Information and Software Technology 74 (2016) 204–218

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Reviewer recommendation for pull-requests in GitHub: What can we learn from code review and bug assignment?

Yue Yu*, Huaimin Wang, Gang Yin, Tao Wang

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

ARTICLE INFO

Article history:
Received 7 April 2015
Revised 20 December 2015
Accepted 11 January 2016
Available online 18 January 2016

Keywords:
Pull-request
Reviewer recommendation
Social network analysis

ABSTRACT

Context: The pull-based model, widely used in distributed software development, offers an extremely low barrier to entry for potential contributors (anyone can submit of contributions to any project, through *pull-requests*). Meanwhile, the project's core team must act as guardians of code quality, ensuring that pull-requests are carefully inspected before being merged into the main development line. However, with *pull-requests* becoming increasingly popular, the need for qualified reviewers also increases. GitHub facilitates this, by enabling the crowd-sourcing of *pull-request* reviews to a larger community of coders than just the project's core team, as a part of their *social coding* philosophy. However, having access to more potential reviewers does not necessarily mean that it's easier to find the right ones (the "needle in a haystack" problem). If left unsupervised, this process may result in communication overhead and delayed pull-request processing.

Objective: This study aims to investigate whether and how previous approaches used in bug triaging and code review can be adapted to recommending reviewers for *pull-requests*, and how to improve the recommendation performance.

Method: First, we extend three typical approaches used in bug triaging and code review for the new challenge of assigning reviewers to *pull-requests*. Second, we analyze social relations between contributors and reviewers, and propose a novel approach by mining each project's *comment networks* (CNs). Finally, we combine the CNs with traditional approaches, and evaluate the effectiveness of all these methods on 84 GitHub projects through both quantitative and qualitative analysis.

Results: We find that *CN*-based recommendation can achieve, by itself, similar performance as the traditional approaches. However, the mixed approaches can achieve significant improvements compared to using either of them independently.

Conclusion: Our study confirms that traditional approaches to bug triaging and code review are feasible for *pull-request* reviewer recommendations on GitHub. Furthermore, their performance can be improved significantly by combining them with information extracted from prior social interactions between developers on GitHub. These results prompt for novel tools to support process automation in social coding platforms, that combine social (*e.g.,* common interests among developers) and technical factors (*e.g.,* developers' expertise).

1. Introduction

The pull-based development model [1,2], used to integrate incoming changes into a project's codebase, is one of the most popular contribution models in distributed software development [3]. External contributors can propose changes to a software project without the need to share access to the central repository with the core team. Instead, they can create a *fork* of the central repository, work independently and, whenever ready, request to have their changes merged into the main development line by submitting a *pull-request*. The pull-based model democratizes contributing to open source projects supported by the distributed version control system (*e.g.,* `git` and `Mercurial`): anyone can contribute to any repository *via* submitting *pull-requests*, even if they are not part of the core team.

* Corresponding author. Tel.: +86 13627319266.
E-mail addresses: yuyue@nudt.edu.cn (Y. Yu), hmwang@nudt.edu.cn (H. Wang), yingang@nudt.edu.com (G. Yin), taowang2005@nudt.edu.com (T. Wang).

http://dx.doi.org/10.1016/j.infsof.2016.01.004
0950-5849/

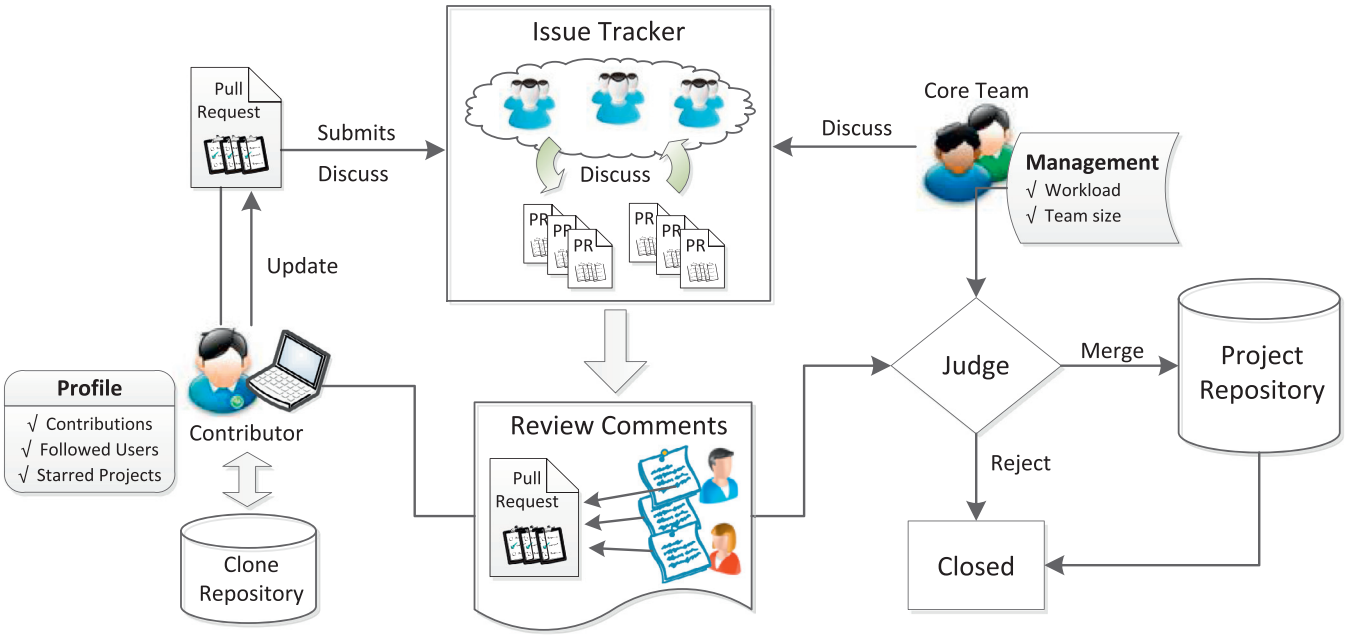


Fig. 1. Overview of the socialized pull-request mechanism in GitHub.

Currently, the pull-based model is widely adopted by open source projects hosted on the modern collaborative coding (or social coding [4]) platforms such as BitBucket, Gitorius, and GitHub. Especially, on GitHub, the pull-request mechanism is integrated with multiple social media features, so users can freely *watch* the development activities of popular repositories, *follow* distinguished developers and *comment* others' contributions. The socialized pull-request paradigm offers greater conveniences for collaborative development compared to the traditional development models [5,6], *e.g.,* email-based patching. A typical contribution process [7] in GitHub involves following steps, as summarized in Fig. 1. First of all, a potential contributor $\mathcal{D}$ locates an interesting project repository $\mathcal{R}$, often by *following* some well-known developers and *watching* their projects. Next, she/he can *fork* this project, and contribute code locally, either by creating a new feature or fixing some bugs. When the work is done, $\mathcal{D}$ sends the changes from the forked repository to $\mathcal{R}$ packaged as a pull-request. Then, exploiting integration with the issue tracker, all developers of $\mathcal{R}$ have the chance to review that pull-request. They can freely discuss whether the project needs that feature, whether the coding style is conformant, or if the code quality is adequate, and potentially make suggestions using the social media features. Next, $\mathcal{D}$ can make changes to the pull-request, responsive to the reviewers' suggestions, perhaps triggering more reviews and discussion. Finally, a responsible manager of the core team (*i.e.,* integrator [3]) takes all the opinions of reviewers into consideration, and then merges or rejects that pull-request.

With the development of projects, the pull-request paradigm can be used in many scenarios beyond basic patch submission, *e.g.,* for conducting code reviews and discussing new features [3]. On GitHub alone, almost half of all collaborative projects use pull-requests [3], and this number is only expected to grow. In those projects, the absolute number of new pull-requests increases dramatically [8]. *Ruby on Rails*[1] (*i.e., rails*), one of the most popular projects on GitHub, receives upwards of three hundred new pull-requests each month. To ensure the project's quality, new pull-requests should be evaluated before eventually accepted. In large projects, the high volume of incoming pull-requests poses a serious challenge to project integrators. Currently, the management of pull-requests is identified as the most important project activity on GitHub [9].

To reduce human workload, GitHub synthesizes various external services to support the evaluation process of pull-request, *e.g.,* continuous integration [10,11] for automatic testing. However, on the one hand, the discussions among developers are still necessary to ensure high quality of code [3]. For example, one of typical responses in the survey [3] from integrators explains that "*we have a rule that at least 2 of the core developers must review the code on all pull-requests*". On the other hand, the responsiveness of reviewers is a key factor to affect the latency of pull-request evaluation [12]. According the survey [13], a lot of contributors (15% of the participants in the survey) complain that their pull-requests hardly get a timely feedback. Thus, if appropriate reviewers can be automatically recommended to new pull-requests, the efficiency of pull-based development model would be improved.

Fortunately, the recommendation approaches for coping with similar challenges of bug triaging and code inspections give us a great inspiration. In those contexts, the qualified developers (always core members) measured by their expertise are recommended to fix incoming bug reports or evaluate the quality of newly received code changes. However, unlike there, GitHub facilitates the evaluation process, by enabling the crowd-sourcing to a larger community of coders rather than just a project's core team, as a part of *social coding* philosophy. Pull-request reviewers can and do come from outside the project, because all the project watchers (even external developers) can received notifications at the time of new pull-requests creation. The outsider reviewers play important roles on the evaluation process, as they not only put forward some particular requirements for their own usage, but also apply pressure to the integrators in order to influence the final decision [14] (*i.e.,* reject, merge or reopen). Thus, it is very interesting to study the performance of traditional recommendation approaches transferring to pull-request model. Moreover, the outsiders are usually not responsible for the pull-requests, but join the discussion driven by their interest (*i.e.,* willingness). Hence, the

[1] https://github.com/rails/rails.

social factors (*e.g.,* prior social interactions) would be conducive to find potential reviewers in the new context (*i.e.,* pull-based development).

In this paper, we firstly review the existing work for bug assignment and code-reviewer recommendation, and extend three typical approaches based on the *Machine Learning* (ML), *Information Retrieval* (IR) and *File Location* (FL) technique respectively into the new challenge of pull-request assignment. Besides, we construct a novel social network called *Comment Network* (CN), and recommend potential reviewers based on it. Furthermore, we combine the comment network with the traditional approaches.

The key contributions of this paper include:

- We extend the traditional approaches used in bug triaging and code-reviewer recommendation to pull-request assignment. These approaches make a recommendation based on measuring the expertise of potential reviewers.
- We propose a novel *CN*-based approach by analyzing comment networks, which can capture common interests in social activities among developers.
- We combine the expertise factor with the common interest, and recommend appropriate reviewers for pull-requests using the mixed approaches.
- We present a quantitative and qualitative analysis to evaluate above approaches in a big dataset containing 84 projects of GitHub. We find that the lightweight approach based on comment networks can reach the same level of performance as traditional approaches; however, the mixed approaches represent significant improvements.

The remainder of this paper is structured as follows: Section 2 illustrates our motivation, related work and research questions. Section 3 presents different recommendation approaches in detail. Empirical analysis and discussion can be found in Sections 4 and 5. Finally, we draw our conclusions in Section 6. This paper is an extension of the paper [8] published in the *21st Asia-Pacific Software Engineering Conference* (APSEC 2014). Compared with the prior work, our new contributions are summarized as follows:

- We extend two approaches based on file-location and information retrieval to pull-request assignment. The file-location based approach [15] is one of state-of-art approaches for code-reviewer recommendation.
- We combine traditional approaches with comment networks, and make a comparison between the mixed approaches and the independent approaches.
- We evaluate all approaches using both quantitative and qualitative analysis in a new large dataset, which ensures the correctness and robustness of our findings.

2. Background and related work

2.1. Peer review of pull-request in GitHub

Pull-request is one of the primary methods for code contributions to a public repository. Both the core team and the users who have starred[2] the repository can mechanically receive notifications of new pull-requests. A pull-request can be manually assigned to one of core developers by integrators. The assignee is in charge of the review process, which is similar to the process in the bug tracking system. However, only a small part of pull-requests have been triaged in that way (only 0.89% of pull-requests are assigned in our previous study [7,8]). Compared to the manually assignment way, the *@mention* tool is more widely used among reviewers in the peer review process. If the @ tag is placed in front of a user's name, the corresponding developer will receive a special notification that he has been mentioned in the discussion. More potential reviewers, both core members and outside contributors, can be involved into the discussion process with the help of the *@mention* tool in GitHub. Reviewers can publish two types of comments: general comments about the overall contributions, and code-inline comments for the specific lines of changes.

We use a real pull-request, submitted by *rono23* from project *rails*, as an example, to explain how the review process works. To start with, as shown in the subgraph at the top right corner of Fig. 2, a core developer (integrator) called *rafaelfranca* is the first one to push forward the evaluation process. As he thought that *javan*'s work would be relevant to it, he mentioned (@) *javan* to ask for his opinion. At the second post, we can see that *javan* indeed joined the discussion. Meanwhile (see the subgraph at top left corner), other two users, *norman* and *pixeltrix*, voluntarily participated in the discussion and presented some important suggestions. Inspired by *norman*'s comment, *rafaelfranca* left his opinion following. Later (see the subgraph at the bottom), the author updated the pull-request by appending a new commit in terms of above suggestions, and then he mentioned (@) the two key reviewers for acknowledgement. Finally, the pull-request was merged into the master branch of *rails* by *rafaelfranca*.

As the example depicted above, apart from the responsible integrator (*i.e., rafaelfranca*), other reviewers commented on that pull-request indirectly, including *javan, norman* and *pixeltrix*. All their opinions affected the decision-making of that pull-request. As for *javan*, an external contributor,[3] he is informed by the integrator manually using the *@mention* tool. In practice, to accomplish this properly every time, the integrator have to not only check each new pull-request manually, but also know every contributor's work clearly. Supposing the reassignment can be changed to automation (at least generating a recommendation list), the workload of integrators obviously would be reduced. Likewise, if the unbidden reviewers, such as *norman* and *pixeltrix* above, do not notice that pull-request in time, the evaluation process would be longer. On the contrary, if the notifications can be automatically sent to the appropriate reviewers (*e.g.,* set up an robot-account that *@mentions* potential reviewers at the time of a new pull-request creation), the time for reviewers aware of their interested pull-requests would be reduced.

2.2. Effect of pull-request assignment

Our previous work [12] found that the evaluation latency of pull-requests is a complex issue, requiring many independent variables to explain adequately, *e.g.,* the size of pull-requests, the submitters' track records and number of discussion comments. During that work, we collected a large dataset[4] containing 650,007 pull-requests across 919 main-line projects in GitHub. Overall, the evaluation time (*i.e.,* latency) is fluctuated over a range of several minutes to half year (median: 14.78 h; mean: 363.94 h), as shown in Fig. 3. We used multiple linear regression to model the latency of evaluating pull-requests on GitHub. More details can be found in the paper [12]. Here, we show two evidences of the benefit from pull-request assignment. Controlling other social and technical factors (22 variables), we find that the factor of *first human response*, explaining over 10% of the variance, is highly significant and have a positive effect on the latency. It means the review process of pull-request would take longer if reviewers do not notice and reply in

[2] It is one of the GitHub-specific social media features, that allows anyone to flag, *i.e.,* star, projects they find interesting.

[3] GitHub uses two labels, owner and collaborator, to identify the core team.

[4] https://github.com/yuyue/pullreq_ci.

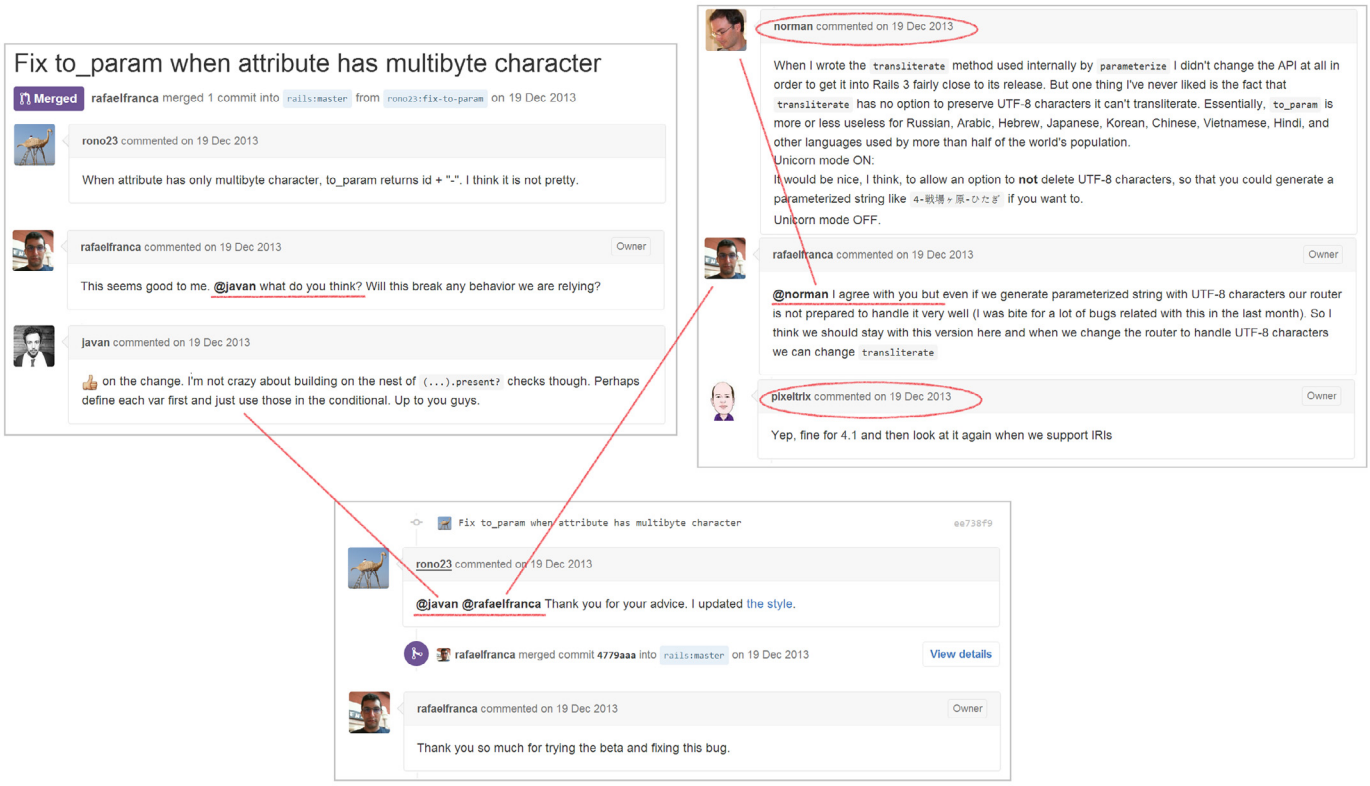


Fig. 2. Example of discussions among reviewers in a pull-request.

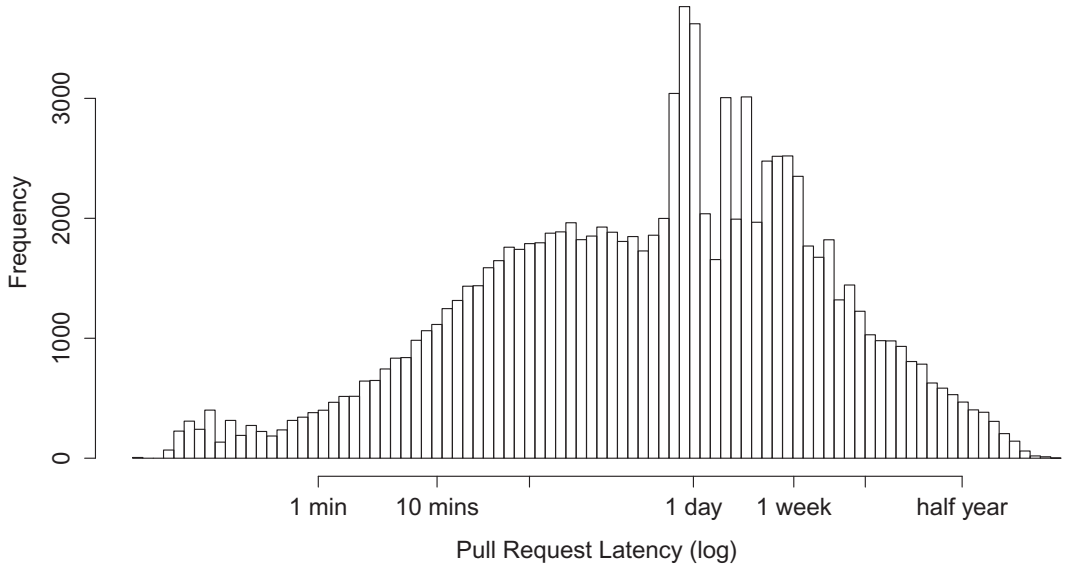


Fig. 3. Distribution of pull-request latency.

time. Even so, the first reviewer is more likely to be the responsible integrator, as shown in the example above. It still needs more time to wait for other potential reviewers. Additionally, the factor of *@mention* (*i.e.,* @ tags appeared in the title or description of a pull-request) is highly significant and have a negative effect on the latency. It clearly shows that pull-requests reassigned to reviewers early tend to be processed quicker.

From what has been discussed above, we argue that recommending highly relevant reviewers to pull-requests is a significant way to improve the efficiency of social coding.

2.3. Bug assignment and modern code review

Triaging bug reports [16,17] and organizing code review [18,19] are two importance practices of open source development. For some large projects, tens of new bug reports are submitted to the bug tracking system (*e.g., Bugzilla*). It is a labor-intensive and time-consuming task that assigning incoming bug reports to appropriate bug fixers. Likewise, there are a large number of new changes in modern code review system (*e.g., Gerrit*) waiting to review before the integration.

There are plenty of researches based on Machine Learning (ML) [16,20–23] and Information Retrieval (IR) [24–26] techniques to triage incoming bug reports (change requests). Cubranic et al. [22] propose a text classification method for bug assignment using the title, descriptions and keywords of bug reports. Anvik et al. [16] improve that ML-based approach by filtering out unfixed bug reports and inactive developers with a heuristic method. After the preprocessing stage, they can achieve the best precision levels of 64% on the Firefox project using the SVM classifier. Jeong et al. [23] find that many bugs have been reassigned to other developers, so they combine classifiers with tossing graphs to mine potential developers. Based on that, Bhattacharya et al. [20,21] introduce an approach adding fine-grained incremental learning and multi-feature tossing graphs to reduce tossing path lengths and improve prediction accuracy. Other researchers also utilize the IR techniques for automatic bug assignment. Canfora and Cerulo [24] use the textual descriptions of resolved reports to index developers as documents in an information retrieval system, and the new reports as a query. Kagdi [25] and Linares-Vasquez [26] extract the comments and identifiers from the source code and index these data by the Latent Semantic Indexing (LSI). For a new bug report, such indexes can be used to identify the fixers who have resolved similar bugs in history. Compared with above methods, Tamrawi et al. [27] build a model of the correlation between developers and technical terms to rank developers. They select a portion of recent bug fixers as the candidates first, then the most relevant developers are recommended by comparing their membership to the terms included in the new bug reports.

Similarly, to support modern code review [28], Jeong et al. [29] extract the features, such as file/module name, source lines of code, code keyword (*e.g., else* and *return*), and then build a prediction model using the Bayesian Network. Balachandran [30] proposes a tool called *Review-Bot* to predict the developers who have modified related code sections in source files as appropriate reviewers. Thongtanunam et al. [15,31] recommends code reviewers based on the measure of file path similarity. They consider that files located in similar folders should be reviewed by similar experienced code-reviewers. That state-of-art approach [15] is 4 times more accurate than the *Review-Bot*.

In the context of pull-based development, the pull-requests is managed in a similar manner as bug reports triaged in bug tracking system and code patches managed in code review system. Each pull-request includes a title and description summarized its contributions, the files it touched and actual code contents. Thus, we draw an analogy between pull-request assignment and bug triaging and code-reviewer recommendation. We ask:

RQ1: *How can traditional approaches used in bug triaging and code-reviewer recommendation be extended to pull-request assignment?*

In the process of bug triaging (similar to code inspection), each bug report is an obligatory task for the corresponding bug fixer who takes the responsibility for fixing the bug. In contrast, the evaluation process in pull-based development, except for project integrators, the reviewers voluntarily join the discussion, especially for external developers. It implies that some social factors concerning with willingness would be taken into consideration in our new context.

2.4. Social factors of distributed development

There are two fundamental elements on software development: technical and social component [32]. Social factors have a very strong impact on the success of software development endeavors and the resulting system [33]. Zhou and Mockus [34] find that the project's climate (*e.g.,* participation density) and the social interaction among developers are associated with the odds that a new joiner to become a long term contributor. Vasilescu et al. [35] argue that increased gender and tenure diversity are positive to improve the productivity of GitHub teams. Thung et al. [36] investigate the social network structure of GitHub, and use *PageRank* to identify the most influential developers and projects on GitHub. Yu et al. [37] present the typical social patterns among developers by mining the *follow-network* constructed from the *following* relations between developers on GitHub. Compared to the traditional model of collaboration in open source [6,38], social coding platforms (*e.g.,* BitBucket, Gitorius and GitHub) provide transparent work environments [9,39] for developers, which is benefit for innovation, knowledge sharing and community building. Apart from the technical factors (*e.g.,* code quality), the transparency allows project managers to evaluate pull-requests utilizing a wealth of social information in submitters' profile pages. Tsay et al. [40] prove that the strength of the social connection between submitters and core members has a strong positive association with the pull-requests acceptance using the logistic regression model. They also find that the core team would be polite to new contributors during the discussion [14].

Inspired from above findings, we consider that the review process of a pull-request is a kind of social activity depending on the discussions among reviewers on GitHub. Thus, prior human communication and social relations should be the key factors of reviewer recommendation. We ask:

RQ2: *Can suitable pull-request reviewers be recommended using only social relations? How effective is this approach compare to traditional approaches?*

However, in the contexts of bug triaging and code-reviewer recommendation, most of traditional approaches measure the developers' expertise first, and then rank the reviewer candidates based on it. Except for reviewer expertise, the social coding platforms make the relationships among developers transparent. In this paper, we analyze the historical comment relations in a given project, and propose a novel social network, *i.e., comment network*, to measure the common interests between reviewers and contributors. We ask:

RQ3: *Does combining prior social relations (RQ2) and review expertise (RQ1) result in increased performance in pull-request assignment?*

3. Recommending reviewer for pull-request

First of all, we extend the Machine Learning (ML), Information Retrieval (IR) and File Location (FL) based recommendation approaches used in bug triaging and code-reviewer recommendation, to pull-request assignment. Next, we propose a novel and lightweight approach (*i.e.,* comment network based approach) to recommend reviewers by mining the strength of prior social connections between the pull-request submitters and potential reviewers. Finally, we incorporate the comment network into traditional approaches.

3.1. Vector space model of pull-request

Each pull-request is characterized by its title and description, and can be labeled with the names of developers who had commented at least once on it. Then, all stop words and non-alphabetic tokens are removed, and remaining words are stemmed (Porter stemmer). We then use vector space model to represent each pull-request as a weighted vector. Each element in the vector is a term, and the value stands for its importance for the pull-request. The more a given word appears in a pull-request, the more important it is for that pull-request. Contrariwise, the more pull-requests a word appears in, the less useful it is to distinguish among these pull-requests. Term frequency–inverse document frequency (tf–idf)

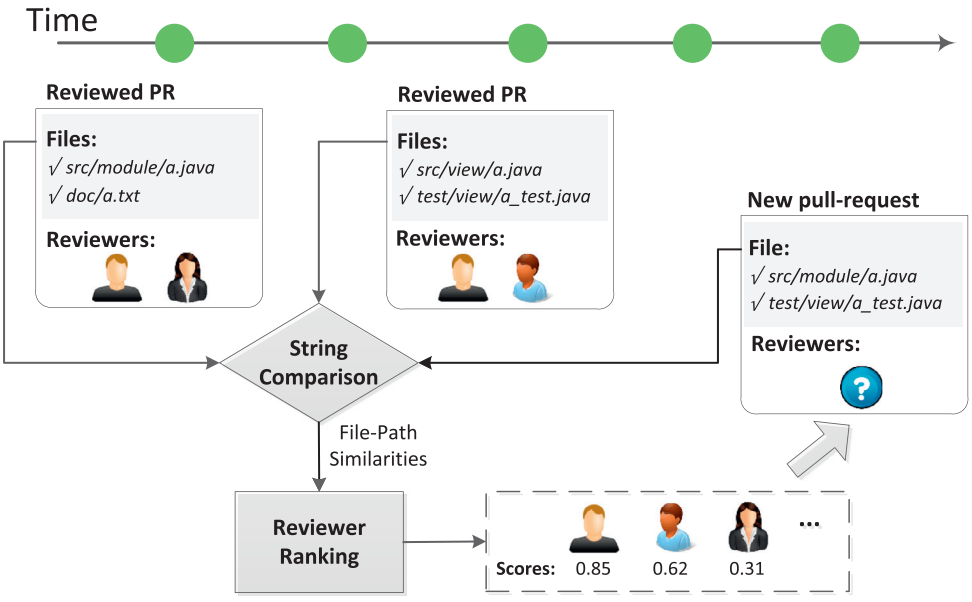


Fig. 4. File location-based reviewer recommendation for pull requests.

is utilized to indicate the value of a term, which can be calculated as Eq. (1).

$$tfidf(t, pr, PR) = log\left(\frac{n_t}{N_{pr}} + 1\right) \times log\frac{N_{PR}}{|pr \in PR : t \in pr|} \quad (1)$$

where t is a term, pr is a pull-request, PR is the corpus of all pull-requests in a given project, n_t is the count of appearance for term t in pull-request pr, and N_{pr} and N_{PR} are the total number of terms in pr and pull-requests in corpus, respectively.

3.2. SVM-based recommendation

In general, each pull-request is reviewed by several developers, so a reviewer recommender should provide more than one label for a pull-request testing instance. This is thus a multi-label classification problem [41] in Machine Learning (ML). We train the ML classifiers after preprocessing pull-requests by vector space model, as described in Section 3.1. For a new pull-request, a ranked list of recommended candidates is generated from top-1 to top-k according to the predicted probability of a given label by the ML classifiers, using the one-against-all strategy. When the probability values are equal, we rank the developers in terms of the number of pull-requests' comments that they had submitted to the given project. In this paper, we choose support vector machines (SVM) as our basis classifier, because it usually has been proved to be a superior classifier in developers' recommendation [16,26] than other common classifiers (*e.g.,* naive Bayes and decision tree). The SVM classifier is implemented by using *Weka* [42].

3.3. IR-based recommendation

We conceptualize new pull-requests as "bag of words" queries, and these queries and resolved pull-requests from our corpus are indexed by vector space model described at Section 3.1. We use *cosine similarity* to measure the semantic similarity [43,44]. Given a new pull-request, the ranked list of resolved pull-requests are generated according to similarity scores, as show in Eq. (2):

$$similarity(pr_{new}, pr_{rsl}) = \frac{\boldsymbol{v}_{new} \cdot \boldsymbol{v}_{rsl}}{|\boldsymbol{v}_{new}||\boldsymbol{v}_{rsl}|} \quad (2)$$

where pr_{new} is a new pull-request, pr_{rsl} is one of resolved pull-requests, $\boldsymbol{v}_{new}$ is the VSM of pr_{new} and $\boldsymbol{v}_{rsl}$ is the VSM of pr_{rsl}.

Next, we calculate the expertise score of a candidate reviewer by summing all the similarity scores of pull-requests that she/he has reviewed. Finally, the recommendation is based on the ranking of reviewers' expertise scores.

3.4. FL-based recommendation

As discussed in Section 2, many approaches [15,26,29,31] recommend reviewers by mining file-level (including source code) data compared to text-level (semantic) data. In this paper, we extend the state-of-art approach [15] for modern code review, *File Location* (FL) based approach, to recommend pull-request reviewers. The intuition is that files located at similar file system paths would be managed and reviewed by reviewers with similar experiences. Thus, we calculate the similarity scores between file paths involved in new pull-requests and file paths of completed pull-requests. Then we can assign reviewers to new pull-requests based on the file path similarities to previously assigned pull-requests.

Fig. 4 shows an example of FL-based recommendation. We firstly extract the file names from the new pull-request and old reviewed pull-requests. For each file, we use a slash character as a delimiter to split the file path into components (*i.e.,* each component is a word). Next, we can get common components that appear in two files by using the same string comparison technique[5] in paper [15]. Then, the file-path similarity score between two files can be calculated by dividing the number of common components by the maximum number of file path components. The similarity scores are propagated to the candidates who has commented the corresponding pull-requests *via* adding them together. Finally, we sort all candidates according their expertise scores, and recommend top-k reviewers to the new pull-request.

3.5. CN-based recommendation

The basic intuition of *Comment Network* (CN) based recommendation is that the developers who share common interests with a pull-request originator are appropriate reviewers. The common interests among developers can be directly reflected by commenting interactions between pull-requests' submitters and reviewers.

[5] https://github.com/patanamon/revfinder/blob/master/stringCompare.py.

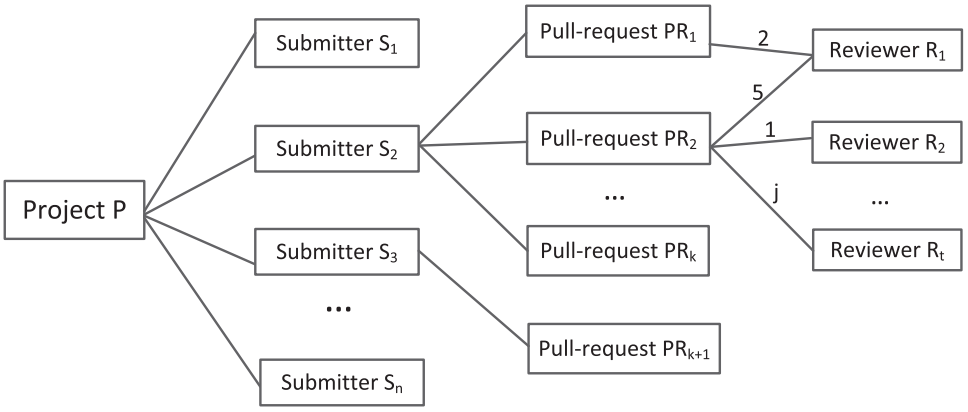


Fig. 5. Comment relations between contributors (submitters) and reviewers.

We propose a novel approach to construct *comment networks* by mining historical comment traces. Based on the *comment network*, we predict highly relevant reviewers to incoming pull-requests. We can also combine this measure with the scores arising from comment interest and reviewer expertise (*e.g.*, using IR-based and FL-based approaches).

3.5.1. Comment network construction

In a given project, the structure of relations among pull-requests, submitters, and reviewers are many-to-many models. As shown in Fig. 5, there are many contributors have submitted pull-requests to Project *P*. A developer can be a contributor submitting several pull-requests, and he could also be a reviewer in other contributors' pull-requests. In addition, a pull-request would be commented by several reviewers more than once. For example, reviewer R_1 had presented 5 comments in the pull-request PR_2 which is contributed from submitter S_2. A reviewer would inspect multiple pull-requests, such as reviewer R_1 has commented PR_1 and PR_2.

We consider that common interests among developers are *project-specific*, so we build a *comment network* for each project separately. The *comment network* is defined as a weighted directed graph $G_{cn} = \langle V, E, W \rangle$, where the set of developers is indicated as vertices V and the set of relations between nodes as edges E. If node v_j has reviewed at least one of v_is pull-requests, there is an edge e_{ij} from v_i to v_j. The set of weights W reflects the importance degree of edges, and the weight w_{ij} of e_{ij} can be evaluated by Eq. (3). Our hypotheses for Eq. (3) are:

(1) the comments in multiple pull-requests are more important than those in one pull-request;
(2) the new comments are more important than old ones.

$$w_{ij} = \sum_{r=1}^{k} w_{(ij,r)} = \sum_{r=1}^{k} \sum_{n=1}^{m} \lambda^{n-1} \times t_{(ij,r,n)} \tag{3}$$

where k is the total number of pull-requests submitted by v_i, and $w_{(ij,\,r)}$ is a component weight related to an individual pull-request r, and m is the sum of comments submitted by v_j in the same pull-request r. In order to distinguish the differences between the comments submitted to multiple pull-requests against a single pull-request, we add an empirical factor λ. When reviewer v_j issued multiple comments ($m \neq 1$) in the same pull-request, his influence is controlled by the decay factor λ (set to 0.8, because the CN-based recommendation can get the highest F-Measure in our prior work [8]). For example, if reviewer v_j commented on 5 different pull-requests of v_i and meanwhile v_q commented one of v_is pull-requests 5 times, the weight of w_{ij} is larger than w_{iq}. Lastly, the element $t_{(ij,r,n)}$ is a time-sensitive factor of corresponding comment which can be calculated as below:

$$t_{(ij,r,n)} = \frac{timestamp_{(ij,r,n)} - start\_time}{end\_time - start\_time} \in (0, 1] \tag{4}$$

where $timestamp_{(ij,r,n)}$ is the date that reviewer v_j presented the comment n in pull-request r which is reported by v_i. The *start_time* and *end_time* are highly related to the selection of training set. In our data set (see Section 4.1), we use the data of from 01/01/2012 to 31/05/2014 to learn the weights of *comment network*, the parameters *start_time* and *end_time* are set to 31/12/2011[6] and 31/05/2014 respectively.

Fig. 6 depicts a small part of the *comment network* in *Ruby on Rails* (*i.e., rails*). We take it as an example to explain how to calculate the weights of edges. The comment-logs show that two different pull-requests (`PR_1 and PR_2`) created by v_1 have been commented by v_2 and v_3, so there are two edges from v_1 to v_2 and v_1 to v_3. Evaluating the relation between v_1 and v_2, k of Eq. (3) equals 2, because v_2 reviewed both two pull-requests. For PR_1, v_2 commented it twice, so we set $m = 2$. The first time-sensitive factor of the date 03/12/2013 can be computed by Eq. (4) that:

$$t_{(12,1,1)} = \frac{value(03/12/2013) - value(31/12/2011)}{value(31/05/2014) - value(01/01/2012)} \approx 0.654.$$

In addition, at the date of 12/01/2013 ($t_{12,\,1,\,2} \approx 0.731$), another review created by v_2 in PR_1 should be modulated by λ (set to 0.8) due to the diminishing impact of one user in the same pull-request, so $w_{(12,\,1)}$ can be calculated as: $t_{(12,1,1)} + \lambda^{2-1} \times t_{(12,1,2)} = 0.654 + 0.8 \times 0.731 \approx 1.24$. Similarly, the weight $w_{12} = w_{(12,1)} + w_{(12,2)} = 2.19$, and $w_{13} = 0.95$. Thus, we can predict that reviewer v_2 share more common interests with contributor v_1 compared with v_3, which has been quantified by the corresponding weights of edges.

The *comment network* has several desirable qualities.

- First, the global collaboration structure between contributors and reviewers in a given project is modeled, and used to select reviewer candidates of incoming pull-requests.
- Secondly, the time-sensitive factor t is introduced to ensure that the recent comments are more pertinent than the old comments.
- Thirdly, the decay factor λ is introduced to impose differences between the comments submitted to multiple pull-requests against a single pull-request. For example, if reviewer v_j commented on 5 different pull-requests of v_i and meanwhile v_q commented one of v_i's pull-requests 5 times, the weight of w_{ij} is larger than w_{iq}.

[6] We set the start_time one day ahead of the earliest date to avoid $w_{ij,r} = 0$.

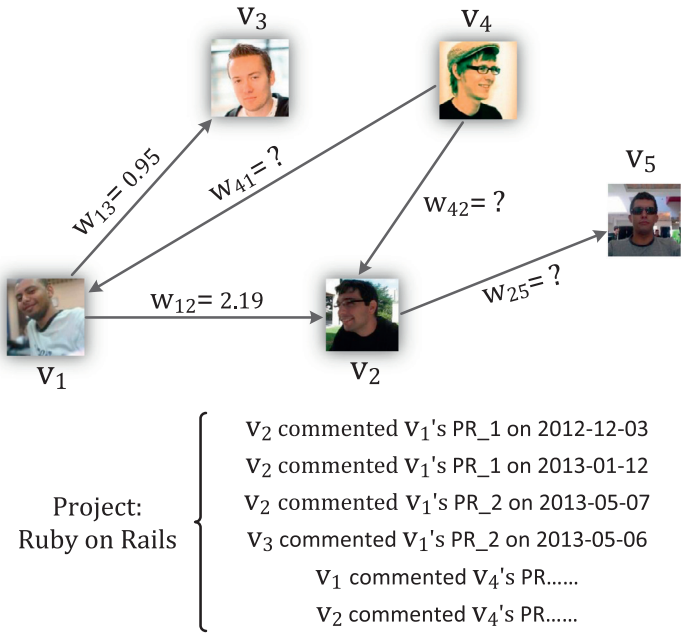


Fig. 6. Example of the *comment network*.

- Finally, it balances these two important factors to quantify the influence that a reviewer has exerted on the contributor in a project.

3.5.2. Reviewers recommendation

Based on *comment networks*, new pull-requests are divided into two parts based on their submitters. The first part are the *Pull-requests from Acquaintance Contributors* denoted as *PAC*. For a *PAC*, starting from the node of its submitter, we can find at least one neighbor in the directed graph. For example, in Fig. 6, when v_1 submits a new pull-request, that pull-request is a *PAC* because two neighbors starting from v_1 can be reached. The other part are *Pull-requests from New Contributors* denoted as *PNC*. For a *PNC*, the submitter only plays a role of reviewer (integrator) but has not submitted any pull-request, or she/he is a newcomer excluded from the training set, so there is no neighbor starting from it in the *comment network*. Hence, we design different algorithms of reviewer recommendation for *PAC*s and *PNC*s.

Recommendation for PAC: For a *PAC*, it is natural to recommend the user who has previously interacted with the contributor directly, *i.e.*,, the node that is a connected neighbor starting from the contributor node in the *comment network*. If there are more than one neighbor, the node with the highest weights get selected first. Hence, reviewer recommendation can be treated as a kind of directed graph traversal problem [45]. In this paper, we improve the classical method of *Breadth-First Search* to recommend top-*k* reviewers for new pull-requests as shown in Algorithm 1. First of all, we initialize a queue and put the source node v_s onto this queue. Then, starting from the unvisited edge with the highest weight (*RankEdges*) every time, we loop to select (*BestNeighbor*) and marked the nearest neighbor as a candidate. If the number of contributor's neighbors is less than top-*k*, we further to visit the child nodes until top-*k* nodes are found.

Recommendation for PNC: For a *PNC*, since there is no prior knowledge of which developers used to review the submitter's pull-request, we want to predict the candidates who share common interests with this contributor by analyzing the overall structure of *comment network*.

Algorithm 1 Top-*k* recommendation for *PAC*.

Require: G_{cn} is the comment network of a given project;
v_s is the contributor of a new pull-request;
top_k is the number of required reviewers;
Ensure: *recSet* is a set of sorted reviewers;
1: $Q.enqueue(v_s)$ and $recSet \leftarrow \emptyset$
2: **repeat**
3: $v \leftarrow Q.dequeue$ and $G_{cn}.RankEdges(v)$
4: **repeat**
5: **if** $top_k = 0$ **then**
6: **return** *recSet*
7: **end if**
8: $v_{nb} \leftarrow G_{cn}.BestNeighbor(v)$
9: $Q.enqueue(v_{nb})$ and $G_{cn}.mark(v_{nb})$
10: $recSet \cup \{v_{nb}\}$ and $top_k = top_k - 1$
11: **until** $G_{cn}.Neighors(v)$ all marked
12: **until** Q is empty
13: **return** *recSet*

Firstly, for a contributor who is a node but without any connected neighbor in the *comment network*, we mine the reviewers based on patterns of co-occurrence across pull-requests. For example, if v_2 and v_3 have reviewed plenty of pull-requests together, we can assume that they would share more common interests than others. Thus, when v_3 submitted a new pull-request (*PNC*), we recommend v_2 to review his pull-request, and vice versa. Each pull-request is a transaction, and the co-occurrent reviewers in that pull-request is the items. We use *Apriori* algorithm of association rule mining [46] to generate top-*k* frequent itemsets, and then rank the candidates according to their *supports*. After that, we can know that who have always reviewed pull-requests together for top-*k* recommendation.

In addition, for a newcomer who is a node excluded from the *comment network*, the most active reviewers in different kinds of communities become the most probable candidates. Each community is a group of developers. Pairs of developers are more likely to be connected if they are both members of the same community, and less likely to be connected if they do not share communities.

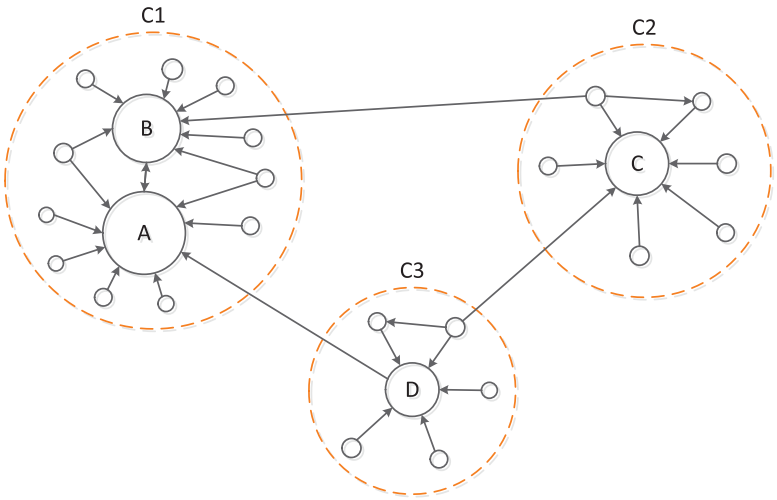


Fig. 7. Community structure of the *comment network*.

In Fig. 7, there are three communities (*i.e.,* C1, C2 and C3). Each set of nodes (*i.e.,* developers) in the same community is densely connected internally. We find that communities will form spontaneously within developers sharing common interests in the *comment network*; this mirrors a well-known phenomenon in email social networks [47] and the "following" network [37]. However, the structure of the *comment network* shows that developers are not always uniformly dispersed, as shown in Fig. 7. It is likely that the most active reviewers belong to the biggest community, such as the top-2 most active nodes *A* and *B* belong to the same community C1. Therefore, we would like our recommendation list to cover multiple communities, instead of always choosing from the biggest communities. In our implementation, we extract the community structure from the *comment network* using the *Gephi* [48] which integrates and optimizes a well-known algorithm of community detection due to Blondel et al. [49]. The size of the communities is represented by the number of developers it contains, and the activity of a reviewer is reflected by the number of pull-requests she/he has reviewed in history (*i.e.,* the in-degree of a node in the *comment network*). The recommendation set is generated by following steps:

(1) rank the communities by their size (number of nodes) and omit the extremely small communities involving less than 2 nodes;
(2) calculate the in-degree of nodes in the remaining communities;
(3) select out the highest in-degree nodes from these communities each time according to the community-ranking, until we have got enough number of reviewers.

3.6. Combination recommendation

Because the reviewer expertise and the comment interest are two different dimensional features, our conjecture is that the recommendation result would be improved if we integrate them together.

When a new pull-request is submitted, we can get a ranking list of reviewer candidates with expertise scores using the IR-based or FL-based approach. Then, the common interests can be calculated by starting from the submitter in the *comment network*. In this paper, we regard that the factor of common interest is as important as the factor of expertise in each project. Thus, we standardize the each factor ranging from 0 to 1 separately, and then add them together to recommend top-*k* reviewers for the new pull-request. In future, we plan to deeply analyze the influence of each factor exerted on different kinds of projects (*e.g.,* scale of team size).

4. Empirical evaluation

4.1. Data collection

Projects and pull-requests: In our previous work [12], we have composed a comprehensive dataset[7] to study the pull-based model, involving 650,007 pull-requests across 919 main-line projects in GitHub (dump dated 10/11/2014 based on GHTorrent [50,51]). In this paper, we need the projects containing enough number of pull-requests for training and test. Thus, we firstly identify candidate projects (154 projects) that received at least 1000 pull-requests throughout their history. Then, for each project, we gather the meta-data (*i.e.,* title, description, comments and files' full path) of the pull-requests submitted from 01/01/2012 to 31/05/2014 as the training set, and the rest part (last 6 months) as the test set. In order to learn the valid classifiers, we first do stop words removal and stemming over the descriptions and titles. Then, we retain those pull-requests with more than 5 words. In the test sets, a part of pull-requests are so tractable that need not be discussed by developers. Hence, we remain the pull-requests commented by at least 2 different reviewers (excluding the submitter), because two reviewers find an optimal number during code review [19,52,53]. Lastly, we filter unbalanced projects based on their sizes of training set and test set, *e.g.,* some young projects contain more pull-requests in the test set than in the training set.

Reviewer candidates: For each project, we identify the reviewer candidates as those developers who have reviewed others' pull-requests in our training set, and calculate the total number of pull-requests they have commented. We remove the reviewers who occasionally commented only one pull-request. In other words, we remain the candidates who have reviewed at least two pull-requests in the training set. We assume that those candidates are more likely to review pull-requests repeatedly, if they already do that over twice in the past. Finally, a ranking list of the rest of candidates can be generated according to the number of comments they posted before.

The final dataset consisted of 84 projects written in the most popular languages on GitHub, including some of the most popular projects (*e.g., rails, jquery* and *angular.js*), as shown in Table 1.

[7] https://github.com/yuyue/pullreq_ci.

Table 1
Summary statistics for 84 GitHub projects.

Statistic	num_projects	Mean	St. dev.	Min	Max
training_pullreqs	84	1251.464	759.275	527	4816
training_comments	84	6156.762	4662.763	1478	22,088
test_pullreqs	84	75.250	81.860	10	444
test_comments	84	911.214	1015.041	43	4688
candidates	84	63.905	67.424	13	457
files	84	2871.786	2464.207	331	14,171

The pool of reviewer candidates is relative large. For example, we need to identity the suitable reviewers out of 197 candidates in the project *angular.js*. The minimum number of pull-requests for a project is 527 in the training set (**training_pullreqs**), 10 pull-requests in the test set (**test_pullreqs**), 13 reviewer candidates (**candidates**), and 331 different files (**files**) in that project have been touched by pull-requests.

4.2. Evaluation metrics

We evaluate the performances of our approaches over each project by *precision, recall* and *F-Measure* which are widely used as standard metrics in previous work of bug assignment [16,54–56] and code-reviewer recommendation [29,57]. In the reviewer recommendation context, precision is the fraction of recommended reviewers that are relevant to the actual reviewers, and recall is the fraction of the actual reviewers that are successfully recommended. F-Measure considers both the precision and the recall of the test. These metrics are computed for different sizes of recommendation ranging from the top-1 to top-10. The formulae for our metrics are listed below:

$$Precision = \frac{|Rec\_Reviewers \cap Actual\_Reviewers|}{|Rec\_Reviewers|}$$

$$Recall = \frac{|Rec\_Reviewers \cap Actual\_Reviewers|}{|Actual\_Reviewers|} \quad (5)$$

$$F\text{-}Measure = \frac{2 * Precision * Recall}{Precision + Recall}$$

5. Results and discussion

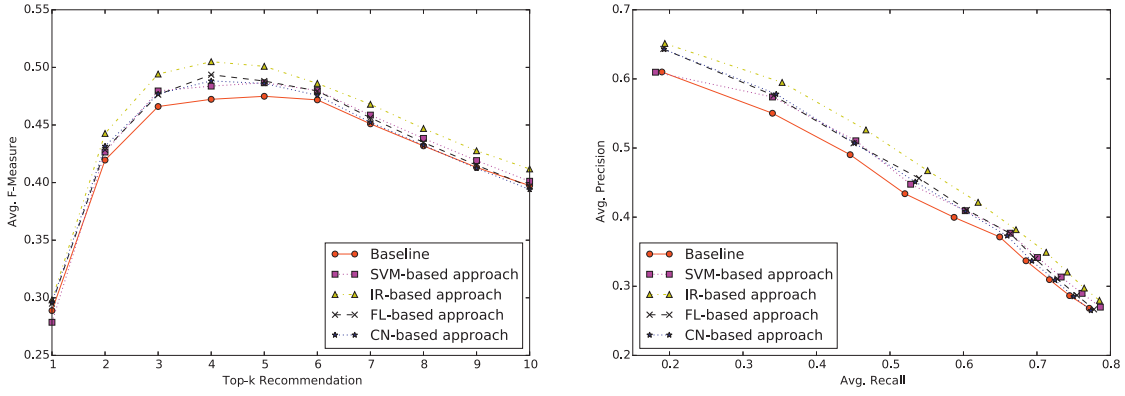
In this section, we present our experiment results and analyze the results compared with a baseline method.

5.1. Experiment baseline

In reality, it is common that most of pull-requests are reviewed by a few of core developers. To demonstrate the effectiveness of sophisticated approaches, we design a baseline method that every new pull-request is assigned to the top-*k* most active developers ranked according to the number of pull-requests they had reviewed before the creation time of the new pull-request.

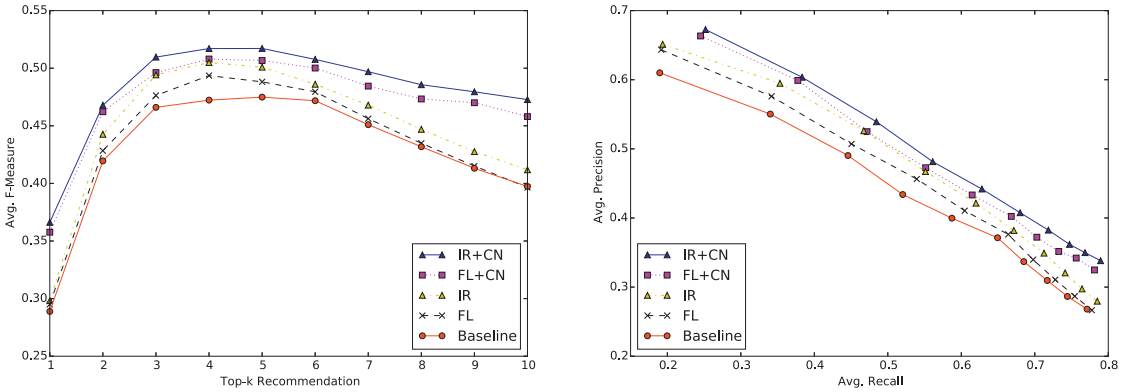
5.2. Recommendation evaluation

Independent approaches: Firstly, we evaluate the SVM-based, IR-based, FL-based and CN-based approaches independently compared to each other. Fig. 8 exhibits the performances of above approaches. Overall, all approaches are superior to the baseline, and achieve the best performance in terms of F-Measure when recommending top-4 and top-5 reviewers. As shown in Fig. 8(a), the IR-based approach performs best compared to other methods, and achieves the highest F-Measure of 50.5% at top-4 recommendation. Except for the IR-based approach, the F-Measures of other approaches are similar to each other. The curve of SVM-based recommendation is lowest at the beginning, and then take the lead from top-7 to top-10 recommendation compared with the CN-based and FL-based approach in Fig. 8(a). Furthermore, we show the performances of different approaches in detail *via* the chart of precision *vs.* recall. Each curve has a point for each recommendation from top-1 to top-10 recommendation in Fig. 8(b). The performance of SVM-based approach is unstable from top-1 to top-6. It gets the worst result both precision and recall at top-1, and then exceeds CN-based and FL-based approach at top-3 recommendation. However, it descends sharply at top-4 recommendation. On the contrary, the curves of other approaches decline steadily from top-1 to top-10, because there is a trade-off between precision and recall. The CN-based approach performs as good as the FL-based approach (the state-of-art approach in code-reviewer recommendation), which achieves 64.3% of precision and 19.3% of recall at top-1 recommendation, and 26.5% of precision and 77.3% of recall at top-10 recommendation.



(a) F-Measure (b) Precision vs. Recall

Fig. 8. Performances of different approaches compared to the baseline.



(a) F-Measure (b) Precision vs. Recall

Fig. 9. Performances of mixed approaches compared to independent approaches.

Mixed approaches: We evaluate the combination approaches among the IR-based, FL-based approach (used to measure reviewer expertise) and CN-based (used to measure social relations), compared to one of them individually. As depicted in Fig. 9, the mixed approach of IR and CN achieves the best performance, which can get 67.3% of precision and 25.2% of recall at top-1 recommendation, and 33.8% of precision and 79.0% of recall at top-10 recommendation. Similarly, the mixed approach of FL and CN make a significant improvement compared to the FL-based approach and the baseline. Furthermore, both F-Measure curves of mixed approaches (*i.e.,* combing reviewer expertise and social relations) decrease more slightly and smoothly than other approaches.

Multiple contrast test: In our previous work [8], we find there is no consistent performance across all projects using the same approach. In addition, we can see that some curves interweave together in Figs. 8 and 9. Thus, we need to assess whether the distributions of the F-Measure are different between the single and mixed recommendation approaches. Traditionally, comparison of multiple groups follows a two-step approach [58]: first, a global null hypothesis is tested, and then multiple comparisons are used to test sub-hypotheses pertaining to each pair of groups. Unfortunately, the global test null hypothesis may be rejected while none of the sub-hypotheses are rejected, or vice versa [59]. Therefore, the one-step approach, multiple contrast test procedure $\widetilde{\mathbf{T}}$ [60], is preferred in this paper. This method produce confidence intervals which always lead to the same test decisions as the multiple comparisons. We implement the procedure $\widetilde{\mathbf{T}}$ by *nparcomp* package [61] in R to evaluate the F-Measure of all the approaches running on 84 projects in our dataset from top-1 to top-10 recommendation. We set the *Tukey* (all-pairs) contrast to compare all groups pairwise. Since we evaluate 6 different approaches (SVM, IR, FL, CN, IR+CN and FL+CN), we conduct $A_6^2 = 15$ comparisons and report the results in Table 2. For each pair of groups, we analyse the 95% confidence interval to test whether the corresponding null sub-hypothesis can be rejected. If the lower boundary of the interval is greater than zero for groups A and B, then we claim that the metric value is higher in A than in B. Similarly, if the upper boundary of the interval is less than zero for groups A and B, then we claim that the metric value is lower in A than in B. Finally, if the lower boundary of the interval is less than zero and the upper boundary is greater than zero, we conclude that the data does not provide enough evidence to reject the null hypothesis.

In Table 2, all the *p-Values* are over 0.05 in the first four rows, and there are flips between lower boundaries (*Lower* < 0) and up boundaries (*Upper* > 0) in the fifth and sixth rows. Thus, there is no statistical significant difference among SVM, IR, FL and CN based recommendation. However, the two mixed approaches (*i.e.,* IR+CN and FL+CN in Table 2) represent statistical significant improvement compared to use those approaches independently. For example, as shown in the last row of Table 2, the $p\text{-}Value < 0.05$ and both lower boundary and upper boundary are greater than zero. It means that the F-Measure of the mixed approach of *IR+CN* is significant higher than the F-Measure of CN-based approach. Likewise, since the $p\text{-}Value < 0.05$ and both lower boundary and upper boundary are less than zero, as shown in the next-to-last row, the F-Measure of IR alone is significant lower than the F-Measure of IR+CN combination.

5.3. Conclusions of quantitative analysis

Based on the experiment results and analysis above, we can draw our conclusions as following:

- For **RQ1**, the IR-based recommendation is the most effective approach for pull-request assignment among the traditional approaches based on our overall analysis.
- For **RQ2**, the *CN*-based approach gets the similar results compared to traditional approaches. All approaches achieve the best performance at top-4 and top-5 recommendation.
- For **RQ3**, our results show that the mixed approaches can achieve significant improvement. We argue that combining social relations and reviewer expertise, complementary to each other, is a novel and effective way for reviewer recommendation of pull-request.

5.4. Qualitative analysis

We manually analyze 50 recommendation cases to deeply investigate the benefits of combining developer expertise with social relations. We start from 5 popular projects with over 5000 stars in GitHub, written in 5 different languages (*i.e., rails*[8] in Ruby, *cocos2d-x*[9] in C++, *ipython*[10] in Python, *zf2*[11] in PHP, and *netty*[12] in Java). Then, we randomly choose 10 pull-requests (from the test data) from every project as the samples for the qualitative analysis. We can get an overall sense of the evaluation process, involving (1) what reviewers usually concern with in pull-request evaluation compared to modern code review (*e.g.,* finding defects is the first motivation during the modern code review [28]); (2) what challenges the recommendation approaches. By performing

[8] https://github.com/rails/rails.
[9] https://github.com/cocos2d/cocos2d-x.
[10] https://github.com/ipython/ipython.
[11] https://github.com/zendframework/zf2.
[12] https://github.com/netty/netty.

Table 2
Results of multiple contrast test procedure.

Group A *vs.* B	Estimator	Lower	Upper	Statistic	*p* Value
FL *vs.* CN	0.009	−0.032	0.051	0.651	$0.987e-01$
SVM *vs.* CN	0.004	−0.037	0.045	0.291	$9.997e-01$
IR *vs.* FL	0.023	−0.018	0.065	1.587	$0.607e-01$
SVM *vs.* FL	−0.005	−0.047	0.037	−0.357	$0.992e-01$
IR *vs.* CN	0.033	−0.008	0.073	2.292	$0.197e-01$
SVM *vs.* IR	−0.028	−0.069	0.013	−1.969	$0.360e-01$
IR+CN *vs.* FL+CN	0.026	−0.011	0.062	2.014	$0.334e-01$
FL *vs.* FL+CN	−0.079	−0.118	−0.038	−5.582	$2.696e-07$
FL+CN *vs.* CN	0.088	0.049	0.127	6.431	$1.034e-09$
IR *vs.* FL+CN	−0.056	−0.095	−0.016	−4.035	$7.813e-04$
IR+CN *vs.* FL	0.105	0.065	0.143	7.557	$4.374e-13$
SVM *vs.* FL+CN	−0.084	−0.123	−0.044	−6.027	$1.518e-08$
SVM *vs.* IR+CN	−0.110	−0.148	−0.071	−8.038	$1.688e-14$
IR *vs.* IR+CN	−0.081	−0.120	−0.043	−6.039	$2.109e-08$
IR+CN *vs.* CN	0.114	0.076	0.152	8.489	$4.441e-16$

the random sampling, we are more likely to avoid some researcher biases [62]. Next, as shown in Figs. 8 and 9, the IR-based recommendation is the most effective approach for pull-request assignment among all independent approaches, and the mixed approach of IR and CN achieves the greatest improvement. Therefore, we focus on the top-10 recommended results generated from the IR-based approach (used to measure reviewer expertise), the CN-based approach (used to measure social relations), and the mixed approach of IR and CN, respectively. We analyze the properties and historical activities of those recommended reviewers, including their social status (*i.e.,* project owner, collaborator, external contributor or user), the amount of pull-requests previously reviewed, the content of past comments, and the ranking position in the corresponding recommendation approach. Finally, we discuss the advantages of combining developer expertise with social relations to recommend reviewers, and perform a purposeful sampling to validate the conclusions. We select out 5 specific cases where the mixed approach works well (F-Measure over 60%), but the independent approaches get a relative low performance (F-Measure below 40%), on average from top-1 to top-10.

First of all, reviewers generally evaluate the quality of a pull-request, including technical details and coding standards. We use a few sampling comments from the project *rails* to present our findings. To evaluate technical details, the reviewers discuss the correctness of the parameters and functions involved in pull-requests. As shown in **Example_1** and **Example_2**, the reviewers ran the patches themselves, and then pointed out the technical mistakes. Apart from manual testing, reviewers can also judge the initial quality of new pull-requests based on the failures detected by a continuous integration system [11]. Additionally, the code style, documentation, and test case inclusion would be checked to make sure the new pull-requests meet their coding standards. As shown in **Example_3** and **Example_4**, the project prefers to accept clean, compact and well-documented contributions.

Example_1: *"I tried that code, but when* `id:false` *is used,the generator isn't created so you can't set the value of the generator."*
Example_2: *"Previously I just had an initializer with the following line:*
`ActionController::Parameters.permit_all_parameters =true`, *but after your change this does not work any more."*
Example_3: *"Maybe we should keep the old code removed in the PR I pointed. It seems to be simpler."*
Example_4: *"Commented what is missing. After you change those things please add a CHANGELOG entry and squash your commits."*

Second, a part of projects have dominant reviewers, who comment on a majority of pull-requests. These dominant reviewers are usually the projects owners (*e.g., takluyver* in *ipython*), or the core-collaborators (*e.g., normanmaurer* in *netty*). In addition to evaluating the technical quality of a pull-request, they are also in charge of reassigning it to appropriate reviewers (**Example_5** from *zf2*), managing the project roadmap (**Example_6** from *rails*), and attracting new contributors (**Example_7** from *ipython*).

Example_5: *"Assigning to @EvanDotPro for review this PR."*
Example_6: *"Good on leaving it out of 4-2-0 branch.4-1-stable I'm not sure, but I think it's fine to backport."*
Example_7: *"Merged-thanks, and congratulationson your first contribution."*

From our case study, we find that the recommendation approaches based on developers' expertise achieve a relatively low hit-rate for the projects with multiple dominant reviewers. Taking the IR-based approach as an example, the technical words in a reviewer's local corpus, such as module and function names (*e.g., GUIReader, ActionController* and *send_raw()*), are the most important features to characterize the expertise of that reviewer at semantic level. These technical words for a reviewer are derived from the old pull-requests commented by them in the past, for any kind of reason discussed above (*e.g.,* managing the project roadmap or attracting the contributors). When a reviewer has commented on many pull-requests (*i.e.,* when she/he becomes a dominant reviewer), the local corpus would contain almost all the frequent technical words that appear in the corresponding project. Thus, if a project has multiple dominant reviewers, all of them are likely to be recommended to a new pull-request, and are likely to be comparably (highly) ranked by their expertise scores. The performance of the IR-based approach would therefore not decrease if all of them indeed participated in the review process for pull-requests. However, in fact, it is not uncommon that the dominant reviewers alternate with each other to review some new pull-requests which are not very complicated (*e.g.,* a pull-request only changed a parameter `location` to `parameter_filtered_location` to make the meaning of that variable clear). It means the decision of those pull-requests (*i.e.,* reject or merge into main branch) can be made by only one or two of those dominant reviewers, which is more likely to happen in the project where pull-requests are mostly reviewed by insiders. In this case, it is hard to distinguish who is the most suitable reviewer for the pull-requests among

all dominant reviewers in terms of expertise, since the technical words in their local corpus are similar.

By analyzing the mixed approach, we find that the recommendation order of the IR-based approach is optimized for the social factor. As described in Section 3.5, our novel comment network can be used to measure the social relations among submitters and reviewers. When the technical factor (*i.e.,* reviewers' expertise) is not significant for assigning pull-requests, the social relations among submitters and potential reviewers would provide more information to help select out the highly relevant reviewers. We explain the improvement by using the examples in the project *ipython*. There are 4 dominant reviewers in that project (*i.e., takluyver, minrk, ellisonbg* and *fperez*), who have reviewed over 50% of the pull-requests in the training set. When using the IR-based approach to recommend reviewers for pull-requests in the test set, they are ranked at top-4 recommendation in 39.5% of test cases (*i.e.,* there are 114 test pull-requests in the test set of *ipython*. For 45 test cases out of 114, those 4 dominant reviewers are ranked at top-4 using the IR-based approach). Within those cases, the differences of expertise scores between those 4 dominant reviewers are very small. For example, when recommending reviewers to *pull-request:6267*[13] submitted by *user:11835, takluyver* is ranked at the first place with 1.000 score (the expertise score has been standardized ranging from 0 to 1), *minrk* at the second place with 0.991, *fperez* at the third place with 0.969, and *ellisonbg* at the fourth place with 0.894. After integrating with the CN-based approach, *minrk* is adjusted to the first place from the second place and *ellisonbg* is moved to the second place from the fourth place, since they have stronger social relations with the submitter than *takluyver* and *fperez* have (*i.e.,* compared to other reviewers, *minrk* and *ellisonbg* have commented on more pull-requests submitted by *user:11835*, or evaluated pull-requests more frequently together with *user:11835* in the past). In reality, only *minrk* and *ellisonbg* indeed did review that pull-request, where *ellisonbg* discussed the technical details and *minrk* managed the project roadmap (adding the pull-request to the 3.0 milestone). Therefore, the precisions from top-1 to top-4 recommendation for the IR-based approach are 0.00, 0.50, 0.33, and 0.50 respectively. Compared to the IR-based approach, the mixed approach achieves 1.00, 1.00, 0.66, and 0.50 of precisions from top-1 to top-4 recommendation. We also confirm the above finding by analyzing 5 specific cases chosen by our purposeful sampling. There are 4 cases in which the performance of the mixed approach is improved in a very similar way as the example discussed above. The other one achieves improvement, because two outsiders are adjusted to the fourth and the seventh place from the tenth and the place beyond the top-10 recommendation. It means there is lack of evidence that assigning the pull-request to those two outsiders based on their prior expertise in this project. The social interactions provide complementary information for our recommendation in this case. We leave a comprehensive study, aiming to find more merits of social factors and its combination, to future work.

5.5. Threats to validity

In this section, we discuss threats to construct validity, internal validity and external validity which may affect the results of our study.

Construct validity

Firstly, the strength of social relations among contributors and reviewers is measured based on the comment network. In addition, GitHub also provides other social information (*e.g.,* follow relations) that can use to model the social distance between developers. In future work, we intend to explore how to use other types of social networks, *e.g.,* follow-fork or watcher-fork networks, to capture more complemented social factors among outside contributors and core members.

Secondly, some technical details in reviewers' comments (*e.g.,* **Example_2** in Section 5.4) would be missing in the descriptions of pull-requests. However, all the technical words are important to measure the expertise of a reviewer, when we utilize the ML-based and IR-based approach. In the future, we plan to categorize the reviewers' comments, and then add the technical comments into our models.

Thirdly, our mixed approaches combine two dimensional features, *i.e.,*developer expertise and social relation, by adding them together to recommend reviewers for every project. However, the technical and social factors would exert different influences on the projects with different features, such as project size (*e.g.,* scale of code size or team size) and popularity (*e.g.,* number of external contributors). For example, a reviewer working on the project with a well-modularized structure would be more likely to review those pull-requests touching the specific modules that she/he is responsible for. The technical words (*e.g.,* module names) are useful to distinguish reviewers in this case, so we need to enhance the impact of technical factors. In the future, we plan to study how to adjust the social or technical effect in accordance with the characteristics of projects.

Internal validity

Firstly, because some pull-requests have not been closed when we dumped the test set, a part of following reviewers have not been taken into consider which may affect our experiment results. If we exclude all open pull-requests, it is regretful that some of projects would contain sufficient training pull-requests but have a lack of test pull-requests.

Secondly, for every project, we use an uniform process to generate the training set and the test set, as described in Section 4.1. After that process, some projects remain a relative small scale of pull-requests (*e.g.,* 10 pull-requests in the test set). Thus, evaluating our approaches on those projects would be introduce a bias compared to other projects.

External validity

Our empirical findings are based on open source projects in GitHub, and it is unknown whether our results can be generalized to the commercial projects or the open source projects hosted on other social coding platforms (*e.g.,* BitBucket).

Additionally, for a given project, there is a small part of the core developers who are in charge of the final decision of pull-requests. They joined so many pull-requests' discussions in the training set that all the approaches tend to assign the new pull-request to these active developers at top-1 recommendation. Hence, the workload of these active reviewers may not be reduced. However, if more and more external contributors present their suggestions to pull-requests, the social network based approach would refresh the weights of corresponding edges, so new pull-requests would be assigned more balanced than before.

6. Conclusion and future work

Social coding is opening a new era of distributed software development and evolution. In GitHub, the high volume of incoming pull-requests poses a serious challenge to project integrators. To improve the evaluation process of pull-requests, we propose various approaches for reviewer recommendation, and evaluate their performances on 84 projects of GitHub using precision, recall and F-Measure. The results show that (1) the traditional approaches

[13] There is a unique ID for each pull-request and each user in our dataset.

using in bug assignment and code-reviewer recommendation perform better than the baseline, especially in top-4 recommendation; (2) our novel approach based on social network analysis achieves similar F-Measure as traditional approaches; (3) the mixed approaches, integrating reviewers' expertise and comment interests, represent significant improvements in precision and recall, and the overall performances of mixed approaches are more stable than using different approaches independently. The results indicate that combining the social factors (*e.g.,* common interests among developers) and technical factors (*e.g.,* developers expertise) is an efficient way to build recommender systems in social coding platforms, *e.g.,* recommending reviewer, bug fixer or coding partner.

In this paper, the traditional approaches (*i.e.,* SVM-based, IR-based and FL-based approach) is regarded as recommending reviewers to new pull-requests using the same dimensional feature, *i.e.,* technical expertise. Therefore, we separately combine two traditional but effective approaches with CN-based recommendation (*i.e.,* IR+CN and FL+CN) to study the performances. In our future work, we plan to explore the combinations of IR+FL, and IR+FL+CN. In the same dimension, the mixed approach of IR and FL may not be expected to rival with the combination of IR and CN, which using two dimensional features (*i.e.,* social and technical features). However, the mixed approach of IR, FL and CN maybe achieve the best performance, or the same level of performance in statistics as the best mixed approach in this paper. Besides, we intend to explore how to use other types of social networks, *e.g., watcher-fork networks*, to capture more complemented social factors among outside contributors and core teams.

Acknowledgment

This research is supported by the National Natural Science Foundation of China (Grants 61432020, 61472430 and 61502512) and the Postgraduate Innovation Fund of University of Defense Technology (Grant no. B130607). We thank Premkumar Devanbu and Bogdan Vasilescu for their very useful feedback on this paper.

References

[1] E.T. Barr, C. Bird, P.C. Rigby, A. Hindle, D.M. German, P. Devanbu, Cohesive and isolated development with branches, in: Proceedings of FASE, Springer, 2012, pp. 316–331.
[2] G. Gousios, M. Pinzger, A.v. Deursen, An exploratory study of the pull-based software development model, in: Proceedings of ICSE, ACM, 2014, pp. 345–355.
[3] G. Gousios, A. Zaidman, M.-A. Storey, A. Van Deursen, Work practices and challenges in pull-based development: the integrator's perspective, in: Proceedings of ICSE, IEEE, 2015, pp. 358–368.
[4] A. Begel, J. Bosch, M.-A. Storey, Social networking meets software development: perspectives from GitHub, MSDN, Stack Exchange, and TopCoder, IEEE Softw. 30 (1) (2013) 52–66.
[5] A. Mockus, R.T. Fielding, J.D. Herbsleb, Two case studies of open source software development: Apache and Mozilla, TOSEM 11 (3) (2002) 309–346.
[6] C. Bird, A. Gourley, P. Devanbu, A. Swaminathan, G. Hsu, Open borders? Immigration in open source projects, in: Proceedings of MSR, IEEE, 2007, p. 6.
[7] Y. Yu, H. Wang, G. Yin, C. Ling, Reviewer recommender of pull-requests in GitHub, in: Proceedings of ICSME, IEEE, 2014a, pp. 609–612.
[8] Y. Yu, H. Wang, G. Yin, C. Ling, Who should review this pull-request: reviewer recommendation to expedite crowd collaboration, in: Proceedings of APSEC, IEEE, 2014b, pp. 609–612.
[9] L. Dabbish, C. Stuart, J. Tsay, J. Herbsleb, Leveraging transparency, IEEE Softw. 30 (1) (2013) 37–43.
[10] R. Pham, L. Singer, O. Liskin, K. Schneider, Creating a shared understanding of testing culture on a social coding site, in: Proceedings of ICSE, IEEE, 2013, pp. 112–121.
[11] B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, V. Filkov, Quality and productivity outcomes relating to continuous integration in GitHub, in: Proceedings of ESEC/FSE, IEEE, 2015, pp. 805–816.
[12] Y. Yu, H. Wang, V. Filkov, P. Devanbu, B. Vasilescu, Wait for it: determinants of pull request evaluation latency on GitHub, in: Proceedings of MSR, IEEE, 2015, pp. 367–371.
[13] G. Georgios, A. Bacchelli, Work Practices and Challenges in Pull-Based Development: The Contributor's Perspective, Internal Report, Delft University of Technology, 2014.
[14] J. Tsay, L. Dabbish, J. Herbsleb, Let's talk about it: evaluating contributions through discussion in GitHub, in: Proceedings of FSE, ACM, 2014, pp. 144–154.
[15] T. Patanamon, T. Chakkrit, K. Raula Gaikovina, Y. Norihiro, I. Hajimu, M. Kenichi, Who should review my code? A file location-based code-reviewer recommendation approach for modern code review, in: Proceedings of SANER, IEEE, 2015, pp. 141–150.
[16] J. Anvik, L. Hiew, G.C. Murphy, Who should fix this bug? in: Proceedings of ICSE, ACM, 2006, pp. 361–370.
[17] J. Anvik, Automating bug report assignment, in: Proceedings of ICSE, ACM, 2006, pp. 937–940.
[18] M. Fagan, Design and code inspections to reduce errors in program development, in: M. Broy, E. Denert (Eds.), Software Pioneers, Springer, 2002, pp. 575–607.
[19] P.C. Rigby, D.M. German, L. Cowen, M.-A. Storey, Peer review on open-source software projects: parameters, statistical models, and theory, ACM Trans. Softw. Eng. Methodol. 23 (4) (2014) 35:1–35:33.
[20] P. Bhattacharya, I. Neamtiu, Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging, in: Proceedings of ICSM, IEEE, 2010, pp. 1–10.
[21] P. Bhattacharya, I. Neamtiu, C.R. Shelton, Automated, highly-accurate, bug assignment using machine learning and tossing graphs, J. Syst. Softw. 85 (10) (2012) 2275–2292.
[22] D. Cubranic, G.C. Murphy, Automatic bug triage using text categorization, in: Proceedings of SEKE, 2004, pp. 92–97.
[23] G. Jeong, S. Kim, T. Zimmermann, Improving bug triage with bug tossing graphs, in: Proceedings of FSE, ACM, 2009, pp. 111–120.
[24] G. Canfora, L. Cerulo, Supporting change request assignment in open source development, in: Proceedings of SAC, ACM, 2006, pp. 1767–1772.
[25] H. Kagdi, D. Poshyvanyk, Who can help me with this change request? in: Proceedings of ICPC, IEEE, 2009, pp. 273–277.
[26] M. Linares-Vasquez, K. Hossen, H. Dang, H. Kagdi, M. Gethers, D. Poshyvanyk, Triaging incoming change requests: bug or commit history, or code authorship? in: Proceedings of ICSM, IEEE, 2012, pp. 451–460.
[27] A. Tamrawi, T.T. Nguyen, J.M. Al-Kofahi, T.N. Nguyen, Fuzzy set and cache-based approach for bug triaging, in: Proceedings of FSE, ACM, 2011, pp. 365–375.
[28] A. Bacchelli, C. Bird, Expectations, outcomes, and challenges of modern code review, in: Proceedings of ICSE, IEEE, 2013, pp. 712–721.
[29] G. Jeong, S. Kim, T. Zimmermann, K. Yi, Improving code review by predicting reviewers and acceptance of patches, Research on Software Analysis for Error-free Computing Center Tech-Memo (ROSAEC MEMO 2009-006) (2009).
[30] V. Balachandran, Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation, in: Proceedings of ICSE, IEEE, 2013, pp. 931–940.
[31] P. Thongtanunam, R.G. Kula, A.E.C. Cruz, N. Yoshida, H. Iida, Improving code review effectiveness through reviewer recommendations, in: Proceedings of CHASE, ACM, 2014, pp. 119–122.
[32] M. Cataldo, J.D. Herbsleb, K.M. Carley, Socio-technical congruence: a framework for assessing the impact of technical and work dependencies on software development productivity, in: Proceedings of ESEM, ACM, 2008, pp. 2–11.
[33] M. John, F. Maurer, B. Tessem, Human and social factors of software engineering: workshop summary, ACM SIGSOFT Softw. Eng. Notes 30 (4) (2005) 1–6.
[34] M. Zhou, A. Mockus, What make long term contributors: willingness and opportunity in OSS community, in: Proceedings of ICSE, IEEE, 2012, pp. 518–528.
[35] B. Vasilescu, D. Posnett, B. Ray, M.G.J. van den Brand, A. Serebrenik, P. Devanbu, V. Filkov, Gender and tenure diversity in GitHub teams, in: Proceedings of CHI, ACM, 2015, pp. 3789–3798.
[36] F. Thung, T.F. Bissyande, D. Lo, L. Jiang, Network structure of social coding in GitHub, in: Proceedings of CSMR, IEEE, 2013, pp. 323–326.
[37] Y. Yu, G. Yin, H. Wang, T. Wang, Exploring the patterns of social behavior in GitHub, in: Proceedings of the 1st International Workshop on Crowd-based Software Development Methods and Technologies, CrowdSoft 2014, ACM, 2014, pp. 31–36.
[38] M. Gharehyazie, D. Posnett, B. Vasilescu, V. Filkov, Developer initiation and social interactions in OSS: a case study of the Apache Software Foundation, Emp. Softw. Eng. (2014) 1–36.
[39] L. Dabbish, C. Stuart, J. Tsay, J. Herbsleb, Social coding in GitHub: transparency and collaboration in an open software repository, in: Proceedings of CSCW, ACM, 2012, pp. 1277–1286.
[40] J. Tsay, L. Dabbish, J. Herbsleb, Influence of social and technical factors for evaluating contribution in GitHub, in: Proceedings of ICSE, ACM, 2014, pp. 356–366.
[41] G. Tsoumakas, I. Katakis, Multi-label classification: an overview., IJDWM 3 (2007) 1–13.
[42] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, I.H. Witten, The weka data mining software: an update, ACM SIGKDD Explor. Newslett. 11 (1) (2009) 10–18.
[43] J. Zobel, A. Moffat, Exploring the similarity space, in: Proceedings of ACM SIGIR Forum, vol. 32, ACM, 1998, pp. 18–34.
[44] M. Bilenko, R.J. Mooney, Adaptive duplicate detection using learnable string similarity measures, in: Proceedings of KDD, ACM, 2003, pp. 39–48.
[45] S. Even, Graph Algorithms, 2nd edition, Cambridge University Press, New York, NY, USA, 2011.
[46] R. Agrawal, R. Srikant, Fast algorithms for mining association rules in large databases, in: Proceedings of VLDB, Morgan Kaufmann Publishers Inc., 1994, pp. 487–499.
[47] C. Bird, D. Pattison, R. D'Souza, V. Filkov, P. Devanbu, Latent social structure in open source projects, in: Proceedings of FSE, ACM, 2008, pp. 24–35.

[48] M. Bastian, S. Heymann, M. Jacomy, Gephi: an open source software for exploring and manipulating networks, in: Proceedings of ICWSM, 2009, pp. 361–362.
[49] V.D. Blondel, J.-L. Guillaume, R. Lambiotte, E. Lefebvre, Fast unfolding of communities in large networks, J. Stat. Mech.: Theory Exp. 2008 (10) (2008) P10008.
[50] G. Gousios, The GHTorrent dataset and tool suite, in: Proceedings of MSR, IEEE, 2013, pp. 233–236.
[51] G. Gousios, B. Vasilescu, A. Serebrenik, A. Zaidman, Lean GHTorrent: GitHub data on demand, in: Proceedings of MSR, ACM, 2014, pp. 384–387.
[52] C. Sauer, D.R. Jeffery, L. Land, P. Yetton, The effectiveness of software development technical reviews: a behaviorally motivated program of research, IEEE Trans. Softw. Eng. 26 (1) (2000) 1–14.
[53] P.C. Rigby, C. Bird, Convergent contemporary software peer review practices, in: Proceedings of FSE, ACM, 2013, pp. 202–212.
[54] M.S. Zanetti, I. Scholtes, C.J. Tessone, F. Schweitzer, Categorizing bugs with social networks: a case study on four open source software communities, in: Proceedings of ICSE, IEEE, 2013, pp. 1032–1041.
[55] X. Xia, D. Lo, X. Wang, B. Zhou, Accurate developer recommendation for bug resolution, in: Proceedings of WCRE, IEEE, 2013, pp. 72–81.
[56] W. Wu, W. Zhang, Y. Yang, Q. Wang, Drex: Developer recommendation with k-nearest-neighbor search and expertise ranking, in: Proceedings of APSEC, IEEE, 2011, pp. 389–396.
[57] J.B. Lee, A. Ihara, A. Monden, K.-i. Matsumoto, Patch reviewer recommendation in OSS projects, in: Proceedings of APSEC, IEEE, 2013, pp. 1–6.
[58] B. Vasilescu, A. Serebrenik, M. Goeminne, T. Mens, On the variation and specialisation of workload—a case study of the Gnome ecosystem community, Empir. Softw. Eng. 19 (4) (2013) 955–1008.
[59] K.R. Gabriel, Simultaneous test procedures—some theory of multiple comparisons, Ann. Math. Stat. 40 (1) (1969) 224–250.
[60] F. Konietschke, L.A. Hothorn, E. Brunner, et al., Rank-based multiple test procedures and simultaneous confidence intervals, Electron. J. Stat. 6 (2012) 738–759.
[61] F. Konietschke, M.F. Konietschke, Package nparcomp, 2011.
[62] J.R. Fraenkel, N.E. Wallen, H. Hyun, How to Design and Evaluate Research in Education, McGraw-Hill, 1993.