a lot, ranging from 0.07% to 34.1%. Moreover, when we remove the top 2 reviewers called "rafaelfranca" and "senny", the accuracy will also drop to some extent. That is to say, the IR-based recommendation algorithm is influenced by the activeness of reviewers and tends to recommend active developers to review PRs; however, these reviewers will participate in the reviewing process spontaneously. Therefore, a more effective recommendation algorithm that focuses on the non-active reviewers is still in demand.

### 3.2 RQ2

For RQ2, in order to verify that code review is a time-consuming process, we firstly calculate the time used for closing PRs, which is shown in Figure 7.

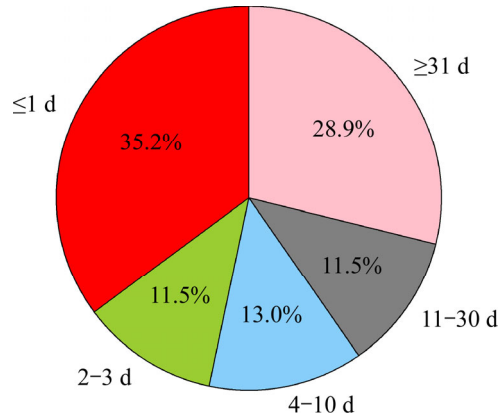


**Figure 7** Time used for closing Pull-Requests

Figure 7 shows that 40.4% PRs are closed for more than 10 d. Moreover, 28.9% PRs are closed for more than a month. That is to say, it is a long time for reviewers to finish reviewing the code change. However, there are two possible reasons for this situation. One is the start time that reviewing the PR is late and the other is that the reviewing process is time-consuming. Therefore, we calculate the standard deviation of the first three reviewers' start time for each PR, which is shown in Figure 8.

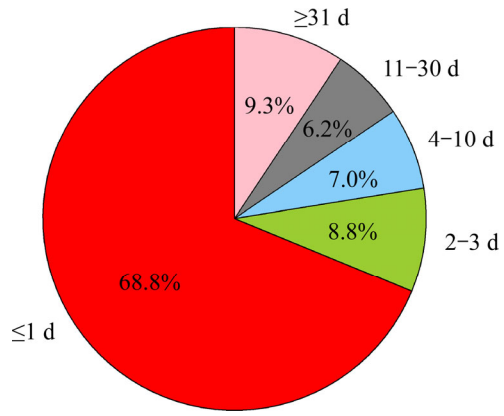


**Figure 8** Standard deviation of first three reviewers' start time

From Figure 8, we can see that for 68.8% PRs, the standard deviation of the first three reviewers' review time is less than one day, which means that the time gap between each reviewer's participation is less than 1 d. Therefore, these PRs can attract reviewers to participate in a short time. However, there are still 32.2% PRs which cannot attract enough PRs in a short time. For them, it is important to apply code reviewer recommendation algorithm so that different reviewers can focus on them at the same time. For all the PRs, code reviewer recommendation can help to find suitable reviewers and shorten the time for reviewing theoretically.

## 4 RevRec: Our approach

In this section, we will firstly present some concepts about the relationship between developers and PRs, and then describe the two-layer recommendation model in detail.

### 4.1 Concepts

There are different relationships between each developer and PR in an open source repository.

*Participate*: If the developer has commented to the PR, then we say that they have the *participate* relationship.

*Not participate*: On the contrary, the relationship between developer and PR is not *participate* if he/she has not commented below the PR.

*Technically participate*: If the developer has participated the PR by commenting below the change of code, then we treat their relationship as *technically participate*.

*Managerially participate*: If the developer has participated the PR by just communicating with the developers or commenting to the PR below the description, then we treat the relationship as *managerially participate*.

We can see that *technically participate* and *managerially participate* are subsets of *participate* relationship.

### 4.2 Workflow of RevRec

For our recommendation model, there are two layers which are used for checking the relationship between each developer and the target PR. The workflow can be seen in Figure 9.

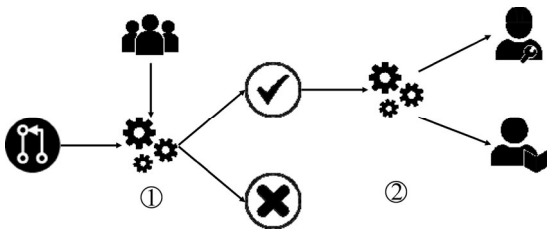


**Figure 9** Workflow of RevRec

Before doing the first step of classification, we pretreat the candidate set by filtering those developers without much experience of reviewing

code. Here we calculate developers' experience according to their times of reviewing. If one has one or more times of code reviewing experiences, we consider that he/she has the ability to review the target PR. If not, we filter the developer from the candidate set.

Firstly, as shown by ① in Figure 9, when the target PR comes, the first layer of our recommendation model checks whether each developer will participate in the PR. According to the relationship of each candidate and the target PR, we recommend the top 10 candidates as the reviewer of the target PR and treat them as having the *participate* relationship and for other candidates, we treat them as *not participate* candidates, and remove them from the result set.

Secondly, as shown by ② in Figure 9, for the second layer of our recommendation model, it checks whether each recommended reviewer will *technically participate* or *managerially participate* in the target PR. We will recommend reviewers according to the rank result of the first layer. Those who are classified as *technically participate* reviewers will be recommended as technical reviewers. Meanwhile, those who are classified as *managerially participate* reviewers are treated as the recommendation result of managerial reviewers.

From the workflow, we can see that RevRec consists of a heuristic recommendation algorithm and a classifier which belong to two layers respectively. The first layer is to get those developers who will review the PR and the other layer is to decide to what extent will the developer participate in the reviewing process, technically or managerially.

### 4.3 First layer

For the first layer of RevRec, we need to firstly generate a recommendation model for recommending suitable developers to review PRs. From YU's work [7], we find that there have already been many kinds of code reviewer recommendation methods, including IR-based approach, path similarity based approach. In this section, we will describe different approaches in detail and come up with our new code reviewer recommendation algorithm. By combining different methods, we can obtain a hybrid approach.

4.3.1 IR-based reviewer recommendation

One popular bug assignment algorithm is the IR (information retrieval)-based method [20], which aims to matching the technical focus of developers and bug reports. This algorithm is also suitable for code reviewer recommendation according to YU's work [7].

Firstly, we generate the technical terms of each PR in the project based on its title and description. By removing the stop words pre-defined before this process, we get the technical focus of each PR. All the technical terms can form into a corpus.

Secondly, we generate the term vector of each PR according to TF-IDF algorithm (Eq. (1)), where each term can be calculated by this method according to different PRs.

$$TF-IDF(t,pr)=\frac{freq(t,pr)}{\sum_{t'\in T_{pr}} freq(t',pr)}\times \lg\left(\frac{\sum_{pr'\in PR}\sum_{t'\in T_{pr'}} freq(t',pr)}{\sum_{pr'\in PR} freq(t,pr')+1}\right) \quad (1)$$

where $t$ represents a specific term; $freq(t, pr)$ means the number of times that term $t$ occurs in Pull-Requests $pr$; $T_{pr}$ represents all the technical terms in $pr$.

Thirdly, when a test PR comes, we calculate the relationship between each PR using the cosine similarity (Eq. (2)). The vector of each PR is formed by the technical terms in the whole corpus.

$$\text{similarity}(pr,pr')=\frac{v_{pr}\cdot v_{pr'}}{|v_{pr}||v_{pr'}|} \quad (2)$$

After that, we summarize the relationship between each PR that a developer has reviewed before and the target PR. And we get the relationship between the developer and the target PR (Eq. (3)).

$$\text{relation}(d,pr)=\sum_{pr'\in PR_d}\text{similarity}(pr,pr') \quad (3)$$

where $d$ represents a developer; $PR_d$ means the PR set that the developer $d$ has reviewed before.

Therefore, when a target PR comes, we calculate the relationship between each developer and the target project. Then rank the values in descending order, and recommend the top several results.

4.3.2 FL-based recommendation

Another popular code reviewer recommendation algorithm is the FL (file location)-based method [2]. This algorithm is based on the assumption that similar modules of a software project are located in the same directory. Thus,

developers can review the source code located in the same directory that they have committed to. The process of the method is shown as below.

Firstly, generate the related files that a developer commit to. When the target PR comes, the model firstly generates the PR's related file directories, i.e., the file directories that the submitter commits to. These file directories are treated as the PR's related module directories.

Secondly, after selecting the candidate reviewers of the target PR, we calculate the relationship between this PR and each candidate. As we all know that each candidate has committed to or reviewed the same or similar files of this module before the submission of the target PR. Thus, we can obtain the PR and candidate relationship by summarizing the relationship of each similar file (Eq. (4)). $F_{pr}$ means the related files that the PR changed, and similarity($f$, $f'$) represents the relationship between two files.

$$\text{relation}(d, pr) = \sum_{pr':PR_d} \sum_{f':F_{pr'}} \sum_{f:F_{pr}} \text{similarity}(f, f') \tag{4}$$

For the similarity of file directory, it uses the string comparison technique, which can be seen in Ref. [2].

Finally, after getting the relationship of each candidate and the target PR, we get the top $n$ candidates as the recommendation result.

4.3.3 Expertise based recommendation

For IR-based algorithm, it focuses on generating the technical terms of each PR, which means that different PRs tend to focus on different technical parts of an open source project. For FL-based method, it is based on an assumption that different modules are located in different directories in a project. However, these two methods just focus on the coarse-grained information of PRs. Actually, PRs are related to code changes of projects, which means that the change of code is more related to the PR. For code reviewers, it is more important to understand the code changes deeply. That is to say, the code changes that developers focus on can directly measure their technical focus. Therefore, we propose a new algorithm which measures developers' expertise towards a specific PR.

Before presenting the recommendation method, we firstly talk about the forgetting curve, which is used to describe the decline of memory relation in time. It can be calculated by Eq. (5).

$$retrievability(t, s) = \mathrm{e}^{-\frac{t}{s}} \tag{5}$$

where $t$ means the time after creating the memory and $s$ represents the memory strength. In our problem, when a reviewer reviews or submits a piece of code, we consider him/her as having this kind of memory. As time goes by, the developer's expertise towards that piece of code becomes lower. And Eq. (5) is used to calculate the remaining expertise.

When a developer reviews or submits PRs, we consider that he/she gains the capacity of the related technique. However, different PRs will cover different parts of related files, i.e., different lines of code. Meanwhile, the review or submit time of different PRs is different. Therefore, in order to calculate developers' expertise towards different PRs' related files, we need to take the lines of code and the correlation time into consideration. The expertise value for a developer to a source code file is shown in Eq. (6), in which $RT(d, pr)$ means the time that the developer submits or reviews the PR. $loc(pr, f)$ represents the lines of code that the PR's related file $f$ changed. We set the memory strength to 1, ignoring the differences between developers.

$$E(d, f) = \sum_{pr:\{PR_d \cap PR_f\}} (retrievability(\max\{kT(d, pr)\}, 1) \times loc(pr, f)) \tag{6}$$

For the relationship between candidates and the target PR, the result is calculated by summarizing the expertise of the candidate towards each related file of the target PR. The equation is shown as below (Eq. (7)).

$$\text{relation}(d, pr) = \sum_{f:F_{pr}} E(d, f) \tag{7}$$

Because we take users' review and submit time into consideration, the rank of users' expertise is more reliable. If one is very active in the past, but less active nowadays, the value of his/her expertise will not be that big when considering about a PR that was submitted recently. Therefore, we can say that this method can solve the problem (active reviewers tend to be recommended by traditional code reviewer recommendation methods) that we mentioned in Section 3.1 to some extent.

4.3.4 Hybrid recommendation approach

Because the above three methods focus on different parts of PRs in open source projects. In order to get a better recommendation result, we synthesize the three recommendation algorithms mentioned above by combining their

recommendation results. The contribution of each result is calculated by its rank value (Eq. (8)). By summarizing different results of each method, we get the final score of each reviewer. After sorting the result in descending order, we get the top $n$ results as the recommendation result of the hybrid approach. The equation of each reviewer's score is shown in Eq. (9).

$$score(d, pr) = 1/rank(d, pr) \tag{8}$$

$$score_{hybrid}(d, pr) = score_{IR}(d, pr) + score_{revfinder}(d, pr) + score_{expertise}(d, pr) \tag{9}$$

Through our validation, we find that the hybrid recommendation result outperforms the other three algorithms when recommending reviewers regardless of their activeness, which will be shown in Section 6.1.

### 4.4 Second layer

In this part, we will present the details for training the second layer of our recommendation model. The training process is shown in Figure 10.

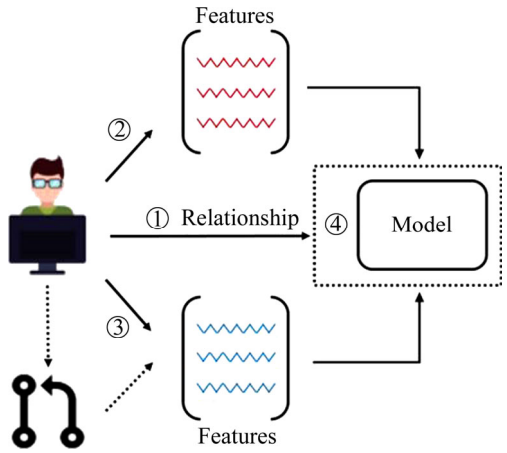


**Figure 10** Training process of RevRec

As Figure 10 shows, there are three steps for training a classifier for *technically participate* and *managerially participate* reviewers, which are shown as follows:

Firstly, as shown by ① in Figure 10, we generate all the developers and Pull-Requests in an open source repository, and check the relationship between each developer and Pull-Request (*technically participate* or *managerially participate*).

Secondly, as shown by ② in Figure 10, generate the features of each candidate, which represent his/her technological level.

Thirdly, as shown by ③ in Figure 10, generate the relative features of candidates towards each PR, which represents candidates' experience for technical or managerial review.

Finally, as shown by ④ in Figure 10, put the training features and their related relationship into a SVM classifier to produce the second-layer classifier.

For different open source projects, we need to generate different classifiers based on the above training process.

For the second layer, we need to generate the absolute and relative features of developers in GitHub. Here in this part, we will describe each feature in detail.

**EXPERTISE**: This feature is to measure the familiarity of each developer with the target PR. The detail information can be seen in Section 4.3.3.

**NTR**: Number of technical reviews. This feature is to calculate the number of times that a developer technically reviews before the target PR created, which is used to measure the probability of the user to become a *technically participate* reviewer.

**NMR**: Number of managerial reviews. This feature is opposite to **NTR**, which is used to describe the probability of the user to become a *managerially participate* reviewer.

In general, these three features can differentiate different kinds of reviewers to some extent. The result of the classifier can be seen in Section 6.2.

There is something that need to be mentioned. There are many reviewers who have no experience in technical review, therefore, we just treat them as managerial reviewers. For the training process, these developers will not be included in the model. When testing the model, those reviewers with no experience for technical review before the creation of the target PR will be automatically classified as managerial developers. We make tuples for each pull request like (*pr*, *reviewer*, *review_type*). A *pr* corresponds to multiple reviewers and a reviewer has two kinds of *review_type*: *technically participate* and *managerially participate.*

## 5 Experiment setting

### 5.1 Dataset

In this section, we will present the dataset that

we use to train and test the RevRec model. We test our recommendation model on 2 popular GitHub projects, namely Ruby on Rails and Angular.js.

We obtain all the PR information including PR description, PR reviewers, PR related commit messages and etc. Using the public API provided by GitHub, we get 18527 PRs for Ruby on Rails and 7403 PRs for Angular.js from the creation of the project till March 16th, 2017. The dataset can be seen in Table 1.

**Table 1** Dataset in GitHub

| Project | PR number | Test PR number | Reviewer number |
|---|---|---|---|
| Ruby on Rails | 18572 | 2284 | 3410 |
| Angular.js | 7403 | 685 | 1508 |

For the first layer of our recommendation model, we select the PRs created between February 2016 and February 2017 which also have more than 1 reviewer as the test PRs. For each test PR, we recommend reviewers among all the developers who have appeared before the creation of the target PR.

For the second layer, we also use the same set of test PRs. Different from the first layer, we differentiate the technical reviewers and managerial reviewers regardless of those reviewers who focus on these two types of reviews.

### 5.2 Experiment metrics

For the validation of our experiments, precision and recall are not suitable to measure the model's performance. For the false positives of our recommendation result (the recommended developers who have not reviewed the PR), they can still review the PR sometime in the future. Even though the PR is closed, it can still be reopened. Therefore, the precision and recall values will change according to the test time frame that we select.

In order to validate the performance of our recommendation model, we measure the accuracy value. The equation is shown as below,

$$\text{accuracy} = \frac{\sum_{i=1}^{|PR_s|} hasTrue(PR_{si})}{|PR_s|} \tag{10}$$

where *PRs* means the set of test PRs in the target project, and *hasTrue* is a function which checks whether a PR has got a true code reviewer.

## 6 Results analysis and discussion

In this section, we do contrast experiments about the accuracy of different code reviewer recommendation methods, and present the performance of the second layer classifier.

For the first layer, we compare our expertise based method and the hybrid method with the pre-existing methods (IR-based method [20] and Revfinder [2]). The IR-based method is to find related code reviewers according to their technical focus, which is described in detail in Section 4.3.1. Even though it is used for bug assignment firstly, it is also suitable for code reviewer recommendation. The Revfinder method is to find code reviewers based on the similarity of file path where code change locates. For the detail of Revfinder, see Section 4.3.2.

For the second layer, we present the accuracy value of our fine-grained code reviewer classifier.

### 6.1 Recommendation results

For this part, we firstly make comparison of different methods regardless of reviewers' activeness to see which method performs the best when recommending 1–10 developers. The result is shown in Figure 11.

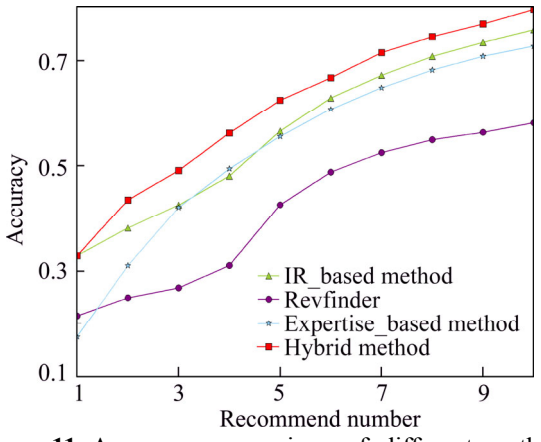


**Figure 11** Accuracy comparison of different methods considering all PRs of 2 projects

From Figure 11, we can see that, when recommending 1–10 reviewers, the accuracy of the hybrid approach is the best compared with those three separate methods, which ranks from 32.9% to 79.6%. That is to say, these three methods can make

up for each other and reach the best result.

However, as we have said in Section 3.1, for active developers, they will attend PRs as many as possible. Therefore, there is no need to recommend these developers.

Here we do an experiment to test the performance of each method when recommending reviewers regarding of their activeness. We remove the most active reviewers in the two projects respectively and calculate the accuracy ("rafaelfranca" and "senny" in Ruby on Rails, "petebacondarwin" and "mary-poppins" in Angular.js). The results of the two projects can be seen in Figures 12(a) and (b).

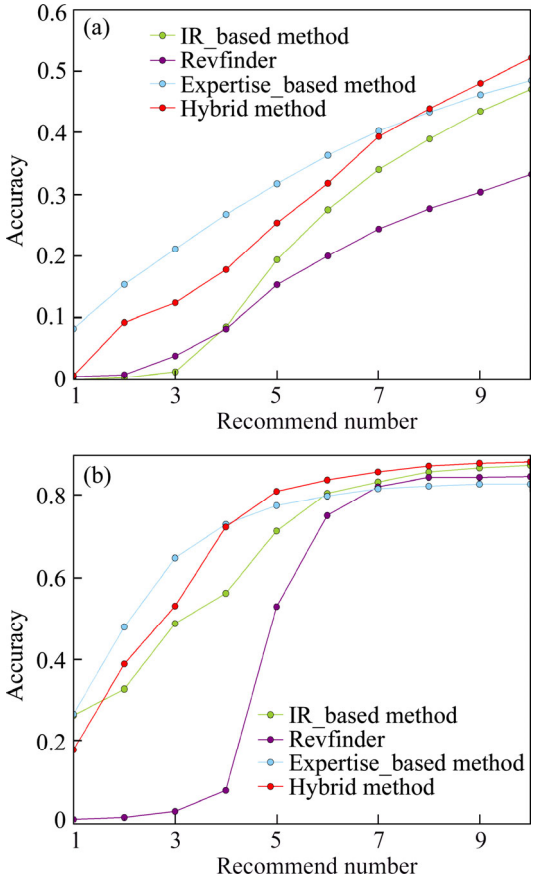


**Figure 12** Accuracy comparison after removing active reviewers: (a) Ruby on Rails; (b) Angular.js

From these figures, we can see that the expertise based recommendation method performs the best when recommending less than 7 reviewers for Ruby on Rails and less than 4 reviewers for Angular.js. That is to say, comparing to expertise based method, IR-based and FL-based recommendation algorithms tend to recommend those active developers. After removing those active ones, the true recommendation results of IR-based and FL-based methods will be included in expertise based method. Therefore, the hybrid approach does not perform better than the expertise based method.

However, when recommending more than 7 reviewers for Ruby on Rails and more than 4 reviewers for Angular.js, the different true results of IR-based and FL-based methods emerge, which makes hybrid method perform the best.

### 6.2 Classification results

In order to test the result of the second layer classifier, we select the top 10 reviewers of the hybrid method in the first layer. Then we classify the result in the second layer to see whether he/she will participate in the technical or managerial review. For each classifier of the two open source projects, the accuracy result can be seen in Figure 13.

From the two figures, we can see that for Ruby on Rails project, the accuracy value is about 23.4% when recommending 10 reviewers, which means that 23.4% PRs find suitable technical or managerial reviewers. For Angular.js, the value is

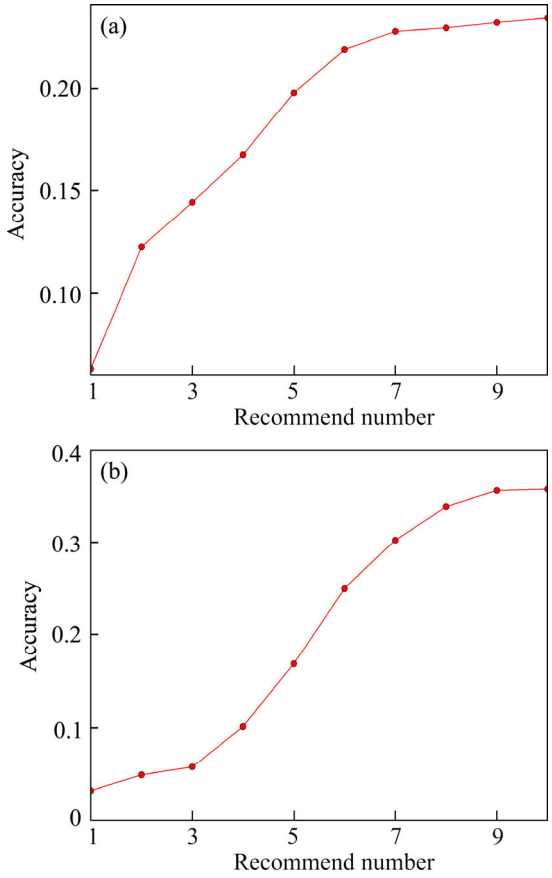


**Figure 13** Accuracy for second layer: (a) Ruby on Rails; (b) Angular.js

35.8%.

Even though the accuracy is not that good, we find that the accuracy values for the two projects are all bigger than 50% when we test the second layer using those truly participating reviewers, especially for Ruby on Rails, (the accuracy value is even bigger than 70%), which means that the second layer classifier does perform effectively.

Therefore, in order to improve RevRec, we need to firstly improve the firstly layer so that there can be enough truly participating reviewers for the second layer to classify.

**Table 2** Accuracy of second layer for truly participating reviewer's viewers

| Project | Accuracy/% |
|---|---|
| Ruby on rails | 72.8 |
| Angular.js | 58.6 |

### 6.3 Threats to validity

There are some threats to validity of our experiment which may affect the results.

One is that the time limitation of the review process. For reviewers who have not reviewed the target PR will probably review sometime in the future. This may lead to a lower value of accuracy.

Meanwhile, for *technically participate* reviewers, there may be some other reviewers who just *managerially participate* in the PR, but mention the code change in the comment. Therefore, the classification of these two categories still needs to be improved.

## 7 Conclusion and future work

This study aims to deeply analyze the pre-existing code reviewer recommendation methods and propose an effectively improved algorithm. After that, we develop a fine-grained multi-layer code reviewer recommendation model for open source projects based on traditional machine learning classifiers. We describe our model in detail and carry out contrast experiments on two popular projects in GitHub. The result shows that RevRec can effectively recommend those non-active reviewers. Meanwhile, RevRec can differentiate reviewers' contribution towards each PR very well (*technically participate* or *managerially participate*), which means that for different requirements of PRs, our model can recommend different kinds of developers to solve the problem. However, there are still some limitations of our work.

For expertise based recommendation method, we have not taken different developers' memory strength into consideration. However, as we all know that different people have different memories in reality. What's more, when calculating reviewers' expertise, we just sum up the weighted lines of code without taking the same code pieces into consideration. Therefore, we need to deal with these problems and improve this method for future work.

For the second-layer recommendation, we do not consider the recommendation results of the first layer, because we are afraid that the first layer result will affect the second layer. Besides, for feature selection, we just select possible features for the recommendation model heuristically, thus RevRec may not cover all the possible information for developers and PRs. Therefore, the recommendation result may be much better if we add or change some features of our model. Next, we will analyze the relevance of each feature towards the model, and select the most suitable ones.

In our model, we generate two categories, the technically participate relation and the managerially participate relation. However, there are still many differences in each category. For testing and changing code, they are all treated as technically related processes. Managerial participation also has many types, including encouraging the PR author, asking questions about the PR, giving suggestions to the PR and so on. For the future work, we will refine the recommendation model for finer granularity classification.

## References

[1] BOEHM B, ROMBACH H D, ZELKOWITZ M V. Software defect reduction top 10 list [M]// Foundations of Empirical Software Engineering: the Legacy of Victor R. Basili, Springer, Verlag New York, Inc., 2005.

[2] THONGTANUNAM P, TANTITHAMTHAVORN C, KULA R G, YOSHIDA V, IIDA H, MATSUMOTO K I. Who should review my code? A file location-based code-reviewer recommendation approach for modern code review [C]// Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on. IEEE, 2015: 141–150.

[3] KOLLANUS S, KOSKINEN J. Survey of software inspection research [J]. The Open Software Engineering Journal, 2009, 3(1): 15–34.

[4] RIGBY P C, STOREY M A. Understanding broadcast based

peer review on open source software projects [C]// Proceedings of the 33rd International Conference on Software Engineering. ACM, 2011: 541–550.

[5] BALACHANDRAN V. Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation [C]// Software Engineering (ICSE), 2013 35th International Conference on. IEEE, 2013: 931–940.

[6] JEONG G, KIM S, ZIMMERMANN T, YI K. Improving code review by predicting reviewers and acceptance of patches [M]// Research on Software Analysis for Error-free Computing Center Tech-Memo (ROSAEC MEMO 2009-006), 2009: 1–18.

[7] YU Y, WANG H, YIN G, WANG T. Reviewer recommendation for pull-requests in GitHub: What can we learn from code review and bug assignment? [J]. Information and Software Technology, 2016, 74: 204–218.

[8] BROY M, DENERT E. Pioneers and their contributions to software engineering [M]. Berlin Heidelberg, Springer: 2001.

[9] MCLNTOSH S, KAMEI Y, ADAMS B, HASSAN A E. The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects [C]// Proceedings of the 11th Working Conference on Mining Software Repositories. ACM, 2014: 192–201.

[10] MCINTOSH S, KAMEI Y, ADAMS B, HASSAN A E. An empirical study of the impact of modern code review practices on software quality [J]. Empirical Software Engineering, 2016, 21(5): 2146–2189.

[11] BOSU A, CARVER J C. Peer code review to prevent security vulnerabilities: An empirical evaluation [C]// Software Security and Reliability-Companion (SERE-C), 2013 IEEE 7th International Conference on. IEEE, 2013: 229–230.

[12] MORALES R, MCLNTOSH S, KHOMH F. Do code review practices impact design quality? a case study of the qt, vtk, and itk projects [C]// Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on. IEEE, 2015: 171–180.

[13] MUKADAM M, BIRD C, RIGBY P C. Gerrit software code review data from android [C]// Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on. IEEE, 2013: 45–48.

[14] GOUSIOS G, ZAIDMAN A, STOREY M A, van DEURSEN A. Work practices and challenges in pull-based development: the integrator's perspective [C]// Proceedings of the 37th International Conference on Software Engineering: Volume 1. IEEE, 2015: 358–368.

[15] PHAM R, SINGER L, LISKIN O, FIGUEIRAFILHO F, SCHNEIDER K. Creating a shared understanding of testing culture on a social coding site [C]// Software Engineering (ICSE), 2013 35th International Conference on. IEEE, 2013: 112–121.

[16] ZHU J, ZHOU M, MOCKUS A. Effectiveness of code contribution: From patch-based to pull-request-based tools [C]// Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, 2016: 871–882.

[17] YU Y, WANG H, YIN G, LING C X. Reviewer recommender of pull-requests in GitHub [C]// Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on. IEEE, 2014: 609–612.

[18] YANG C, ZHANG X, ZENG L, FAN Q, YIN G, WANG H. An empirical study of reviewer recommendation in pull-based development model [C]// Proceedings of the 9th Asia-Pacific Symposium on Internetware. ACM, 2017: 14.

[19] JEONG G, KIM S, ZIMMERMANN T. Improving bug triage with bug tossing graphs [C]// Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering. ACM, 2009: 111–120.

[20] KAGDI H, POSHYVANYK D. Who can help me with this change request? [C]// Program Comprehension, 2009. ICPC'09. IEEE 17th International Conference on. IEEE, 2009: 273–277.

[21] BHATTACHARYA P, NEAMTIU I. Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging [C]// Software Maintenance (ICSM), 2010 IEEE International Conference on. IEEE, 2010: 1–10.

[22] CANFORA G, CERULO L. Supporting change request assignment in open source development [C]// Proceedings of the 2006 ACM Symposium on Applied Computing. ACM, 2006: 1767–1772.

[23] LINARES-VÁSQUEZ M, HOSSEN K, DANG H, KAGDI H, GETHERS M, POSHYVANYK D. Triaging incoming change requests: Bug or commit history, or code authorship? [C]// Software Maintenance (ICSM), 2012 28th IEEE International Conference on. IEEE, 2012: 451–460.

[24] JONSSON L, BORG M, BROMAN D, SANDAHL K, ELDH S, RUNESON P. Automated bug assignment: Ensemble-based machine learning in large scale industrial contexts [J]. Empirical Software Engineering, 2016, 21(4): 1533–1578.

[25] TAMRAWI A, NGUYEN T T, AL-KOFAHI J M, NGUYEN T N. Fuzzy set and cache-based approach for bug triaging [C]// Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering. ACM, 2011: 365–375.

[26] XUAN J, JIANG H, REN Z, YAN J, LUO Z. Automatic bug triage using semi-supervised text classification [J]. arXiv: 1704.04769, 2017.

[27] HAN D, ZHUO H, XIA L, LI L. Permission and role automatic assigning of user in role-based access control [J]. Journal of Central South University, 2012, 19(4): 1049–1056.

[28] ANVIK J, HIEW L, MURPHY G C. Who should fix this bug? [C]// Proceedings of the 28th International Conference on Software Engineering. ACM, 2006: 361–370.

[29] SHOKRIPOUR R, ANVIK J, KASIRUN Z M, ZAMANI S. Improving automatic bug assignment using time-metadata in term-weighting [J]. IET Software, 2014, 8(6): 269–278.

[30] ZHANG X, WANG T, YIN G, YANG C, YU Y, WANG H. DevRec: A developer recommendation system for open source repositories [C]// International Conference on Software Reuse. Springer, 2017: 3–11.

[31] ZHANG X, WANG T, YIN G, YANG C, WANG H. Who will be interested in? A contributor recommendation approach for open source projects [C]// Proceedings of the

29th International Conference on Software Engineering & Knowledge Engineering, 10.182931SEKE2017-067.

[32] THONGTANUNAM P, KULA R G, CRUZ A E C, YOSHIDA N, IIDA H. Improving code review effectiveness through reviewer recommendations [C]// Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering. ACM, 2014: 119–122.

[33] RAHMAN M M, ROY C K, COLLINS J A. CoRReCT: Code reviewer recommendation in GitHub based on cross-project and technology experience [C]// Software Engineering Companion (ICSE-C), IEEE/ACM International Conference on. IEEE, 2016: 222–231.

[34] YU Y, WANG H, YIN G, LING C X. Who should review this pull-request: Reviewer recommendation to expedite crowd collaboration [C]// Software Engineering Conference (APSEC), 2014 21st Asia-Pacific. IEEE, 2014, 1: 335–342.

[35] ZANJANI M B, KAGDI H, BIRD C. Automatically recommending peer reviewers in modern code review [J]. IEEE Transactions on Software Engineering, 2016, 42(6): 530–543.

[36] XIA Z, SUN H, JIANG J, WANG X, LIU X. A hybrid approach to code reviewer recommendation with collaborative filtering [C]// 2017 6th International Workshop on Software Mining (Software Mining). IEEE, 2017: 24–31.

**(Edited by YANG Hua)**

## 中文导读

### RevREC：一个基于 Pull-Request 开发模型的双层审阅人推荐算法

**摘要：**代码审查是减少代码缺陷和提高软件质量的重要过程。在像 GitHub 这样的社交编码社区，由于每个人都可以提交 Pull-Request，所以代码审查扮演着比以往更重要的角色，而且这个过程非常耗时。因此，寻找并推荐正确的评审人员来应对新兴的 Pull-Request 成为一项重要任务。然而，目前大部分的研究主要集中在评估人员是否参与，并没有对人员参与的类型进行区分。在本文中，我们开发了一个两层审阅人推荐模型，从技术和管理角度为 GitHub 项目中的 Pull-Request(PR)推荐审阅人。对于第一层，我们根据混合推荐方法推荐合适的审阅人对目标 PR 进行审阅。对于第二层，在从第一层获得推荐结果之后，我们指定被推荐的审阅人是技术还是管理上参与审阅过程。我们在 GitHub 的两个热门项目上进行了实验，并使用 2016 年 2 月至 2017 年 2 月期间创建的 PR 来测试该方法。结果显示，我们的推荐模型的第一层比以前的工作表现得更好，第二层可以有效地区分参与类型。

**关键词：**Pull-Request；代码审阅人推荐；GitHub；开源社区