Information and Software Technology 84 (2017) 48–62

Contents lists available at ScienceDirect

Information and Software Technology

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

Who should comment on this pull request? Analyzing attributes for more accurate commenter recommendation in pull-based development

Jing Jiang[a], Yun Yang[a], Jiahuan He[a], Xavier Blanc[b], Li Zhang[a,*]

[a] State Key Laboratory of Software Development Environment, Beihang University, Beijing, China
[b] University of Bordeaux, LaBRI, UMR 5800, F-33400, Talence, France

ARTICLE INFO

Article history:
Received 24 November 2015
Revised 22 October 2016
Accepted 24 October 2016
Available online 6 January 2017

Keywords:
Commenter recommendation
Reviewer recommendation
Attribute selection
Pull-based software development

ABSTRACT

Context: The pull-based software development helps developers make contributions flexibly and efficiently. Commenters freely discuss code changes and provide suggestions. Core members make decision of pull requests. Both commenters and core members are reviewers in the evaluation of pull requests. Since some popular projects receive many pull requests, commenters may not notice new pull requests in time, and even ignore appropriate pull requests.

Objective: Our objective in this paper is to analyze attributes that affect the precision and recall of commenter prediction, and choose appropriate attributes to build commenter recommendation approach.

Method: We collect 19,543 pull requests, 206,664 comments and 4817 commenters from 8 popular projects in GitHub. We build approaches based on different attributes, including activeness, text similarity, file similarity and social relation. We also build composite approaches, including time-based text similarity, time-based file similarity and time-based social relation. The time-based social relation approach is the state-of-the-art approach proposed by Yu et al. Then we compare precision and recall of different approaches.

Results: We find that for 8 projects, the activeness based approach achieves the top-3 precision of 0.276, 0.386, 0.389, 0.516, 0.322, 0.572, 0.428, 0.402, and achieves the top-3 recall of 0.475, 0.593, 0.613, 0.66, 0.644, 0.791, 0.714, 0.65, which outperforms approaches based on text similarity, file similarity or social relation by a substantial margin. Moreover, the activeness based approach achieves better precision and recall than composite approaches. In comparison with the state-of-the-art approach, the activeness based approach improves the top-3 precision by 178.788%, 30.41%, 25.08%, 41.76%, 49.07%, 32.71%, 25.15%, 78.67%, and improves the top-3 recall by 196.875%, 36.32%, 29.05%, 46.02%, 43.43%, 27.79%, 25.483%, 79.06% for 8 projects.

Conclusion: The activeness is the most important attribute in the commenter prediction. The activeness based approach can be used to improve the commenter recommendation in code review.

© 2017 Published by Elsevier B.V.

1. Introduction

The pull-based software development is an emerging paradigm for distributed software development [1,2]. Developers pull code changes from other repositories or the same repository in different branches, and merge them locally, rather than push changes to a central repository. Various open source software hosting sites, notably Github, provide support for pull-based development and allow developers to make contributions flexibly and efficiently. In GitHub, developers are allowed to fork repositories and make changes without asking for permission. Developers can submit pull requests when they want to merge their changes into the repositories they fork from. This pull-based software development separates making modification and integrating change, and makes contributing to others' repositories much easier than it has ever been [3].

As shown in Fig. 1, the pull request process mainly includes three roles in GitHub, namely contributors, core members and commenters. Firstly, contributors modify codes to fix bugs or

* Corresponding author.
E-mail addresses: jiangjing@buaa.edu.cn (J. Jiang), ayonel@qq.com (Y. Yang), lightbot.johnson@gmail.com (J. He), xavier.blanc@labri.fr (X. Blanc), lily@buaa.edu.cn (L. Zhang).

http://dx.doi.org/10.1016/j.infsof.2016.10.006
0950-5849/© 2017 Published by Elsevier B.V.

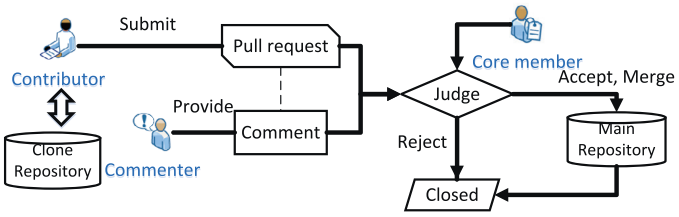


Fig. 1. The overview of contribution process.

improve attributes. When a set of changes is ready, contributors create and submit pull requests to code review platform. Secondly, core members are trusted members of the community. Only experienced and excellent developers are chosen as core members, and they are granted the privilege of directly committing codes to project repositories [4]. Core members evaluate some submitted codes, and decide whether to merge these code changes into repositories or not. Thirdly, any developer can leave comments on pull requests and become a commenter [5,6]. Commenters freely discuss whether code style meets the standard [7], whether repositories require modification, or whether submitted codes have good quality. These comments provide suggestions for the evaluation of pull requests [8]. Core members and commenters both belong to reviewers, but they play different roles in code review process.

In GitHub, popular projects receive many pull requests and make core members feel overwhelmed [4]. Previous works [9,10] propose methods to recommend suitable core members for new pull requests. Besides, core members solicit opinions of the community about the merging decision [1,4]. Developers in the community leave comments on pull requests, and assist core members to make decision. The code review process benefits from the wisdom of the crowd. However, popular projects receive many pull requests, and commenters may not notice new pull requests in time, or even ignore appropriate pull requests [5,6]. Thus, an automatic commenter recommendation approach could remind commenters of appropriate pull requests, and encourage commenters to provide opinions.

We compare the importance of different attributes for recommending commenters in GitHub. Given a new pull request of a contributor and an candidate, we mainly consider 4 kinds of attributes: The activeness is based on recent activities of the candidate; The text similarity measures whether the candidate leaves comments on pull requests with similar description as the new pull request; The file similarity describes distance between modified code files of the new pull request and pull requests commented on by the candidate; The social relation measures whether the candidate prefers to comment on the pull request submitted by this contributor. We take a further step and consider composite attributes, including time-based text similarity, time-based file similarity and time-based social relation. The time-based social relation approach is the state-of-the-art approach [5,6]. Yu et al. uses social relations and builds comment networks to predict appropriate commenters of incoming pull requests [5,6]. We give detailed definitions of these attributes and build corresponding approaches (Section 3).

Approaches based on different attributes achieve various precision and recall, and the good attribute selection is critical for accurate recommendation. The goal of this work is to analyze attributes that affect the performance of commenter prediction, and choose appropriate attributes to build commenter recommendation approach. We find the attribute which has the best precision and recall in the commenter recommendation. We further explore whether the combination of attributes improves the precision and recall.

In this paper, we collect 19,543 pull requests, 206,664 comments and 4817 commenters from 8 popular projects in GitHub (Section 2). We measure precisions and recalls of approaches based on different attributes (Section 4). The experimental results show that (1) for 8 projects, the activeness based approach achieves the top-3 precision of 0.276, 0.386, 0.389, 0.516, 0.322, 0.572, 0.428, 0.402, and achieves the top-3 recall of 0.475, 0.593, 0.613, 0.66, 0.644, 0.791, 0.714, 0.65. The activeness based approach outperforms approaches based on text similarity, file similarity or social relation by a substantial margin. The activeness is the most important attribute in the commenter recommendation. (2) The activeness based approach achieves better precision and recall than composite approaches. In comparison with the state-of-the-art approach [5], the activeness based approach improves the top-3 precision by 178.788%, 30.41%, 25.08%, 41.76%, 49.07%, 32.71%, 25.15%, 78.67%, and improves the top-3 recall by 196.875%, 36.32%, 29.05%, 46.02%, 43.43%, 27.79%, 25.483%, 79.06% for 8 projects.

The main contributions of this paper are as follows:

- We propose approaches based on different attributes to solve the commenter recommendation problem, including activeness, text similarity, file similarity, time-based text similarity and time-based file similarity.
- We experiment on a broad range of datasets containing a total of 19,543 pull requests and 206,664 comments to compare the importance of attributes in the commenter recommendation. Experiment results show that the activeness is the most important attribute. Moreover, the activeness based approach outperforms state-of-the-art approach [5] by a substantial margin.

2. Background and data collection

Before diving into the commenter recommendation, we begin by providing background information about contribution evaluation process in GitHub. Then, we introduce how our datasets are collected, and report statistics of our datasets.

2.1. Contribution evaluation process

GitHub is a web-based hosting service for software development repositories. It has become one of the world's largest open source communities. As shown in Fig. 1, the typical contribution process includes following steps in GitHub [1,11]. First of all, a contributor forks a repository and makes changes to implement new attributes or fix bugs. The contributor submits a pull request when he or she wants to merge code changes into the main repository. The pull request needs to be evaluated by a core member. The core member inspects code changes, evaluates potential contributions, and judges whether to accept the pull request and merge code changes into the main repository. The core member sometimes asks the contributor to make updates and submit new commits for re-evaluation. In GitHub, only core members can make final decisions of pull requests. In contrast, any developer can leave comments on pull requests and become a commenter [5,6]. Commenters find appropriate pull requests, and freely discuss code quality and code style of submitted codes [7,12]. Core members take opinions of commenters into consideration, and then accept or reject pull requests. Core members and commenters both belong to reviewers, but they play different roles in code review process.

The transparency in GitHub improves collaboration, and code review process benefits from the wisdom of the crowd [13]. Due to the large amount of pull requests in popular repositories, commenters become important for providing suggestions and reducing burden of core members [4].

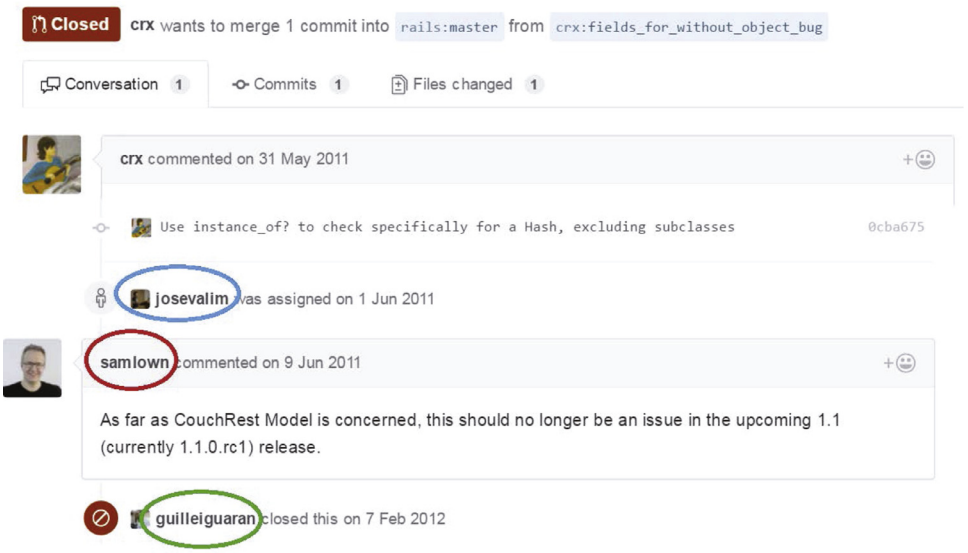


Fig. 2. An example of pull request evaluation.

To illustrate the contribution process, Fig. 2 shows an example of the pull request ID 1414 in the project *rails*.[1] In order to make the figure clear, we mainly show participants of this pull request, and delete some comments. Firstly, a contributor *crx* created a pull request and submitted it for evaluation. Secondly, a core member *josevalim* was manually assigned to evaluate this pull request. Thirdly, a developer *samlown* thought that code modification in the pull request was unnecessary. However, *samlown* was not a core member, and he was not allowed to make the decision of this pull request. Instead, *samlown* left a comment and made suggestion. Finally, a core member *guilleiguaran* rejected the pull request. From this example, we observe that the commenter provides the suggestion for the core member to make decision. Therefore, finding appropriate commenters would reduce workload of core members in the code review.

2.2. Data collection

GitHub provides access to its internal data stores through an API.[2] It allows us to access a rich collection of open source software (OSS) projects, and provides valuable opportunities for research. We gather information from GitHub API and create datasets of commenters and pull requests.

We choose 8 projects in GitHub. The project *rails* is a framework to create database-backed web applications according to the Model-View-Controller pattern; the project *symfony* is a PHP full-stack web framework, which allows developers to build better and easy to maintain websites with PHP; the project *scala* is built for the Scala programming language; the project *bitcoin* is an experimental new digital currency that enables instant payments; the project *zf2* is the framework for modern, high-performing PHP applications; the project *akka* is a toolkit for building highly concurrent, distributed, and resilient message-driven applications on the JVM; the project *cakephp* is a rapid development framework for PHP which uses commonly known design patterns; the project *gitlabhq* is version control for the server.

As shown in Table 1, projects in our datasets have at least 1479 pull requests, 13,374 comments and 135 commenters. The project *rails* even has 5033 pull requests, 48,579 comments and 1972 commenters. We choose these popular projects, because they receive many pull requests and need the commenter recommendation. Unpopular projects receive few pull requests, and core members can make evaluation by themselves. Unpopular projects do not need commenters or commenter recommendation.

We collected pull requests of these 8 projects through GitHub API in July 2014. We sent queries to GitHub API, received its replies, and extracted data from project creation time to July 2014. For each pull request, we crawled its ID, the contributor who submitted it, the creation time, the close time, paths of modified files and the developer who closed the pull request. The contributor wrote the title to summarize the modification of the pull request, and we gathered this text information. We also collected comments of the pull request, including their submission time and commenters. We ignored pull requests with no comments. This is because we do not know actual commenters, and we cannot compare actual results and recommendation results. Contributors wrote comments as replies, but contributors were not considered as commenters in the recommendation. This was because we did not recommend contributors to provide suggestions by themselves.

According to above criteria, we build our datasets, and recommendation approaches consider all pull requests in datasets by default. The activeness based approach excludes pull requests which are created more than γ days before the new pull request. We describe details of this exception in the Section 3.1. We use equations in Section 3 to compute scores of commenters and make recommendation. We do not make any special data normalization or handing on attributes. Different commenters join projects at various times, and only pull requests created before a new pull request can be used to make recommendation. Therefore, we do not make cross-validation, which ignore chronological order of pull requests.

2.3. Data statistics

We report statistics of our datasets in Table 1. In GitHub, project name and its owner name may be different. A developer may fork a project from an original creator [1]. The forked project and original project always have the same project names, but their owner names are different. Therefore, we describe project name and its owner in Table 1.

Our datasets are extracted from project creation time to July 2014. In total, our datasets include 19,543 pull requests, 206,664 comments and 4817 commenters. The project *rails* has 5033 pull

[1] https://github.com/rails/rails/pull/1414, July 2015.
[2] http://developer.github.com/v3/, July 2015.

Table 1
Basic statistics of projects.

Owner	Project	# Pull requests	# Comments	# Commenters	# Files
rails	rails	5033	48,579	1972	3169
symfony	symfony	2830	32,958	898	5414
scala	scala	2709	29,143	168	10,632
bitcoin	bitcoin	2075	22,369	326	1395
zendframework	zf2	2014	18,192	483	5519
akka	akka	1829	26,230	135	3546
cakephp	cakephp	1574	15,819	296	1271
gitlabhq	gitlabhq	1479	13,374	539	4962

Table 2
Percentage of pull requests which have a specific number of files.

	1	2	3	4	≥ 5
rails	44.389	22.615	14.135	5.98	12.881
symfony	45.837	21.422	8.572	4.929	19.24
scala	21.387	14.45	14.018	9.049	41.096
bitcoin	43.994	15.073	9.921	6.044	24.968
zf2	33.782	24.87	8.97	5.621	26.757
akka	28.857	15.7	9.732	6.918	38.793
cakephp	36.912	27.96	7.347	5.029	22.752
gitlabhq	55.273	12.57	6.931	4.907	20.319

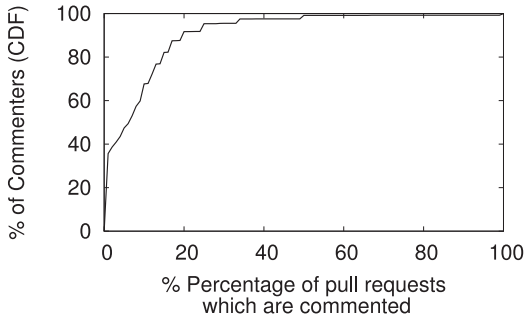


Fig. 3. Percentage of pull requests which are commented by a commenter in the project *rails*.

requests and 1972 commenters. The project *rails* receives many pull requests, and commenters may not notice new pull requests in time, or even ignore appropriate pull requests. For example, the project *rails* received 131 pull requests, and 201 commenters were active in April 2012. Every commenter received 131 pull requests, and tried to find appropriate pull requests from 131 pull requests. It caused redundant work for 201 commenters. The project *rails* even received 181 pull requests, and 189 commenters were active in May 2012. Other projects also have many pull requests and commenters. Therefore, an automatic commenter recommendation approach is needed to remind commenters of appropriate pull requests, encourage commenters to provide opinions and reduce their workload.

We collected the number of files in projects in September 2016, and describe results in Table 1. The project *scala* has as many as 10,632 files, while the project *cakephp* only has 1271 files. We collected paths of modified files for pull requests in July 2014. We compute the number of files for every pull request, and present aggregation results in Table 2. In the project *rails*, 44.389% of pull requests have only 1 file, and 22.615% of pull requests have 2 files. Only 12.881% of pull requests have more than or equal to 5 files. In the project *scala*, 41.096% of pull requests have more than or equal to 5 files. Pull requests in the project *scala* have more files than those in the project *rails*. The majority of pull requests have fewer than 5 files in 8 projects, and some projects have more files in pull requests than other projects.

We take a further step to analyze pull requests. For every commenter, we define the active period in a project as the time between this commenter's first comment and the last comment in this project. If the commenter only leaves 1 comment, the active period is the day when the commenter leaves the comment. For each commenter, we compute the number of pull requests commented by this commenter, divided by the total number of pull requests which are created in the commenter's active period. Without any recommendation approach, the commenter receives all pull requests created in the active period, but only leaves comments in a part of pull requests. We then aggregate across all commenters the percentage of pull requests in the project *rails*, and plot cumulative distribution functions (CDF) in Fig. 3. 35.669% of commenters only leave comments in less than 1% of pull requests created in their active periods. 91.704% of commenters leave comments in less than 20% of pull requests created in their active periods. The majority of commenters are interested in a small part of pull requests, and they do not need to receive all pull requests. Results further show that an automatic commenter recommendation approach is required to remind commenters of appropriate pull requests, and remove inappropriate pull requests.

3. Attributes in commenter recommendation

Fig. 4 presents the overall framework of commenter recommendation approach. When a contributor submits a new pull request, the recommendation approach firstly analyzes historical data, and finds developers who ever leave comments before. These former commenters become candidates for the recommendation. Secondly, every commenter's previous comments and corresponding pull requests are extracted from the historical data. Thirdly, the recommendation approach extracts attribute values from previous comments and pull requests. Fourthly, the score of a commenter is computed based on pull requests on which this commenter has left comments. Finally, we sort commenters by their scores, find commenters with the highest scores, and generate a suggestion list.

In the framework, the most important step is the attribute extraction. Approaches based on different attributes achieve various precisions and recalls, and the appropriate attribute selection is critical for accurate recommendation. In this paper, we mainly consider 4 kinds of attributes, including activeness, text similarity, file similarity and social relation. We take a further step and consider composite attributes, including time-based text similarity, time-based file similarity and time-based social relation. The time-based social relation approach is the state-of-the-art approach [5] proposed by Yu et al. We describe detailed definitions of attributes and why we choose these attributes in following subsections.

3.1. Activeness

In OSS projects, developers may not always be available, and may even leave projects [14]. Developers come and go as they please (especially the volunteers), resulting in high turnover [15].

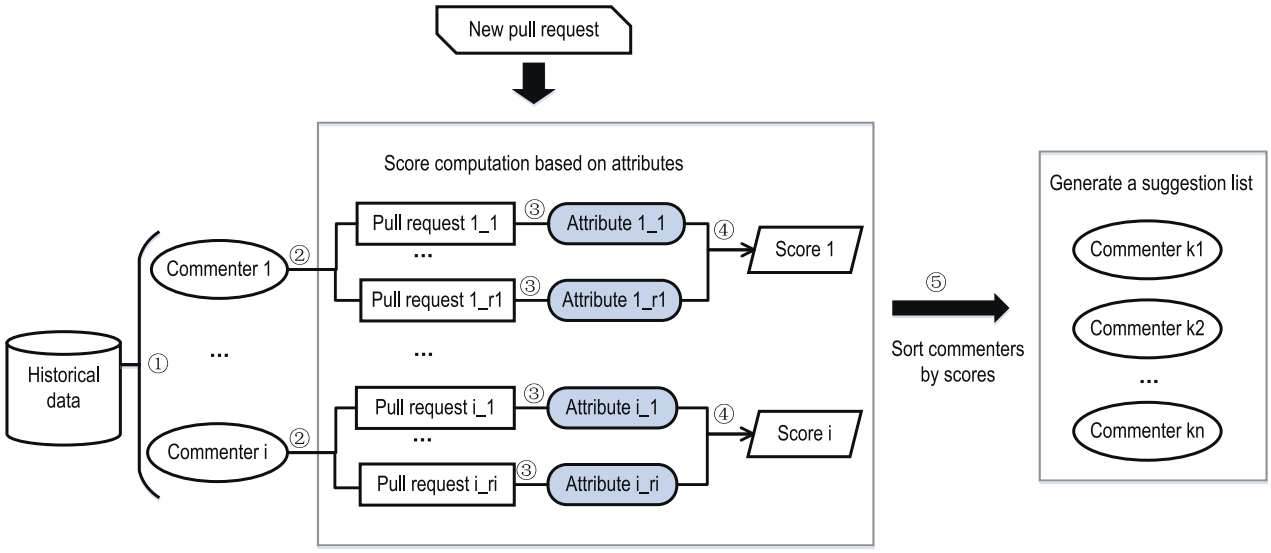


Fig. 4. Overall framework for commenter recommendation approach.

In some OSS projects, 80% of developers are either newcomers or leavers, which confirms that turnover is an important phenomenon [16]. Every three months one-third of the contributors discontinue to participate in the repository ITS of Google Chromium Project [17].

These prior findings motivate us to consider the activeness of commenters. Some commenters may be inactive or short-term active in providing comments. Recently active commenters may be inclined to comment on new pull requests.

In order to leverage this temporal locality, we use recent comments to measure the activeness of a commenter. As shown in Fig. 4, former commenters become candidates for the recommendation. Let us consider that there exists l potential commenters in a project, and denote the i^{th} commenter as C_i. $CSet_i$ includes pull requests on which the commenter C_i has ever left comments. For a pull request P_j ($P_j \in CSet_i$), its creation time is $time_{P_j}$. For a new pull request P_{new}, we compute the activeness of the commenter C_i, denoted as $activeness(P_{new}, C_i)$, as follows:

$$activeness(P_{new}, C_i) = \sum_{(P_j \in CSet_i) \wedge ((time_{P_{new}} - time_{P_j}) <= \gamma)} (time_{P_{new}} - time_{P_j})^{-\lambda} \quad (1)$$

where λ is the time-decaying parameter, and γ is used to describe the length of temporal window. For each pull request P_j, we use time-decaying function [18] to compute the weight of the pull request. The more recent pull request has higher weight. We exclude pull requests which are created more than γ days before the new pull request. In order to obtain activeness $activeness(P_{new}, C_i)$, we sum up results of pull requests on which the commenter C_i has left comments in recent γ days. The activeness depends on frequency and time of comments left by the commenter. If the commenter leaves comments frequently in recent pull requests, the activeness is high; otherwise, the activeness is low.

3.2. Text similarity

The text information is often used in the developer recommendation for bug resolution [19–21]. When contributors submit pull requests, they write titles to briefly introduce code changes they make. If contributors do not write titles, GitHub automatically generates titles to describe modified files. All pull requests have titles in GitHub. The semantic information may be useful in the commenter recommendation. The intuition is that similar pull requests are often described in a similar way, and a commenter may provide suggestions in similar pull requests.

We use titles to measure the text similarity between pull requests. For titles of pull requests, we make word segmentation, remove stop words and stem words. All remaining words construct the vocabulary. We use vector space model to represent each pull request as a weighted vector. The length of the vector is the number of words in the vocabulary. Each element in the vector is a word, and the value stands for the number of times that the word appears in the title of the pull request.

For a pull request P_j, its title vector is denoted as $title_{P_j}$. For a new pull request P_{new}, we compute the text similarity of the commenter C_i, denoted as $textSimilarity(P_{new}, C_i)$, as follows:

$$textSimilarity(P_{new}, C_i) = \sum_{P_j \in CSet_i} \frac{title_{P_{new}} \cdot title_{P_j}}{\| title_{P_{new}} \| \times \| title_{P_j} \|} \quad (2)$$

For each pull request P_j, we use cosine similarity [22] to compute the text similarity between the pull request P_j and the new pull request P_{new}. Then we compute the sum of pull requests on which the commenter C_i has left comments. The commenter will have high text similarity, if the commenter leaves comments on pull requests with similar titles as the new pull request.

We take a further step, and consider activeness and text similarity together. For a new pull request P_{new}, we compute the time-based text similarity of the commenter C_i, denoted as $textTime(P_{new}, C_i)$, as follows:

$$textTime(P_{new}, C_i) = \sum_{P_j \in CSet_i} \frac{title_{P_{new}} \cdot title_{P_j}}{\| title_{P_{new}} \| \times \| title_{P_j} \|} \times (time_{P_{new}} - time_{P_j})^{-\lambda} \quad (3)$$

The time-based text similarity combines the Eq. (1) and the Eq. (2) together.

3.3. File similarity

As described in the Section 2.2, we collected paths of modified files of pull requests. Distance between files was studied in latent social structure [23] and focus-shifting patterns of OSS developers [24]. Initial study [25] observed that files located in similar paths would be reviewed by similar experienced code reviewers.

Distance between files may be also helpful in the commenter recommendation. The intuition is that commenters are familiar with some files, and they may comment on same files or files in similar locations.

A pull request may include several paths of modified files. Let us consider that files in the pull request P_j construct the file set $FSet_{P_j}$, and denote the a^{th} file as $F^a_{P_j}$ ($F^a_{P_j} \in FSet_{P_j}$). The b^{th} file in the pull request P_{new} is defined as $F^b_{P_{new}}$. In order to compute the file distance between pull requests, we need to define the file similarity at first. Given two files $F^a_{P_j}$ and $F^b_{P_{new}}$, the path distance $fileDistance(F^a_{P_j}, F^b_{P_{new}})$ is calculated as follow:

$$pathDistance(F^a_{P_j}, F^b_{P_{new}}) = \frac{commonLength(F^a_{P_j}, F^b_{P_{new}})}{maxLength(F^a_{P_j}, F^b_{P_{new}})} \quad (4)$$

The max length $maxLength(F^a_{P_j}, F^b_{P_{new}})$ is the maximum value of lengths of file $F^a_{P_j}$ and file $F^b_{P_{new}}$. We also take pull request ID 1414 in the project *rails* as an example. This pull request has 1 modified file, namely "actionpack/lib/action_view/helpers/form_helper.rb". The file path include 5 substrings, including "actionpack", "lib", "action_view", "helpers" and "form_helper.rb". Therefore, the length of this file is 5. Another pull request ID 1523 has 1 file "actionpack/lib/action_dispatch/routing/mapper.rb", and its length is also 5. When we compare above 2 files, their maximum value of lengths is still 5.

The common length $commonLength(F^a_{P_j}, F^b_{P_{new}})$ is the number of the longest consecutive substrings that appear in the beginning of both files [25]. We still take above 2 files as an example. "actionpack" and "lib" are the longest consecutive substrings that appear in the beginning of both file paths. Therefore, the number of common substrings is 2. The path distance is computed as the ratio of the common length to the max length. The path distance between above 2 files is $\frac{2}{5}$.

Next, we define file distance between pull requests. For a new pull request P_{new} and a former pull request P_j, we compute pull request distance, denoted as $prDistance(P_j, P_{new})$, as follows:

$$prDistance(P_{new}, P_j) = \frac{\sum_{F^a_{P_j} \in FSet_{P_j}} \sum_{F^b_{P_{new}} \in FSet_{P_{new}}} pathDistance(F^a_{P_j}, F^b_{P_{new}})}{| FSet_{P_j} | \times | FSet_{P_{new}} |} \quad (5)$$

We sum up the file similarity of all possible pairs of files from pull requests P_{new} and P_j, and finally compute the average file similarity by dividing the sum with the number of possible pairs. The pull request distance is the average file similarity of all possible pairings of files.

Then, we compute the file similarity of the commenter C_i, denoted as $fileSimilarity(P_{new}, C_i)$, as follows:

$$fileSimilarity(P_{new}, C_i) = \sum_{P_j \in CSet_i} prDistance(P_{new}, P_j) \quad (6)$$

The commenter will have high file similarity, if the commenter leaves comments on pull requests with similar files as the new pull request.

Finally, we also consider activeness and file similarity together. For a new pull request P_{new}, we compute the time-based file similarity of the commenter C_i, denoted as $fileTime(P_{new}, C_i)$, as follows:

$$fileTime(P_{new}, C_i) = \sum_{P_j \in CSet_i} prDistance(P_{new}, P_j) \times (time_{P_{new}} - time_{P_j})^{-\lambda} \quad (7)$$

The time-based file similarity combines the Eq. (1) and the Eq. (6) together.

3.4. Social relation

Developer social networks are often used in quality prediction, defect prediction, bug triage and bug fixing [26]. Previous work uses comments to build social relationships and predict appropriate commenters of incoming pull requests [5]. The basic intuition is that developers who share common interests with a contributor are appropriate commenters. For commenter recommendation, the common interests among developers can be directly reflected by comment relationships between commenters and contributors. We consider the social relation attribute and make basic introduction in this subsection. Details can be found in the previous work [5].

Given a new pull request P_{new}, its contributor is defined as O_{new}. $OSet_{new}$ includes pull requests which are submitted by the contributor O_{new} before the new pull request P_{new}. As described in the Section 3.1, $CSet_i$ includes pull requests on which the commenter C_i has ever left comments. The intersection $OSet_{new} \bigcap CSet_i$ includes pull requests, which are submitted by the contributor O_{new} and the commenter C_i has ever left comments on. This intersection $OSet_{new} \bigcap CSet_i$ reflects the commenter C_i's interest in the contributor $OSet_{new}$. If the commenter C_i often leaves comments on pull requests submitted by the contributor $OSet_{new}$, the commenter C_i may be interested in this contributor and like to leave comments on his or her pull requests. A commenter can leave comments on a pull request for several times, and the number of comments is also considered in the previous work [5]. Let us consider that the commenter C_i leaves $n_{i,j}$ comments on the pull request P_j. For a new pull request P_{new}, we compute the relationship of the commenter C_i, denoted as $relation(P_{new}, C_i)$, as follows:

$$relation(P_{new}, C_i) = \sum_{P_j \in OSet_{new} \bigcap CSet_i} \sum_{k=1}^{n_{i,j}} \beta^{k-1} \quad (8)$$

where β is set as 0.8 [5]. If the commenter has left many comments on pull requests submitted by the contributor before, the commenter will have close relation with the contributor.

The previous work [5] also considers the time of comments. We compute the time-based relation of the commenter C_i, denoted as $relationTime(P_{new}, C_i)$, as follows:

$$relationTime(P_{new}, C_i) = \sum_{P_j \in OSet_{new} \bigcap CSet_i} \sum_{k=1}^{n_{i,j}} \beta^{k-1} \times t_{P_j,k} \quad (9)$$

where $t_{P_j,k}$ is a time-sensitive factor and its detailed description can be found in the Eq. (4) of the previous work [5]. The relation time based approach is the CN-based approach in the initial study [5]. The initial study [5] uses this equation to compute the time-based relation between commenters and the contributor, and recommends commenters with the high value. If no developers have ever commented on the contributor before, the initial study [5] utilizes co-occurrence pattern or popular stars in the community to make recommendation.

3.5. Approach

In above subsections, we describe definitions of attributes. We use these attributes to build different commenter recommendation approaches. As shown in Fig. 4, the main difference of approaches is the step 3, namely attribute extraction. For example, the activeness based approach uses the Eq. (1) to compute attribute values from comments; the text similarity based approach uses the Eq. (2) to compute attribute values for candidates; the text time based approach uses the Eq. (3) to compute attribute values for candidates. We use the similar way to define the file similarity based approach, the file time based approach, the relation based approach and the relation time based approach. As described in

the Section 3.4, the relation time based approach is the CN-based approach in the initial study [5]. In the Section 4, we compare the precision and recall of approaches based on different attributes, and choose the best attribute to make recommendation.

4. Experiments and results

In this section, we evaluate precisions and recalls of approaches based on different attributes. The experimental environment is a windows server 2012, 64-bit, Intel(R) Xeon(R) 1.90 GHz server with 24GB RAM. In this experimental environment, we compute results of all approaches. Since these approaches do not use techniques like machine learning, they do not cost much time or resource. We first present our experiment setup and evaluation metrics (Sections 4.1 and 4.2). We then present our experiment results (Sections 4.3–4.6).

4.1. Experiment setup

In order to simulate the usage of methods in practice, we sort all pull requests in chronological order of their creation time. For a given pull request, historical data in Fig. 4 includes all pull requests which are created before the given pull request. It ensures that only past pull requests are used to recommend commenters for pull requests submitted in the future. The experimental process is proceeded as follows: first of all, the historical data has the first pull request. Following the framework in Fig. 4, we use the historical data to generate a suggestion list for the second pull request. Next, the historical data has the first and the second pull requests. We use the updated historical data to generate a suggestion list for the third pull request. We use the similar way to recommend commenters for all pull requests. We compute precisions and recalls for pull requests created in each month, and then compute the average precision and recall of all months.

The time-decaying parameter λ in Eqs. (1), (3) and (7) is set as 1 by default. The temporal window length γ in Eq. (1) is set as no time limit by default, and we consider all pull requests on which the commenter has left comments before. In Sections 4.5 and 4.6, we explore how different parameter settings influence the precision and recall. Results show that the best precision and recall are always obtained when λ is set as 1; The activeness based approach achieves the best precision and recall when γ is set as 60 days or no time limit, but the difference between these two settings is small.

4.2. Evaluation metrics

In order to evaluate our method, we use the top-k precision and top-k recall, which are commonly used in evaluating recommendation approaches in the software engineering literature [5,19–21].

Given a pull request P_j, $ActualSet_j$ includes commenters who actually comment on this pull request. Note that the contributor who submits the pull request is not included in $ActualSet_j$. The contributor may leave comments as replies, which is different from suggestions in the code review. We do not consider the number of comments left by a developer. If a developer leaves several comments on a pull request, this developer is still considered as 1 commenter in $ActualSet_j$. We recommend the set of top-k commenters $RecSet_{j,k}$ for the pull request P_j.

Precision is the percentage of suggested developers who actually comment on the pull request. Given a pull request P_j, the top-k precision $Precision_{j,k}$ is defined as:

$$Precision_{j,k} = \frac{|ActualSet_j \bigcap RecSet_{j,k}|}{|RecSet_{j,k}|} \tag{10}$$

k is the number of recommended commenters, and we choose the k value to be 1, 2, 3 and 4, 5 in experiments. The top-k recommendation means that the approach recommends k commenters, and the top-k precision means the precision of the top-k recommendation. The top-k precision calculates the percentage of k recommended developers who really comment on the pull request. The higher the top-k precision values, the better a recommendation approach performs.

Recall is the percentage of real commenters who are actually suggested. Given a pull request P_j, the top-k recall $Recall_{j,k}$ is given by:

$$Recall_{j,k} = \frac{|ActualSet_j \bigcap RecSet_{j,k}|}{|ActualSet_j|} \tag{11}$$

The top-k recall means the recall of the top-k recommendation.

In order to compare two methods, we define the gain to compare how the method 1 outperforms the method 2. As described in the initial study [27], the gain for performance is defined as follows:

$$Gain_{\text{Precision,j,k}} = \frac{(Precision_{j,k}(1) - Precision_{j,k}(2))}{Precision_{j,k}(2)} \tag{12}$$

$$Gain_{\text{Recall,j,k}} = \frac{(Recall_{j,k}(1) - Recall_{j,k}(2))}{Recall_{j,k}(2)} \tag{13}$$

where $Precision_{j,k}(1)$ and $Recall_{j,k}(1)$ are top-k performance for method 1, and $Precision_{j,k}(2)$ and $Recall_{j,k}(2)$ are top-k performance for method 2. If the gain value is above 0, it means the method 1 has better results than the method 2; otherwise the method 2 has better results.

4.3. Performance of different attributes

RQ1: What are the precision and recall of approaches based on different attributes? Which attribute has the best precision and recall in the commenter recommendation?

In the Section 3, we introduce 4 kinds of attributes in the commenter recommendation, including activeness, text similarity, file similarity and social relation. Approaches based on different attributes achieve various precisions and recalls, and appropriate attribute selection is critical for accurate recommendation. In this subsection, we aim to evaluate the performance of different attributes, and find the best performing attribute in the commenter recommendation.

We evaluate approaches based on various attributes, and record the top-k precisions in Table 3. Best results are shown in bold. 4 rows in the bottom of Table 3 shows average precisions for pull requests in 8 projects. The activeness based approach achieves the highest precisions on average. In the project *symfony*, the top-1 precision is 0.591, 0.539, 0.524 and 0.372 for activeness based approach, test similarity based approach, file similarity based approach and relation based approach, respectively. The activeness based approach has the highest top-1 precision. The activeness based approach also has the highest top-2, top-3, top-4 and top-5 precisions. In 8 projects, the activeness based approach achieves the top-3 precision of 0.276, 0.386, 0.389, 0.516, 0.322, 0.572, 0.428, 0.402, which is hither than other approaches. The activeness based approach has the highest precision in the commenter recommendation. In projects *symphony* and *zf2*, the text similarity based approach has higher precision than the file similarity based approach; in other 6 projects, the file similarity based approach is better than the text similarity based approach. Some pull requests have short titles and provide limited information for text similarity calculation, which may cause the low precision of the text similarity based approach. The relation based approach always has the lowest precision in all projects.

Table 3
Top-*k* precisions of approaches based on different attributes. (Best results in bold.)

Project	Approach	Top-1	Top-2	Top-3	Top-4	Top-5
rails	activeness	**0.393**	**0.324**	**0.276**	**0.238**	**0.209**
	textSimilarity	0.294	0.243	0.21	0.184	0.166
	fileSimilarity	0.323	0.264	0.229	0.198	0.176
	relation	0.136	0.11	0.091	0.084	0.077
symfony	activeness	**0.591**	**0.495**	**0.386**	**0.318**	**0.274**
	textSimilarity	0.539	0.47	0.362	0.297	0.252
	fileSimilarity	0.524	0.462	0.346	0.286	0.244
	relation	0.372	0.276	0.234	0.239	0.208
scala	activeness	**0.654**	**0.475**	**0.389**	**0.341**	**0.297**
	textSimilarity	0.449	0.368	0.332	0.295	0.264
	fileSimilarity	0.463	0.365	0.335	0.307	0.273
	relation	0.253	0.224	0.188	0.172	0.16
bitcoin	activeness	**0.698**	**0.599**	**0.516**	**0.451**	**0.401**
	textSimilarity	0.525	0.485	0.436	0.393	0.351
	fileSimilarity	0.594	0.517	0.454	0.406	0.368
	relation	0.351	0.288	0.282	0.262	0.237
zf2	activeness	**0.569**	**0.399**	**0.322**	**0.266**	**0.228**
	textSimilarity	0.479	0.313	0.245	0.207	0.184
	fileSimilarity	0.463	0.294	0.232	0.199	0.176
	relation	0.455	0.278	0.198	0.16	0.138
akka	activeness	**0.798**	**0.672**	**0.572**	**0.497**	**0.428**
	textSimilarity	0.64	0.562	0.501	0.45	0.407
	fileSimilarity	0.657	0.575	0.515	0.459	0.416
	relation	0.575	0.396	0.336	0.301	0.268
cakephp	activeness	**0.685**	**0.516**	**0.428**	**0.367**	**0.313**
	textSimilarity	0.636	0.476	0.388	0.325	0.273
	fileSimilarity	0.678	0.492	0.411	0.353	0.295
	relation	0.478	0.315	0.25	0.209	0.179
gitlabhq	activeness	**0.64**	**0.496**	**0.402**	**0.342**	**0.294**
	textSimilarity	0.467	0.374	0.308	0.258	0.222
	fileSimilarity	0.539	0.413	0.328	0.277	0.238
	relation	0.407	0.292	0.22	0.18	0.154
(average)	activeness	**0.588**	**0.468**	**0.387**	**0.332**	**0.288**
	textSimilarity	0.468	0.385	0.326	0.282	0.249
	fileSimilarity	0.491	0.395	0.334	0.291	0.256
	relation	0.331	0.243	0.202	0.183	0.163

Table 4
Top-*k* recalls of approaches based on different attributes. (Best results in bold.)

Project	Approach	Top-1	Top-2	Top-3	Top-4	Top-5
rails	activeness	**0.235**	**0.381**	**0.475**	**0.538**	**0.583**
	textSimilarity	0.173	0.278	0.36	0.414	0.461
	fileSimilarity	0.197	0.312	0.396	0.454	0.499
	relation	0.075	0.117	0.147	0.181	0.21
symfony	activeness	**0.327**	**0.518**	**0.593**	**0.641**	**0.682**
	textSimilarity	0.283	0.475	0.544	0.594	0.626
	fileSimilarity	0.28	0.477	0.537	0.587	0.62
	relation	0.19	0.275	0.344	0.476	0.512
scala	activeness	**0.38**	**0.517**	**0.613**	**0.703**	**0.76**
	textSimilarity	0.232	0.381	0.51	0.605	0.671
	fileSimilarity	0.245	0.385	0.523	0.639	0.705
	relation	0.14	0.235	0.286	0.342	0.39
bitcoin	activeness	**0.326**	**0.533**	**0.66**	**0.745**	**0.811**
	textSimilarity	0.224	0.406	0.533	0.625	0.686
	fileSimilarity	0.267	0.439	0.559	0.651	0.727
	relation	0.148	0.237	0.345	0.419	0.472
zf2	activeness	**0.409**	**0.548**	**0.644**	**0.696**	**0.735**
	textSimilarity	0.343	0.434	0.504	0.556	0.606
	fileSimilarity	0.339	0.417	0.49	0.551	0.6
	relation	0.329	0.398	0.419	0.444	0.47
akka	activeness	**0.435**	**0.662**	**0.791**	**0.887**	**0.933**
	textSimilarity	0.337	0.544	0.69	0.804	0.887
	fileSimilarity	0.343	0.547	0.698	0.807	0.899
	relation	0.309	0.389	0.47	0.542	0.603
cakephp	activeness	**0.416**	**0.596**	**0.714**	**0.796**	**0.841**
	textSimilarity	0.382	0.553	0.65	0.705	0.735
	fileSimilarity	0.41	0.577	0.695	0.773	0.804
	relation	0.293	0.369	0.428	0.466	0.491
gitlabhq	activeness	**0.368**	**0.55**	**0.65**	**0.721**	**0.757**
	textSimilarity	0.265	0.41	0.497	0.556	0.591
	fileSimilarity	0.312	0.46	0.54	0.605	0.645
	relation	0.225	0.323	0.363	0.393	0.412
(average)	activeness	**0.339**	**0.509**	**0.61**	**0.681**	**0.728**
	textSimilarity	0.259	0.408	0.505	0.574	0.625
	fileSimilarity	0.277	0.425	0.524	0.6	0.653
	relation	0.185	0.259	0.313	0.371	0.408

Table 4 shows the top *k* recalls of approaches based on different attributes. In 8 projects, the activeness based approach achieves the top-3 recall of 0.475, 0.593, 0.613, 0.66, 0.644, 0.791, 0.714, 0.65. Results also show that the activeness based approach has the highest recall, and the relation based approach has the lowest recall. We compute the recall gain to compare how the activeness based approach outperforms another approach. In the project *rails*, the activeness based approach even outperforms the relation based approach by 213.65%, 224.09%, 224.091%, 196.973% and 178.214% in terms of top-1, top-2, top-3, top-4 and top-5 prediction recalls, respectively. 4 rows in the bottom of Table 4 shows average recalls for pull requests in 8 projects. The activeness based approach achieves the highest recalls on average. The activeness attribute has obvious better in the commenter recommendation.

Results in Tables 3 and 4 show that the activeness based approach has the highest precision and recall in all projects. The activeness is the most important attribute in the commenter recommendation. Given a new pull request, the recommendation approach should firstly consider which developers are recently active and may have time to make comments in code review. In the majority of projects, the file similarity based approach has higher precision and recall than the text similarity based approach. The relation based approach always has the worse performance.

> For 8 projects, the activeness based approach achieves the top-3 precision of 0.276, 0.386, 0.389, 0.516, 0.322, 0.572, 0.428, 0.402, and achieves the top-3 recall of 0.475, 0.593, 0.613, 0.66, 0.644, 0.791, 0.714, 0.65. The activeness based approach outperforms approaches based on other attributes by a substantial margin. This indicates that the activeness is the most important attribute in the commenter recommendation.

4.4. Performance of composite attributes

RQ2: What are the precision and recall of composite approaches? Does the combination of attributes improve the precision and recall?

In the Section 4.3, results show that the activeness attribute has the best precision and recall in the commenter recommendation. We wonder whether better precision and recall can be obtained, when we combine the activeness attribute and other attributes. In the Section 3, we have defined time-based attributes in Eqs. (3), (7) and (9), and use these attributes to build corresponding approaches. The relation time based approach is the CN-based approach in the initial study [5]. We evaluate these approaches and record the top-*k* prediction precisions and recalls.

Table 5 shows the top-*k* precisions of composite approaches which combine activeness and other attributes. In order to make comparison, we also put results of the activeness based approach. In projects *rails, symfony, zf2* and *gitlabhq*, the precision of the activeness based approach is still higher than other composite approaches. In these 4 projects, the combination of the activeness and other attributes does not bring benefits for the commenter recommendation. In the project *scala*, the top-2 and top-3 precisions are 0.481 and 0.395 for the file time based approach, while the top-2 and top-3 precisions are 0.475 and 0.389 for the activeness based approach. In the project *bitcoin*, the file time based

Table 5
Top-*k* precisions of composite approaches which combine activeness and other attributes. (Best results in bold.)

Project	Approach	Top-1	Top-2	Top-3	Top-4	Top-5
rails	activeness	**0.393**	**0.324**	**0.276**	**0.238**	**0.209**
	textTime	0.287	0.245	0.212	0.187	0.169
	fileTime	0.367	0.304	0.26	0.222	0.194
	relationTime	0.158	0.123	0.099	0.087	0.08
symfony	activeness	**0.591**	**0.495**	**0.386**	**0.318**	**0.274**
	textTime	0.523	0.446	0.36	0.3	0.258
	fileTime	0.569	0.481	0.378	0.311	0.268
	relationTime	0.481	0.366	0.296	0.24	0.2
scala	activeness	**0.654**	0.475	0.389	**0.341**	**0.297**
	textTime	0.591	0.449	0.377	0.326	0.283
	fileTime	0.639	**0.481**	**0.395**	0.336	0.296
	relationTime	0.392	0.327	0.311	0.274	0.243
bitcoin	activeness	0.698	0.599	**0.516**	**0.451**	**0.401**
	textTime	0.613	0.529	0.45	0.397	0.355
	fileTime	**0.709**	**0.601**	0.507	0.441	0.389
	relationTime	0.479	0.432	0.364	0.333	0.302
zf2	activeness	**0.569**	**0.399**	**0.322**	**0.266**	**0.228**
	textTime	0.502	0.355	0.286	0.239	0.207
	fileTime	0.515	0.367	0.297	0.244	0.208
	relationTime	0.468	0.298	0.216	0.176	0.151
akka	activeness	**0.798**	**0.672**	**0.572**	**0.497**	0.427
	textTime	0.738	0.646	0.556	0.484	0.422
	fileTime	0.769	0.667	0.571	0.492	**0.428**
	relationTime	0.546	0.524	0.431	0.401	0.359
cakephp	activeness	0.685	**0.516**	**0.428**	**0.367**	**0.313**
	textTime	0.544	0.417	0.349	0.304	0.265
	fileTime	0.672	0.505	0.413	0.352	0.299
	relationTime	**0.689**	0.447	0.342	0.275	0.227
gitlabhq	activeness	**0.64**	**0.496**	**0.402**	**0.342**	**0.294**
	textTime	0.506	0.412	0.338	0.287	0.244
	fileTime	0.619	0.476	0.377	0.317	0.271
	relationTime	0.472	0.307	0.225	0.184	0.154
(average)	activeness	**0.588**	**0.468**	**0.387**	**0.332**	**0.288**
	textTime	0.5	0.408	0.342	0.295	0.258
	fileTime	0.567	0.455	0.376	0.319	0.277
	relationTime	0.406	0.315	0.257	0.222	0.194

Table 6
Top-*k* recalls of composite approaches which combine activeness and other attributes. (Best results in bold.)

Project	Approach	Top-1	Top-2	Top-3	Top-4	Top-5
rails	activeness	**0.235**	**0.381**	**0.475**	**0.538**	**0.583**
	textTime	0.171	0.282	0.365	0.42	0.47
	fileTime	0.218	0.356	0.449	0.502	0.544
	relationTime	0.09	0.134	0.16	0.188	0.217
symfony	activeness	**0.327**	**0.518**	**0.593**	**0.641**	**0.682**
	textTime	0.279	0.453	0.544	0.599	0.638
	fileTime	0.313	0.508	0.586	0.634	0.674
	relationTime	0.244	0.362	0.435	0.466	0.484
scala	activeness	**0.38**	0.517	0.613	**0.703**	**0.76**
	textTime	0.332	0.484	0.596	0.674	0.721
	fileTime	0.372	**0.53**	**0.632**	0.698	0.759
	relationTime	0.199	0.322	0.475	0.554	0.607
bitcoin	activeness	0.326	**0.533**	**0.66**	**0.745**	**0.811**
	textTime	0.279	0.461	0.562	0.644	0.701
	fileTime	**0.33**	0.527	0.641	0.717	0.77
	relationTime	0.207	0.363	0.452	0.54	0.603
zf2	activeness	**0.409**	**0.548**	**0.644**	**0.696**	**0.735**
	textTime	0.36	0.499	0.577	0.632	0.671
	fileTime	0.382	0.516	0.61	0.659	0.694
	relationTime	0.34	0.415	0.449	0.479	0.506
akka	activeness	**0.435**	**0.662**	**0.791**	**0.887**	**0.933**
	textTime	0.385	0.624	0.767	0.862	0.915
	fileTime	0.407	0.645	0.779	0.865	0.921
	relationTime	0.3	0.52	0.619	0.74	0.808
cakephp	activeness	0.416	**0.596**	**0.714**	**0.796**	**0.841**
	textTime	0.32	0.479	0.583	0.664	0.712
	fileTime	0.408	0.584	0.692	0.763	0.801
	relationTime	**0.418**	0.507	0.569	0.601	0.616
gitlabhq	activeness	**0.368**	**0.55**	**0.65**	**0.721**	**0.757**
	textTime	0.275	0.448	0.538	0.605	0.636
	fileTime	0.36	0.528	0.609	0.672	0.709
	relationTime	0.262	0.331	0.363	0.4	0.416
(average)	activeness	**0.339**	**0.509**	**0.61**	**0.681**	**0.728**
	textTime	0.28	0.437	0.534	0.602	0.648
	fileTime	0.326	0.495	0.594	0.655	0.701
	relationTime	0.225	0.327	0.396	0.447	0.483

approach has better precisions than the activeness based approach with top-1 and top-2 recommendation. In the project *akka*, the file time based approach outperforms the activeness based approach by 0.23% in the term of top-5 predication precision. In projects *scala, bitcoin* and *akka*, the combination of the activeness and file similarity sometimes increases the precision, but the gain is low for the commenter recommendation. In the project *cakephp*, the relation time based approach outperforms the activeness based approach by 0.58% in the term of top-1 predication precision. The combination of activeness and relation does not obviously increase the precision.

Table 6 shows top-*k* recalls of composite approaches. In order to make comparison, we also put results of the activeness based approach. In projects *rails, symfony, zf2, akka* and *gitlabhq*, the activeness based approach has higher recall than other composite approaches. In comparison with the activeness based approach, the file time based approach improves the recall by 2.514% and 3.1% for top-2 and top-3 recommendation in the project *scala*. In the project *bitcoin*, the file time based approach outperforms the activeness based approach by 1.227% in the term of top-1 predication recall. In projects *scala* and *bitcoin*, the combination of the activeness and file similarity sometimes slightly improves the recall. In the project *cakephp*, the relation time based approach outperforms the activeness based approach by 0.481% in the term of top-1 predication recall.

Results in Tables 5 and 6 show that composite approaches occasionally have better precision and recall than the activeness based approach, but the improvement is not obvious. In most of the time, approaches based on activeness and other attributes obtain worse precision and recall than the approach only based on the activeness. These results show that more attributes do not increase the precision or the recall, but often cause worse performance. The approach based on the activeness alone is enough to recommend commenters. The activeness is so dominantly influencing on the commenter ranking. This is probably due to high turnover of developers [15]. In some OSS projects, 80% of developers are either newcomers or leavers [16]. In GitHub, only experienced and excellent developers are chosen as core members, but any developers can leave comments and become commenters. Some developers join projects for a short time, and leave comments in some pull requests. For these short-term commenters, their activeness is more important than other attributes.

The relation time based approach is the CN-based approach in the initial study [5]. Tables 5 and 6 show that the activeness based approach always outperforms the CN-based approach [5], except the project *cakephp*. In comparison with CN-based approach (the relation time based approach), the activeness based approach improves the top-3 precision by 178.788%, 30.41%, 25.08%, 41.76%, 49.07%, 32.71%, 25.15%, 78.67%, and improves the top-3 recall by 196.875%, 36.32%, 29.05%, 46.02%, 43.43%, 27.79%, 25.483%, 79.06% for 8 projects.

Tables 5 and 6 show the precision and recall of CN-based approach, which are different from results in Table III of the previous work [5]. This is because this paper and the previous work [5] have different datasets. Firstly, the initial study [5] and this paper collected datasets in different time periods. Yu et al. collected pull requests from January 2012 to October 2013, while we collected pull requests from project creation time to July 2014. Secondly, the initial study [5] and this paper had different methods to filter datasets. Yu et al. deleted pull requests with less than 4 com-

Table 7
Correlation between the number of commenters and prediction results of the activeness based approach.

	Top 1	Top 2	Top 3	Top 4	Top 5
Precision	−0.907	−0.727	−0.692	−0.68	−0.663
Recall	−0.87	−0.854	−0.845	−0.85	−0.86

ments, and pull requests with less than 10 words in descriptions and titles. We ignored pull requests with no comments, and pull requests only with comments from their contributors who created pull requests. Though our datasets and previous work's datasets [5] are different, CN-based approach is a state-of-the-art approach, and it is not limited to some specific conditions [5]. Previous work [5] clearly described details of CN-based approach. According to previous work [5], we implement CN-based approach by ourselves, and compare it with other approaches based on our datasets. In our experiments, all approaches are evaluated with the same datasets.

In Tables 5 and 6, different projects have various results of the activeness based approach. The project *akka* always has the highest precision and recall, and the project *rails* has the lowest precision and recall. Table 1 shows the number of commenters in each project. Pearson's correlation coefficient provides a measure of the strength of linear association between two variables [28]. In Table 7, we use Pearson's correlation coefficient to measure the correlation between the number of commenters and the precision, and the correlation between the number of commenters and the recall of the activeness based approach. The Pearson correlation coefficient between the number of commenters and the top-1 precision is as low as −0.907. It means that the more commenter a project has, the lower the top-1 precision is. Results show that there is strongly negative correlation between the number of commenters and the precision of the activeness based approach. The number of commenters and the recall is also strongly negative correlated. The activeness based approach works better in projects with fewer commenters. When projects have many commenters, these projects have many candidates for the recommendation, which may affect prediction results.

> The activeness based approach achieves better precision and recall than composite approaches, including CN-based approach [5].

4.5. Effect of varying the time-decaying parameter

RQ3: What is the best setting of time-decaying parameter?

The time-decaying parameter λ is used in the activeness based approach, the text time based approach and the file time based approach. By default, we set the time-decaying parameter λ as 1. In this subsection, we investigate the effect of the time-decaying parameter λ on the performance of these approaches. We increase λ values from 0.5 to 5 with an interval of 0.5, and evaluate the precision and recall of approaches with different λ values. Projects have similar results, and thus we only plot results of projects *rails* and *symfony* in the paper. We put results of 8 projects in the web link.[3]

Fig. 5 presents precisions of the activeness based approach with different time-decaying parameter λ. In the project *rails*, we notice that all top *k* precisions obviously increase when we increase λ from 0.5 to 1, and decrease substantially when we increase λ from 1 to 2.5. Then top *k* precisions drops slightly with the parameter λ varying from 2.5 to 5. The activeness based approach achieves the highest precision with the time-decaying parameter λ as 1. In Fig. 5, the project *symfony* has similar trend of precisions with the parameter λ varying from 0.5 to 5.

Fig. 6 presents recalls of the activeness based approach with different time-decaying parameter λ. Recall results are similar as precision results. The setting of $\lambda = 1$ can also achieve good recalls in projects *rails* and *symfony*.

According to results in Figs. 5 and 6, the activeness based approach always has the best precision and recall when the time-decaying parameter λ is set as 1. We also evaluate the performance of the text time based approach and the file time based approach, and we obtain similar results. Therefore, time-decaying parameter λ is set as 1 by default.

> The best precision and recall are always obtained when time-decaying parameter λ is set as 1.

Table 8
Precisions with different temporal window length γ for the activeness based approach. (Best results in bold.)

Project	Temporal window	Top-1	Top-2	Top-3	Top-4	Top-5
rails	7 days	0.364	0.297	0.259	0.222	0.194
	15 days	0.384	0.314	0.271	0.232	0.201
	30 days	0.384	0.319	0.275	0.236	0.206
	60 days	0.388	0.322	**0.277**	**0.239**	**0.209**
	no time limit	**0.393**	**0.324**	0.276	0.238	**0.209**
symfony	7 days	0.557	0.467	0.37	0.301	0.257
	15 days	0.568	0.48	0.383	0.312	0.265
	30 days	0.58	0.49	0.386	0.316	0.27
	60 days	0.582	**0.496**	**0.387**	**0.318**	0.272
	no time limit	**0.591**	0.495	0.386	**0.318**	**0.274**
zf2	7 days	0.589	0.392	0.3	0.241	0.201
	15 days	0.606	0.406	0.314	0.254	0.213
	30 days	0.608	**0.408**	0.319	0.258	0.219
	60 days	**0.612**	0.407	**0.322**	0.261	0.222
	no time limit	0.569	0.399	**0.322**	**0.266**	**0.228**

4.6. Effect of temporal window length

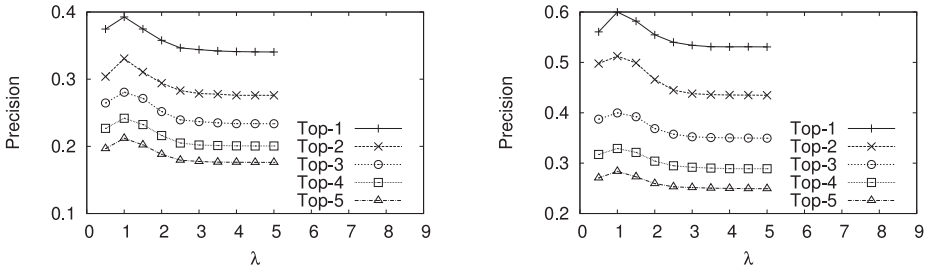
RQ4: What is the best setting of temporal window length?

The temporal window length γ is used in the activeness based approach to exclude pull requests which are created more than γ days before the new pull request. By default, we set the temporal window length γ as no time limit, and consider all pull requests on which the commenter has left comments before. In this subsection, we investigate the effect of the temporal window length γ on the precision and recall of the activeness based approach. We compute the approach performance with different γ values, including 7 days, 15 days, 30 days, 60 days and no time limit. We plot results of projects *rails, symfony* and *zf2* in the paper. The project *cakephp* has similar results as the project *zf2*, and other projects have similar results as projects *rails* and *symfony*. We plot their results in the web link [4].

Tables 8 and 9 plot precisions and recalls of the activeness based approach with different temporal window length γ. In projects *rails* and *symfony*, the activeness based approach achieves

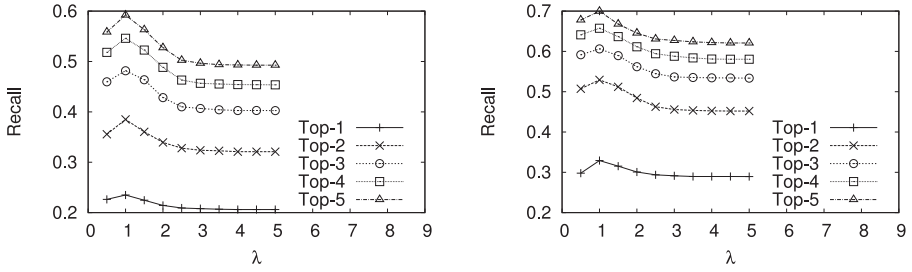
[3] https://github.com/researchjingjiang/cr/blob/master/timeDecaying.pdf, April 2016.

[4] https://github.com/researchjingjiang/cr/blob/master/windowLength.pdf, April 2016.



(*a*) Project *rails* (*b*) Project *symfony*

Fig. 5. Precisions with time-decaying parameter λ varying from 0.5 to 5 for the activeness based approach.



(*a*) Project *rails* (*b*) Project *symfony*

Fig. 6. Recalls with time-decaying parameter λ varying from 0.5 to 5 for the activeness based approach.

Table 9
Recalls with different temporal window length γ for the activeness based approach. (Best results in bold.)

Project	Temporal window	Top-1	Top-2	Top-3	Top-4	Top-5
rails	7 days	0.219	0.347	0.442	0.5	0.54
	15 days	0.233	0.37	0.466	0.522	0.559
	30 days	0.231	0.374	0.473	0.53	0.575
	60 days	0.232	0.376	**0.476**	0.537	**0.583**
	no time limit	**0.235**	**0.381**	0.475	**0.538**	**0.583**
symfony	7 days	0.306	0.484	0.565	0.607	0.638
	15 days	0.311	0.499	0.587	0.63	0.659
	30 days	0.318	0.511	0.591	0.636	0.672
	60 days	0.321	**0.519**	**0.593**	0.64	0.677
	no time limit	**0.327**	0.518	**0.593**	**0.641**	**0.682**
zf2	7 days	0.449	0.566	0.634	0.67	0.692
	15 days	0.463	0.588	0.661	0.707	0.734
	30 days	0.465	**0.593**	0.67	0.714	0.754
	60 days	**0.469**	0.592	**0.682**	**0.726**	**0.759**
	no time limit	0.409	0.548	0.644	0.696	0.735

the highest precision and recall with temporal window length γ as 60 days or no time limit. The difference between the setting of 60 days and no time limit is small. Other projects have similar results as projects *rails* and *symfony*, except projects *zf2* and *cakephp*.

In the project *zf2*, the top-1 precision is 0.612 with temporal window length γ as 60 days, while the top-1 precision is only 0.569 with temporal window length γ as no time limit. The precision difference between the setting of 60 days and no time limit is small for top-2, top-3, top-4 and top-5 recommendation. In comparison with γ as no time limit, the activeness based approach achieves much higher recalls with γ as 60 days.

Tables 8 and 9 show that in most of projects, the activeness based approach has the best precision and recall with temporal window length γ as 60 days or no time limit, and difference between two settings is small. Therefore, temporal window length γ is set as no time limit by default. Results of the project *zf2* imply that in practice, OSS projects should run experiments to define the best setting of temporal window length.

> The best precision and recall are always obtained when temporal window length γ is set as 60 days or no time limit, and difference between two settings is small.

5. Threats to validity

Threats to internal validity relate to experimenter bias and errors. We use a set of scripts to download and process the large GitHub data. We have checked these scripts and fixed errors that we found. However, we may still ignore few errors.

Threats to external validity relates to the generalizability of our study. Firstly, our experimental results are limited to 8 popular projects. We find that activeness based approach has higher precision and recall than CN-based approach, which are based on 8 projects in our datasets. We cannot claim that the same results would be achieved in other projects. Our future work will focus on the evaluation in other projects to better generalize results of our method. We will conduct broader experiments to validate whether activeness is always the most significant factor in commenter recommendation. Secondly, our empirical findings are based on OSS projects in GitHub, and it is unknown whether our results can be generalized to other OSS platforms. In the future, we plan to study a similar set of research questions in other platforms, and compare their results with our findings in GitHub. Thirdly, we mainly evaluate whether one of top k recommended commenters really leave comments. We do not distinguish commenters by the number of comments either. The amount of comments is beyond the scope of this paper.

Threats to construct validity refer to the suitability of our evaluation measures. We use precision and recall, which are also used by previous works to evaluate the effectiveness of a commenter recommendation approach [5], and various automated software engineering techniques [19,20,29]. Therefore, we believe there is little threat to construct validity.

Table 10
Percentage of pull requests which have a specific number of commenters .

Project	The number of commenters				
	1	2	3	4	≥5
rails	43.621	25.165	13.905	7.37	9.939
symfony	37.533	26.224	15.755	8.974	11.514
scala	31.121	33.491	19.046	9.509	6.833
bitcoin	21.665	26.496	21.197	14.464	16.178
zf2	56.153	25.95	10.059	4.387	3.451
akka	23.548	26.884	21.115	15.659	12.794
cakephp	42.838	26.667	16.436	7.525	6.534
gitlabhq	32.766	33.182	16.364	8.466	9.222

Table 11
Correlation between the number of commenters and evaluation time .

Project	Pearson's correlation coefficient
rails	0.329
symfony	0.328
scala	0.284
bitcoin	0.415
zf2	0.32
akka	0.178
cakephp	0.226
gitlabhq	0.352

6. Discussion

6.1. The usefulness of comments

Some commenters may discuss trivial, tangential, or unrelated issues, and their comments are useless for the code review [30]. In this subsection, we randomly select 400 comments, and manually analyze their usefulness. Table 1 shows that our datasets have 206,664 comments. 400 comments from a population of 206,664 yield a 95% confidence level with 4.9% error margin.

Previous work [30] proposes an approach for assessing discussion usefulness. A comment is considered as useful if it is directly contributes to improving code changes in Gerrit [30] . In GitHub, we find that some commenters do not provide suggestions for improvement, but they support code changes and suggest core members to accept code changes. Their comments are also helpful for core members to make decision. Therefore, we define a useful comment as one that directly contributes to the evaluation of a pull request, and a useless comment as one that does not. The manual classification is subjective. In order to reduce human errors, three authors participate in the classification. The first, the third and the fifth authors read comments, and independently decide whether comments are useful or not. If at least two authors consider a comment as useful, the comment is finally regarded as useful; otherwise, the comment is finally regarded as useless.

We manually classify 400 comments into two categories. We find that 399 (99.75%) comments are considered as useful for the evaluation of pull requests. Only 1 (0.25%) comment is considered as useless. The useless comment is in the project *zf2*.[5] The commenter *EvanDotPro* says that "I'll try to take a closer look at this over the weekend." This comment only describes the plan for code review, but does not provide any valuable information. Results show that almost all of comments are useful and contribute to the evaluation of pull requests. In this paper, commenter recommendation approaches do not distinguish useful comments from useless comments. Few useless comments have almost no impacts on commenter recommendation approaches.

6.2. The number of commenters

In this subsection, we discuss the number of commenters and its relationship with pull request evaluation. As described in the Section 2.2, our datasets ignored pull requests with no comments. For each pull request with comments, we compute the number of commenters who participate in the evaluation. We make statistics of the number of commenters, and present results in Table 10. In the project *rails*, 43.621% of pull requests have 1 commenter, and 25.165% of pull request have 2 commenters. Only in 9.939% of pull requests, the number of commenters is great than or equal to 5. The majority of pull requests have fewer than 5 commenters. Other projects have similar results.

We take a further step, and study the relationship between the number of commenters and the evaluation time. The evaluation time is defined as the interval between a pull request's creation time and closed time. In Table 11, we use Pearson's rank correlation coefficient to measure the correlation between the number of commenters and evaluation time. Except the project *akka*, other projects have Pearson's correlation coefficients larger than 0.2, and the number of commenters is weakly positively correlated with the evaluation time. Pull requests with more commenters sometimes need more evaluation time.

We find that the minority of pull requests have more than 5 commenters, and pull requests with more commenters sometimes cost more time for evaluation. Previous work [2] observes that pull requests with many associated comments are much less likely to be accepted. Uncertain pull requests tend to require negotiation, and pull requests with lots of comments may signal controversy [2]. Commenter recommendation approaches aim to remind commenters of appropriate pull requests, rather than decreasing the number of commenters or evaluation time. The evaluation time is impacted by multiple factors, such as the developer's previous track record [1].

6.3. Reduction of workload

We discuss how the recommendation approach reduces workload of commenters. As described in the Section 2.2, the project *rails* received 131 pull requests, and 201 commenters were active in April 2012. The developer *tenderlove* received 131 pull requests, and tried to find appropriate pull requests from 131 pull requests. In fact, *tenderlove* only left comments in 12 pull requests, and he did not need to receive all pull requests. With the top-3 recommendation of activeness based approach, *tenderlove* only received 22 pull requests, which was much less than 131 pull requests.

We take a further step and study the percentage of recommended pull requests. For every commenter, the active period in a project is defined as the time between this commenter's first comment and the last comment in this project. If the commenter only leaves 1 comment, the active period is the day when the commenter leaves the comment. For each commenter, we compute the total number of pull requests which are created in the commenter's active period. Without any recommendation approach, the commenter receives all pull requests created in the active period. The activeness based approach recommends a few appropriate pull requests to commenters, rather than all pull requests. We compute the number of pull requests which are recommended to a specific commenter by the activeness based approach. Then we compute the total number of recommended pull requests, divided by the total number of pull requests in their active periods. Table 12 shows the percentage of recommended pull requests. In the project *rails*, the activeness based approach only recommends 0.656% of pull requests created in commenters' active periods with

[5] https://github.com/zendframework/zf2/pull/3643.

Table 12
Percentage of recommended pull requests .

Project	Top-1	Top-2	Top-3	Top-4	Top-5
rails	0.219	0.438	0.656	0.875	1.093
symfony	0.488	0.976	1.446	1.93	2.408
scala	1.813	3.626	5.438	7.249	9.06
bitcoin	1.256	2.511	3.764	5.013	6.255
zf2	1.053	2.042	3.009	3.985	4.939
akka	2.929	5.853	8.744	11.625	14.434
cakephp	1.719	3.432	5.134	6.823	8.476
gitlabhq	1.221	2.423	3.579	4.713	5.754

top-3 recommendation. It greatly reduces the number of pull requests which commenters receive. In these 8 projects, the activeness based approach recommends less than 15% of pull requests created in commenters' active periods with top-5 recommendation. The activeness based approach recommends appropriate pull requests to commenters, and remove many inappropriate pull requests, which greatly reduces workload of commenters.

6.4. Alternative attributes or techniques

In Section 3, we give some equations to define attributes. These attributes can be defined in some alternative ways. For example, the activeness in Eq. (1) depends on frequency and time of comments left by the commenter. This definition of activeness favors developers who leave comments frequently in recent pull requests. Activeness can be defined in other ways to favor some other developers. We give Eq. (2) to define text similarity. Some further contextual factors may better capture semantic aspects of pull requests. Some other techniques can be used to define text similarity, such as topic model. Time-based file similarity is defined in Eq. (7). It can be defined in some other ways. One alternative method is to use some cluster algorithms, identify pull requests with similar file paths, and then sort their commenters by activeness. In future work, we will try more methods to define attributes, and study their impacts on commenter recommendation.

GitHub provides rich information, and some other attributes can be used in commenter recommendation. Future works may consider following links, which directly reflect relationships between developers [31]. We use titles to measure the text similarity between pull requests. More text information can be used to analyze the text similarity, such as text information in comments. Software projects are not developed in isolation. In software ecosystems, technical dependencies between projects may be useful to find appropriate commenters who participate in relevant projects [32].

In developer recommendation, more techniques may be used to analyze attributes and build approaches. Machine-learning algorithms identify important attributes and make classification. Collaborative filtering can be used to find similar developers or pull requests. In future work, we will try these techniques, and explore whether they can improve the commenter recommendation.

6.5. Implications

We propose approaches based on different attributes to solve the commenter recommendation problem. We experiment on a broad range of datasets to compare the importance of attributes in the commenter recommendation, including activeness, text similarity, file similarity, and social relation. Experimental results show that the activeness is the most important attribute. It provide insights for future researchers to value the activeness of developers. Developers may not always be available, and may even leave projects [14]. Research works about developer recommendation in OSS projects may consider whether developers are active or not.

We further consider composite attributes to recommend commenters, including time-based text similarity, time-based file similarity and time-based social relation. However, results show that the combination of attributes does not improve the precision and recall. The activeness based approach achieves better precision and recall than composite approaches. It may be inapposite to combine the activeness attribute with other attributes. When researchers use the activeness in developer recommendation, they should make experiments and find appropriate ways.

In the pull-based software development, some popular projects receive many pull requests. Commenters may not notice new pull requests in time, and even ignore appropriate pull requests [5,6]. We build the activeness based approach to recommend commenters. The activeness based approach outperforms state-of-the-art approach [5] by a substantial margin. Therefore, we believe that the activeness based approach can be used to remind developers of appropriate pull requests.

Finally, we discuss some issues in practice. The activeness based approach has 3 parameters, including time-decaying parameter λ, temporal window length γ, and the number of recommended commenters k. According to results in the Sections 4.5, time-decaying parameter λ should be set as 1. The Section 4.6 shows that the best precision and recall are always obtained when temporal window length γ is set as 60 days or no time limit. In practice, OSS projects should make experiments, and decide whether temporal window length γ is set as 60 days or no time limit. Tables 5 and 6 shows that as k increases, precision decreases and recall increases. OSS projects should consider project requirements, choose appropriate value of k, and make the balance of precision and recall. Generally speaking, recall is more important than precision in the recommendation. Higher recall means that the approach finds more real commenters who actually give suggestions.

7. Related work

Studies on Code Review. Several previous studies explored review process of code contribution. Nurolahzade et al. discovered that core members were often overwhelmed with many patches they had to review [33]. Rigby et al. observed that if modified codes were not reviewed immediately, they were likely not to be reviewed [34]. Rigby et al. further understood broadcast based peer review in open source software projects [8]. Rigby et al. found that code reviews were expensive, because they needed core reviewers to read, understand and judge code changes. Bosu et al. made empirical studies to evaluate code review process using a popular open source code review tool in OSS communities [35]. Baysal et al. found that nontechnical factors significantly impacted contribution review outcomes [36]. Our work is orthogonal to the above works: we focus on commenter recommendation, which is a different problem than the ones addressed in the above works.

Some works designed methods to recommend code reviewers. Lee et al. proposed a graph based method to automatically recommend suitable reviewers for patches [37]. Balachandran et al. designed a tool called Review Bot to predict developers who modified related code sections frequently as appropriate reviewers [38]. Thongtanunam et al. proposed a method RevFinder to recommend developers who examined files with similar directory paths [25]. Xia et al. designed a hybrid and incremental approach TIE to recommend code-reviewers [39]. These works recommend code reviewers in traditional open source software platforms, while we recommend commenters in pull-based development platform.

Jing et al. proposed CoreDevRec to recommend core members for pull requests in GitHub [10]. Limeira et al. used classification strategies to suggest appropriate core members [9]. As shown in Fig. 1, only core members can make final decisions of pull requests. Any developers can become commenters, and provide suggestions.

Core members and commenters both belong to reviewers, but play different roles in the evaluation. Therefore, core member recommendation approaches [9,10] are different from our commenter recommendation approaches.

Previous works [5,6,40] are mostly related to our work. Yu et al. proposed the CN-based (comment networks) approach to predict appropriate commenters of incoming pull requests in GitHub [5,6,40]. Commenters freely discuss pull requests, and provide suggestions for pull requests' evaluation. Therefore, commenters are called reviewers in previous works [5,6,40]. The reviewer recommendation approach in previous works [5,6,40] is the commenter recommendation approach. Experiments in the Section 4.4 show that the activeness based approach has better precision and recall than the CN-based approach.

Studies on Developer Recommendation. Recommendation systems specific to software engineering (RSSE) assist developers in a wide range of activities. Finding appropriate developers is an important need in recommendation systems specific to software engineering. Many works designed approaches to assign bug reports or change requests [19–21,27,29,41–46]. Hossen et al. considered source code authors, maintainers, and change proneness to triage change requests [27]. Anvik et al. applied a machine learning algorithm and suggested a small number of developers to resolve bug reports [19]. Vasquez et al. utilized source code authorship for assigning expert developers to change requests [29]. Jeong et al. used bug tossing history to assign developers for bug reports [42]. Matter et al. used a text-based method to identify expertise of developers for bug reports [20]. Commenter recommendation is different from above works: Recommending expertise for bugs or new features is to find suitable developers, who write codes and satisfy requirements. Commenter recommendation is a different problem, which finds developers to provide suggestions and leave comments in the code review.

8. Conclusion

In this paper, we build approaches based on different attributes, including activeness, text similarity, file similarity and social relation. We also build composite approaches, which combine the activeness and other attributes together. To investigate the importance of attributes, we perform experiments on 8 projects in GitHub, which include 19,543 pull requests, 206,664 comments and 4817 commenters. The experimental results show that the activeness based approach outperforms approaches based on text similarity, file similarity or social relation by a substantial margin. The activeness is the most important attribute in the commenter recommendation. Furthermore, the activeness based approach always achieves better precision and recall than composite approaches. The activeness based approach outperforms state-of-the-art approach [5] by a substantial margin. Therefore, we believe that the activeness based approach can be used to improve the commenter recommendation and code review process.

Acknowledgment

This work is supported by National Natural Science Foundation of China under Grant No. 61300006, and the State Key Laboratory of Software Development Environment under Grant No. SKLSDE-2015ZX-24, and Beijing Natural Science Foundation under Grant No. 4163074.

References

[1] G. Gousios, M. Pinzger, A. van Deursen, An exploratory study of the pull-based software development model, in: Proceedings of the 36th ICSE, Hyderabad, India, 2014, pp. 345–355.
[2] J. Tsay, L. Dabbish, J. Herbsleb, Influence of social and technical factors for evaluating contribution in github, in: Proceedings of the 36th ICSE, Hyderabad, India, 2014, pp. 356–366.
[3] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D.M. German, D. Damian, The promises and perils of mining github, in: Proceedings of MSR, Hyderabad, India, 2014.
[4] G. Gousios, A. Zaidman, M.-A. Storey, A. van Deursen, Work practices and challenges in pull-based development: the integrators perspective, in: Proceedings of the 37th ICSE, Florence, Italy, 2015, pp. 1–11.
[5] Y. Yu, H. Wang, G. Yin, C.X. Ling, Who should review this pull-request: reviewer recommendation to expedite crowd collaboration, in: Proceedings of the 21st APSEC, Jeju, Korea, 2014, pp. 335–342.
[6] Y. Yu, H. Wang, G. Yin, C. Ling, Reviewer recommender of pull-requests in github, in: Proceedings of the 30th ICSME, Victoria, Canada, 2014, pp. 609–612.
[7] V.J. Hellendoorn, P.T. Devanbu, A. Bacchelli, Will they like this? Evaluating code contributions with language models, in: Proceedings of the 12nd MSR, Florence, Italy, 2015.
[8] P.C. Rigby, M.-A. Storey, Understanding broadcast based peer review on open source software projects, in: Proceedings of the 33rd ICSE, Honolulu, USA, 2011, pp. 541–550.
[9] M.L. de Lima, D. Moreira, A. Plastino, L. Murta, Developers assignment for analyzing pull requests, in: Proceedings of SAC, Salamanca, Spain, 2015, pp. 1567–1572.
[10] J. Jiang, J.-H. He, X.-Y. Chen, Coredevrec: automatic core member recommendation for contribution evaluation, J. Comput. Sci. Technol. 30 (5) (2015) 998–1016.
[11] B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, V. Filkov, Quality and productivity outcomes relating to continuous integration in github, in: Proceedings of FSE, Bergamo, Italy, 2015.
[12] Z. Wang, D.E. Perry, Role distribution and transformation in open source software project teams, in: Proceedings of APSEC, New Delhi, India, 2015, pp. 119–126.
[13] L. Dabbish, C. Stuart, J. Herbsleb, Social coding in github: transparency and collaboration in an open software repository, in: Proceedings of CSCW, Washington, USA, 2012, pp. 1277–1286.
[14] C. Bird, A. Gourley, P. Devanbu, A. Swaminathan, G. Hsu, Open borders? Immigration in open source projects, in: Proceedings of the 4th MSR, Minneapolis,USA, 2007, pp. 1–8.
[15] G. Robles, J.M. Gonzalez-Barahona, Contributor turnover in libre software projects, in: Open Source Systems, 2006, pp. 273–286.
[16] M. Foucault, M. Palyart, X. Blanc, G.C. Murphy, J.-R. Falleri, Developer turnover in open-source software, in: Proceedings of FSE, Bergamo, Italy, 2015.
[17] A. Rastogi, A. Sureka, What community contribution pattern says about stability of software project? in: Proceedings of APSEC, Jeju, Korea, 2014, pp. 31–34.
[18] E. Cohen, M.J. Strauss, Maintaining time-decaying stream aggregates, J. Algorithm 59 (1) (2006) 19–36.
[19] J. Anvik, L. Hiew, G.C. Murphy, Who should fix this bug? in: Proceedings of the 28th ICSE, Shanghai, China, 2006, pp. 361–370.
[20] D. Matter, A. Kuhn, O. Nierstrasz, Assigning bug reports using a vocabulary-based expertise model of developers, in: Proceedings of the 6th MSR, Vancouver, Canada, 2009, pp. 131–140.
[21] X. Xia, D. Lo, X. Wang, B. Zhou, Accurate developer recommendation for bug resolution, in: Proceedings of the 20th WCRE, Koblenz, Germany, 2013, pp. 72–81.
[22] I.S. Dhillon, D.S. Modha, Concept decompositions for large sparse text data using clustering, Mach. Learn. 42 (January,1) (2001) 143–175.
[23] C. Bird, D. Pattison, R. Douza, V. Filkov, P. Devanbu, Latent social structure in open source projects, in: Proceedings of the 16th FSE, Georgia, USA, 2008, pp. 24–35.
[24] Q. Xuan, A. Okano, P.T. Devanbu, V. Filkov, Focus-shifting patterns of oss developers and their congruence with call graphs, in: Proceedings of the 22nd FSE, Hong Kong, China, 2014, pp. 401–412.
[25] P. Thongtanunam, C. Tantithamthavorn, R.G. Kula, N. Yoshida, H. Iida, K. ichi Matsumoto, Who should review my code? A file location-based code-reviewer recommendation approach for modern code review, in: Proceedings of the 22nd SANER, Montreal, Canada, 2015, pp. 141–150.
[26] Z. WeiQiang, N. LiMing, J. He, C. ZhenYu, L. Jia, Developer social networks in software engineering: construction, analysis, and applications, Sci. China Inf. Sci. 57 (12) (2014) 1–23.
[27] M.K. Hossen, H. Kagdi, D. Poshyvanyk, Amalgamating source code authors, maintainers, and change proneness to triage change requests, in: Proceedings of the 22nd ICPC, Hyderabad, India, 2014, pp. 1–12.
[28] P. Sedgwick, Pearson's correlation coefficient, Br. Med. J. 345 (2012) 1–2.
[29] 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 the 28th ICSM, Riva del Garda, Italy, 2012, pp. 451–460.
[30] T. Pangsakulyanont, P. Thongtanunam, D. Port, H. Iida, Assessing mcr discussion usefulness using semantic similarity, in: The 6th International Workshop on Empirical Software Engineering in Practice, Osaka, Japan, 2014, pp. 49–54.
[31] J. Jiang, L. Zhang, L. Li, Understanding project dissemination on a social coding site, in: Proceedings of the 20th WCRE, Koblenz, Germany, 2013, pp. 132–141.
[32] K. Blincoe, F. Harrison, D. Damian, Ecosystems in github and a method for ecosystem identification using reference coupling, in: Proceedings of MSR, Florence, Italy, 2015, pp. 202–207.

[33] M. Nurolahzade, S.R. Seyed Mehdi Nasehi, S.H. Khandkar, The role of patch review in software evolution: an analysis of the mozilla firefox, in: Proceedings of IWPSE-EVOL, Amsterdam, The Netherlands, 2009, pp. 9–17.
[34] P.C. Rigby, D.M. German, M.-A. Storey, Open source software peer review practices: a case study of the apache server, in: Proceedings of the 30th ICSE, Leipzig, Germany, 2008, pp. 541–550.
[35] A. Bosu, J.C. Carver, Peer code review in open source communities using reviewboard, in: Proceedings of the 4th PLATEAU, Arizona, USA, 2012, pp. 17–24.
[36] O. Baysal, O. Kononenko, R. Holmes, M.W. Godfrey, The influence of non-technical factors on code review, in: Proceedings of the 20nd WCRE, Koblenz, Germany, 2013, pp. 122–131.
[37] J.B. Lee, A. Ihara, A. Monden, K. ichi Matsumoto, Patch reviewer recommendation in oss projects, in: Proceedings of the 20th APSEC, Bangkok, Thailand, 2013, pp. 1–6.
[38] V. Balachandran, Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation, in: Proceedings of the 35th ICSE, San Francisco, USA, 2013, pp. 931–940.
[39] X. Xia, D. Lo, X. Wang, X. Yang, Who should review this change? Putting text and file location analyses together for more accurate recommendations, in: Proceedings of ICSME, Bremen, Germany, 2015.
[40] Y. Yu, H. Wang, G. Yin, T. Wang, Reviewer recommendation for pull-requests in github: what can we learn from code review and bug assignment? Inf. Softw. Technol. 74 (2016) 204–218.
[41] H. Kagdi, M. Gethers, D. Poshyvanyk, M. Hammad, Assigning change requests to software developers, J. Softw. 24 (2012) 3–33.
[42] G. Jeong, S. Kim, T. Zimmermann, Improving bug triage with bug tossing graphs, in: Proceedings of the 17th FSE, Amsterdam, The Netherlands, 2009, pp. 111–120.
[43] H. Hu, H. Zhang, J. Xuan, W. Sun, Effective bug triage based on historical bug-fix information, in: Proceedings of the 25th ISSRE, Naples, Italy, 2014, pp. 122–132.
[44] W. Wu, Q.W. Wen Zhang Ye Yang, Drex: developer recommendation with k-nearest-neighbor search and expertise ranking, in: Proceedings of the 18th APSEC, Phan Thiet, Vietnam, 2011, pp. 389–396.
[45] D. Cubranic, G.C. Murphy, Automatic bug triage using text categorization, in: Proceedings the 16th SEKE, 2004, pp. 1–6.
[46] H. Liu, Z. Ma, W. Shao, Z. Niu, Schedule of bad smell detection and resolution: a new way to save effort, IEEE Trans. Softw. Eng. 38 (1) (2012) 220–235.