SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

July 2017, Vol. 60 072105:1–072105:18
doi: 10.1007/s11432-015-0582-8

Developer recommendation on bug commenting: a ranking approach for the developer crowd

Jifeng XUAN[1*], He JIANG[2*], Hongyu ZHANG[3,4] & Zhilei REN[2]

[1]*State Key Laboratory of Software Engineering, School of Computer, Wuhan University, Wuhan* 430072*, China;*
[2]*School of Software, Dalian University of Technology, Dalian* 116621*, China;*
[3]*Microsoft Research Asia, Beijing* 100080*, China;*
[4]*School of Electrical Engineering and Computer Science, The University of Newcastle, Callaghan NSW* 2308*, Australia*

Received April 29, 2016; accepted May 27, 2016; published online April 26, 2017

Abstract A bug tracking system provides a collaborative platform for the developer crowd. After a bug report is submitted, developers can make comments to supplement the details of the bug report. Due to the large number of developers and bug reports, it is hard to determine which developer (also called commenter) is able to comment on a particular bug report. We refer to the problem of recommending developers for commenting on bug reports as commenter recommendation. In this paper, we perform an empirical analysis on commenter recommendation based on five-year bug reports of four open source projects. First, we preliminarily analyze bug comments and commenters in three categories, the relationship between commenters and fixers, the data scale of comments, and the collaboration on bug commenting. Second, we design a recommendation approach via ranking developers in the crowd to reduce the manual effort of identifying commenters. In this approach, we formulize the commenter recommendation problem as a multi-label recommendation task and leverage both developer collaboration and bug content to find out appropriate commenters. Experimental results show that our approach can effectively recommend commenters; 41% to 75% of the recall value is achieved for top-10 recommendation. Our empirical analysis on bug commenting can help developers understand and improve the process of fixing bugs.

Keywords developer recommendation, bug comments, empirical analysis, recommendation for the crowd, collaborative filtering, software repositories

Citation Xuan J F, Jiang H, Zhang H Y, et al. Developer recommendation on bug commenting: a ranking approach for the developer crowd. Sci China Inf Sci, 2017, 60(7): 072105, doi: 10.1007/s11432-015-0582-8

1 Introduction

Fixing bugs is a common and expensive task in software development. In modern software projects, dealing with bugs is an activity of the developer crowd rather than individual ones [1]. To support the collaboration of bug fixing, bug tracking systems are deployed to facilitate bug management. A newly submitted bug report contains a basic description about the failure, but may not provide sufficient information for bug fixing [2, 3]. Therefore, developers who have knowledge about this bug, can make

* Corresponding author (email: jxuan@whu.edu.cn, jianghe@dlut.edu.cn)

comments on the bug report to supplement helpful information, such as how to reproduce, localize, or fix the bug [4–6].

Bug comments enrich the content of bug reports; meanwhile, developers can leverage bug comments to collaborate with each other and share their understandings [7]. The process of commenting on a bug report is an instance of development among the developer crowd, which leverages collective intelligence of developers during the whole life cycle of the bug [8]. In a bug tracking system, such as Bugzilla, each bug report is associated with an email list [9]. Developers in the email list can receive a notification email if the bug-related information is updated; on the other hand, during the process of fixing the bug, developers can manually modify the email list to involve other potential developers, who have expertise in commenting on the bug. However, if the size of the email list is set too large, many developers have to pay effort to read irrelevant bug reports; if the size of the email list is too small, relevant developers would be missed. The manual effort of identifying developers for commenting on bugs should be reduced. This leads to the necessity of recommending developers for commenting on bug reports. We refer to such recommendation as commenter recommendation[1)]. In contrast to assigning one fixer for each bug report in bug triage [4, 10], multiple commenters can add comments to one bug report in commenter recommendation.

In the community of mining software repositories, much work has focused on the bug reports themselves, e.g., bug quality [2], defect prediction [11], and bug triage [4]. Few prior results, however, can be found for bug comments and commenters. Many questions can be raised. For example, what is the relationship between commenters and fixers? How do developers collaborate for bug commenting? Can we leverage bug comments and developer collaboration to recommend developers to add more comments? This paper addresses these questions, which can benefit collaborative development and help developers understand the process of bug commenting.

In this paper, we propose a ranking approach to automatic commenter recommendation for bug reports based on the developer crowd. Our empirical analysis is conducted on five-year bug reports of four open source projects, including Eclipse IDE, JDT, Firefox, and Thunderbird. Our work mainly contributes in two parts, analyzing bug comments with commenters and recommending developers on bug commenting.

First, we preliminarily investigate three problems to analyze the characteristics of comments and commenters, namely the relationship between commenters and fixers, the data scale of comments, and the collaboration on bug commenting. For commenters, we find out that there is an overlap between commenters and fixers; for comments, we show that there exist a large amount of bug comments by commenters; for collaboration on commenting, we examine the collaboration patterns of commenters.

Second, based on our analysis, we address the problem of commenter recommendation, i.e., to recommend expertise developers to comment on bug reports. We formulize such a recommendation problem as a multi-label recommendation task and propose ReComm, a ReCommendation approach on Commenting via ranking for the developer crowd. This approach ranks developers based on recommendation techniques by leveraging both the collaborative patterns and the specific bug content of developers. Experimental results show that 41% to 75% of the recall value is achieved for top-10 recommendation. Our approach can effectively recommend developers for bug commenting, which is helpful to understand and to improve the process of bug resolution.

The major contributions of this paper are as follows:

(1) We propose a ranking approach for the developer crowd, ReComm, to address the problem of commenter recommendation. This approach leverages both collaboration and bug content to rank developers for bug commenting.

(2) We analyze the characteristics of bug commenting and illustrate our recommendation results with five-year bug reports of four open source projects.

The remainder of this paper is organized as follows. Section 2 presents the background and the data collection. Section 3 shows the preliminary study on analyzing comments and commenters. Section 4

1) In this paper, to avoid ambiguity, a developer refers to a contributor in software development, in a board sense. When comparing the roles of bug commenting and fixing, we use the terms "commenter" and "fixer". A commenter denotes a developer who makes comments on a bug report while a fixer denotes a developer who fixes a bug.

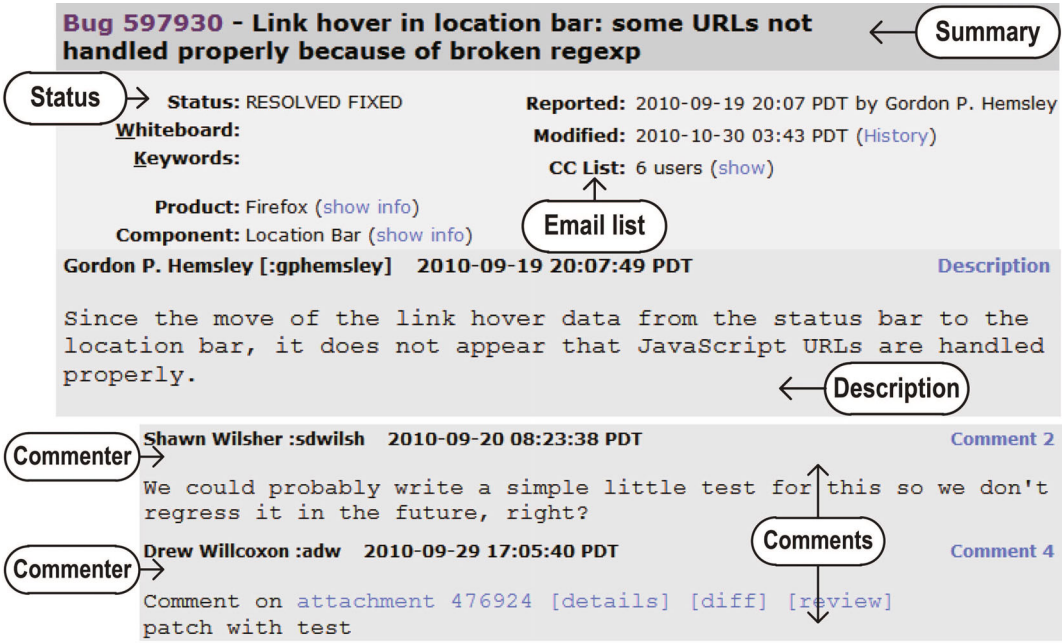

Figure 1 (Color online) A part of bug report for bug 597930 in Firefox and its two comments. This bug describes an error about a regular expression for the location bar.

proposes an approach to commenter recommendation and Section 5 empirically examines its performance. Section 6 shows the threats to validity and Section 7 lists the related work. Section 8 concludes this paper.

2 Background and data sets

2.1 Background

In open source software projects, bug tracking systems are widely deployed to maintain bug reports and to assist developer collaboration. A bug report is a textual form of a software bug, which is stored for reproducing and fixing the bug. In software maintenance, automatic techniques employ bug reports to cope with software tasks. For example, duplicate bug detection (e.g., [12, 13]) searches for similar bugs, which have the same root causes to the given bugs; bug triage (e.g., [4, 10]) aims to assign a bug report to an appropriate developer, who can fix the bug; reopened bug prediction (e.g., [14]) identifies whether bugs will be correctly fixed in the future.

Figure 1 shows a bug report of bug 597930 in Firefox, an open source web browser[2]. Once a bug report is submitted to the bug tracking system, two items, "summary" and "description", are used to indicate the basic content of the bug; another item, "status", records the progress of bug resolving (e.g., a value "fixed" indicates that the bug has been fixed). To facilitate the process of bug fixing, developers make comments to add information to the bug report. For a bug report, developers can manually update an item, email list, to involve other developers who have potential knowledge to add the comments. Figure 1 shows that six developers in the email list and two of comments by two of these developers.

2.2 Data collection

Our experiment is conducted on five-year bug reports of four open source projects, including Eclipse IDE[3], Eclipse Java Development Tools (JDT, for short)[4], Firefox[5], and Thunderbird[6]. For each

2) See the bug report of bug 597930 in Firefox. http://bugzilla.mozilla.org/show_bug.cgi?id=597930.
3) Eclipse Integrated Development Environment. http://wiki.eclipse.org/Platform/.
4) JDT. http://www.eclipse.org/jdt/.
5) Firefox. http://www.mozilla.org/en-US/firefox/.
6) Thunderbird. http://www.mozilla.org/en-US/thunderbird/.

Table 1 Data sets for fixed bugs in four open source projects

Project	Description	#Bugs
Eclipse IDE	An integrated development environment for Java	38189
JDT	A tool suite for Java development	20891
Firefox	A multiple-platform web browser	11575
Thunderbird	An email, news, and chat client	4060

project, we extract all the fixed bug reports from Jan. 1, 2006 to Dec. 31, 2010. We do not use recent bug reports because recent bugs are not stable and updated by developers in daily development [4,15,16]. All the bugs with a status of new, invalid, or will-not-fix are removed. The reason for only selecting fixed bugs is that such fixed bugs are well handled by developers and are more reliable than unfixed bugs [10]. Table 1 lists the details of data sets in the four projects.

For each bug report, we extract its bug ID, commenters, comments, and their associated timestamps. For further experiments (see Section 5), we extract the bug content (summary and description) as well as the bug fields (including reporter, component, priority, and severity). We convert the bug content into a vector of terms based on three steps (namely tokenization, stemming, and stop-word removal [4]). After these three steps, the bug content of each bug report is formed as a vector of terms.

3 Preliminary analysis on bug commenting

In this section, we investigate the characteristics of bug comments and commenters based on an empirical analysis of four open source projects. In a bug tracking system, a commenter makes comments via commenting. This paper explores bug commenting in two categories, namely an empirical analysis of bug comments and commenters (Section 3) and commenter recommendation for bug reports (Sections 4 and 5).

To conduct our preliminary analysis, Subsections 3.1 and 3.2 examine the commenters and comments, respectively; Subsection 3.3 investigates the behavior between commenters and comments (i.e., the process of commenting). These results motivate our approach to developer recommendation on commenting.

3.1 Commenters and fixers: what is the relationship between commenters and fixers?

Commenters add comments to bug reports. In this way, they join the discussion and supplement information for bug fixing. In this section, we examine the contributions of commenters and fixers in open source projects.

A fixer plays an important role in bug fixing. Existing work has investigated the behavior of fixers, such as empirical studies of social networks [2,17] and automatic recommendation techniques [4,10]. One bug report can be commented by many commenters, but can have only one or zero final fixer. In this section, we find out the relationship between commenters and fixers based on their contributions recorded in bug reports.

First, we show the numbers of commenters and fixers in four open source projects in Table 2. We define the similarity between a set of commenters and a set of fixers using a widely-used similarity metric, Jaccard index [18]. Given a set of commenters and a set of fixers, the similarity of these two sets is defined as similarity $= \frac{|\text{Commenters} \cap \text{Fixers}|}{|\text{Commenters} \cup \text{Fixers}|}$, where $|\text{Commenters} \cap \text{Fixers}|$ and $|\text{Commenters} \cup \text{Fixers}|$ are the sizes of the intersection and the union of two given sets, respectively. Among these four projects, Firefox has the largest numbers of commenters and fixers. In Eclipse IDE and JDT, all the fixers also play a role of commenters; 50 out of 837 fixers do not make comments in Firefox and 3 out of 218 fixers do not make comments in Thunderbird. Moreover, Firefox has the highest similarity, i.e., 0.1264, between commenters and fixers among these projects. Such similarity in the other three projects is less than 0.0700. On one hand, there exists an intersection between bug commenters and fixers. This indicates that commenters share similar knowledge with fixers; meanwhile, techniques for recommending fixers, such as bug content [4], can be used for recommending commenters (will be shown in Subsection 4.4).

Table 2 Number of commenters and fixers in four projects

Project	#Commenters	#Fixers	|Commenters ∩ Fixers|	|Commenters ∪ Fixers|	Similarity
Eclipse IDE	5303	319	319	5303	0.0602
JDT	2513	69	69	2513	0.0275
Firefox	6174	837	787	6224	0.1264
Thunderbird	3087	218	215	3090	0.0696

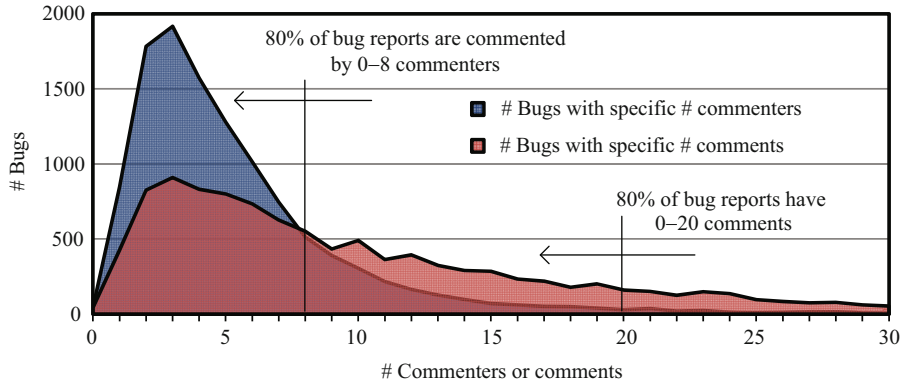

Figure 2 (Color online) Illustration of the number of bugs with a specific number of commenters or comments for Firefox. For example, a point in the illustration with 5 comments (x-axis) and 800 bugs (y-axis) denotes that there exist 800 bugs in Firefox, each of which consists of 5 comments.

Table 3 Comments of bugs in four projects

Project	#Comments in total	#Comments in one bug report			
		Min.	Max.	Average	Std.
Eclipse IDE	236414	0	194	6.19	7.63
JDT	120669	1	121	5.78	6.00
Firefox	172800	0	548	14.93	22.73
Thunderbird	54556	0	312	13.44	18.23

On the other hand, in Subsection 5.4, we will show that the high similarity in Firefox may increase the difficulty to commenter recommendation.

Then, taking the project Firefox as an example, we present the distributions of bugs in Figure 2. Each point in this figure denotes the number of bugs with a specific number of commenters or comments. Meanwhile, we mark the boundaries of 80% of bugs in the projects. As shown in Figure 2, 80% of bugs in Firefox have 0 to 20 comments and 80% of bugs are commented by 0 to 8 commenters. This observation shows the difficulty of commenter recommendation, i.e., one bug report relates to multiple commenters as well as multiple comments.

Summary. In open source projects, the sets of commenters and fixers have an overlap. Meanwhile, commenters contribute different among the four projects in our work. For example, Firefox owns more commenters than Eclipse IDE and commenters in Firefox make more comments than those in Eclipse IDE.

3.2 Comments in bug reports: what is the data scale of comments in open source projects?

A bug tracking system accumulates a large amount of bug comments over time. This question explores the scale of bug comments, as well as the number of comments contributed by each commenter. Large-scale projects accumulate bugs in software development. As shown in Tables 1 and 2, in Firefox, the bug tracking system stores 11575 fixed bugs from 2006 to 2010, which are contributed by 6174 developers. In this section, we show the scale of bug comments for all the four studied projects.

Table 3 lists the number of comments in four open source projects. We present the minimum, maximum, average, and standard deviation for the number of bugs in each project. Among the four projects, Eclipse

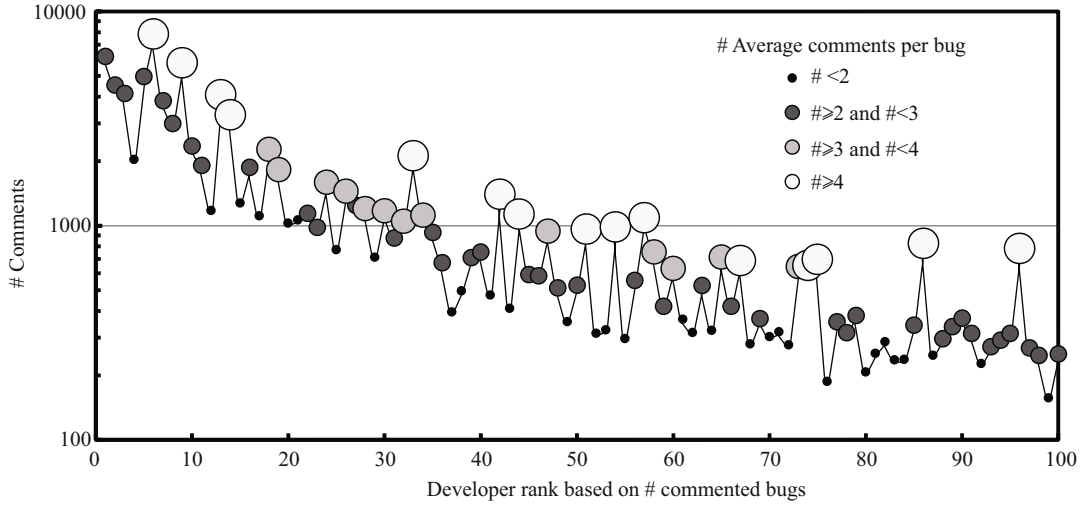

Figure 3 Illustration of comments for top 100 developers in five-year bugs in Firefox. Diameter of a circle denotes the average number of comments per bug for each developer. For example, the largest circles denote developers who contribute to over four comments per bug. Note that the vertical axis is on a log scale. Top 100 developers refer to the 100 developers who make comments on the largest numbers of bug reports.

IDE has the largest number of comments in total while Firefox has the largest average number of comments in one bug report. Firefox also has the highest standard deviation for the number of bugs. That is, the variation from the average in Firefox is the highest. We also manually check the bug reports, which receive the largest number of comments. For example, in Eclipse IDE, bug 67384 has 194 comments, which are contributed by 75 commenters during the discussion over 28 months. As shown in Table 3, the deviation among bug reports is large. That means, it is hard to know how many developers work on one bug; this also leads to our major evaluation measure of commenter recommendation, i.e., Recall@k, which will be shown in Subsection 5.1.

In Figure 3, we take Firefox as an example to illustrate the number of comments, which are made by top 100 developers. Among the top 100 developers, over 30 developers have made over 1000 comments and over 15 developers have contributed to more than 4 comments for each relevant bug. The top 10 developers make 44635 comments (25.83% of total 172800 comments in Table 3) while the following ten developers (from top 11 to top 20) make 19839 comments (11.48% of total comments). A large number of bug comments can facilitate the bug fixing by developers; on the other hand, such large-scale data will add workload to developers. Figure 3 further shows that the number of comments per bug is hard to be predicted.

Summary. Open source projects consist of a large number of comments on bug reports. The deviation of the number of comments per bug is high. The deviation and the large number of comments add the complexity to commenter recommendation.

3.3 Collaboration for commenting: how do commenters collaborate based on comments?

One bug report can be commented by multiple commenters. Thus, commenters naturally collaborate during the process of commenting. We investigate the collaboration patterns among commenters. Bug commenting is a collaborative activity. When commenting on a bug report, developers cooperate with each other to supplement information. Such cooperation is similar to a pairwise based clustering process [19].

In this paper, we regard commenting on the same bug as developer collaboration. To examine the collaboration in bug reports, we use a collaboration pattern to denote a set of developers, who made comments on over 10 same bugs; we use pattern size to denote the size of the set of developers in a collaboration pattern. Table 4 shows the number of collaboration patterns in each size in four projects. Note that collaboration patterns are exclusive. For example, if three commenters A, B & C form a size-3

Table 4 Collaboration pattern size in four projects

Project	Pattern size					
	2	3	4	5	6	7
Eclipse IDE	999	903	80	18	4	0
JDT	249	291	157	26	0	0
Firefox	1382	2321	1460	331	25	3
Thunderbird	267	390	261	61	8	0

pattern, we do not count the size-2 pattern of A & B (or B & C, or C & A) again. As shown in Table 4, in Firefox, 2321 patterns exist in the collaboration among 3 developers while 1460 patterns exist among 4 developers. Moreover, 3 patterns in Firefox have a size of 7 commenters. As shown in Table 4, there exist many collaboration patterns. This motivates us to apply collaborative filtering to rank developers.

Summary. In open source projects, commenters collaborate with each other in the process of bug commenting. We identify many collaboration patterns in all the four projects. Such collaboration patterns can be leveraged to recommend commenters.

4 Commenter recommendation approach

In this section, we leverage the results in Section 3 to recommend developers for bug commenting. As mentioned in Subsection 2.1, developers can manually invite other developers to make comments on a bug report through an associated email list. This section explores the approach to automatically recommending developers for commenting. Based on the above analysis, we design an automatic recommendation approach for bug commenting.

4.1 Problem formulization

Problem. Given existing bug reports and their comments, commenter recommendation aims at ranking candidate developers for making future comments on bug reports to supplement information for bug processing.

Formally, given a historical set R_{H} of bug reports and a set D of developers, for each $r_{\mathrm{H}} \in R_{\mathrm{H}}$, let $D_{\mathrm{H}} \subseteq D$ be a set of relevant developers who have commented on r_{H}. Given a current (newly submitted) bug report r, let T_r denote an unknown set of developers who will comment on r in the future (called the truth in the recommendation for r).

The goal of commenter recommendation is to recommend a set $M_r \subseteq D$ of developers to match T_r based on each historical bug report r_{H} and its D_{H}. In practice, instead of directly comparing M_r to T_r, we assign a score $f_{d,r}$ for each developer $d \in D$ to form a detailed comparison between M_r and T_r. That is, each d in D is ranked to match the truth T_r based on the score $f_{d,r}$, which is generated by a recommendation approach.

Commenter recommendation is a complex and multi-label recommendation task. In software maintenance, most of existing tasks are formulized as the single-label classification or recommendation. For example, in bug triage [4], each new bug report can be mapped to only one fixer (only one label for each bug report). In contrast to single-label tasks, commenter recommendation is formed as a multi-label recommendation task. That is, in this task, each bug report is commented by multiple developers, i.e., multiple labels.

Application scenario. As mentioned in Subsection 2.1, during the whole life cycle of a bug report, developers can make comments to supplement information. For a new bug report r (which needs comments), we recommend commenters, i.e., M_r, based on historical bug reports in R_{H}. In this paper, such recommendation approach is designed via learning from developer collaboration and capability on existing bug reports.

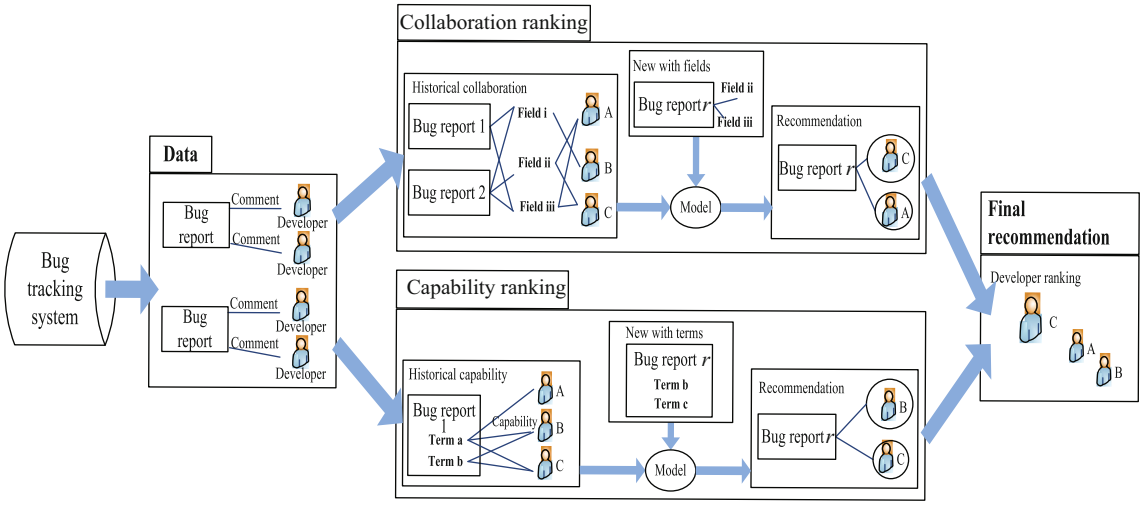

Figure 4 (Color online) Illustration of commenter recommendation via ranking for the developer crowd. In collaboration ranking, developers are ranked based on historical collaboration; in capability ranking, developers are ranked based on specific capability on bug reports. The final recommendation is formed by integrating the above two developer rankings.

4.2 Framework of our approach

In this paper, to solve the problem of commenter recommendation, we propose ReComm, a ReCommendation approach on Commenting via ranking for the developer crowd, in particular developer collaboration and capability. Figure 4 illustrates the framework of ReComm, which integrates recommendation techniques based on two ranking methods.

As shown in Figure 4, our approach ranks developers for given bug reports based on historical bug reports and comments. This approach consists of four phases. First, we extract bug reports and their comments from bug tracking systems; second, collaboration ranking leverages developer collaboration since developers collaboratively make comments on the same bugs; third, capability ranking leverages bug content to measure the capability of developers on specific bugs; fourth, developer rankings obtained by above two phases are combined into a final developer ranking.

4.3 Collaboration ranking

Two developers comment on the same bug report since they share common interest in handling bugs. As shown in Table 4, such developer collaboration widely occurs in bug commenting. Thus, we recommend collaborative developers to a bug if one developer in the collaboration makes a comment on the bug report.

Collaborative filtering is a typical recommendation technique based on the analysis of user-item behavior [20, 21]. As its name suggests, collaborative filtering explores historical data to find the collaboration among users, which is used to recommend users or items in the future. An example of applying collaborative filtering is the book recommendation in online bookstores.

In this paper, we employ collaborative filtering to identify developer collaboration in bug commenting. However, for a new bug report, no developer has made comments. Thus, we cannot identify which collaboration is helpful. In our approach, instead of knowing existing developers, we use bug fields to bridge the historical collaboration and the new bug report. A bug field refers to a specific item in a bug report. We extract four bug fields: reporter, component, priority, and severity. Each bug field has several values. For example, a reporter could be any developer who submits a bug report while a severity could be several pre-defined levels (e.g., in Eclipse IDE, a severity consists of six pre-defined values). Figure 5 illustrates our bug field based collaboration ranking method.

In our work, collaboration ranking consists of two main steps. In modeling, we explore historical bug comments and model developer collaboration on bug fields based on Pearson correlation coefficient. In recommendation, if a new bug report contains a specific bug field, developers, who have collaborated on

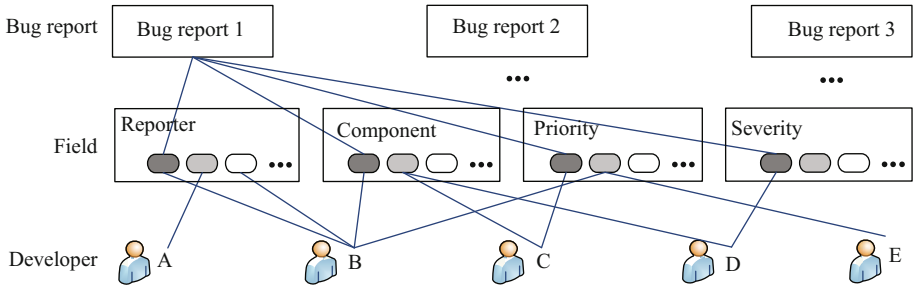

Figure 5 (Color online) Example of bug field based collaborative filtering for developer ranking.

the field, are ranked to add comments for the same bug report. In collaboration ranking, no bug content (such as summary or description) is needed.

Modeling. A set E denotes the set of all bug field values, e.g., a value $e \in E$ could be a severity value of a bug report r. For all the historical bug reports, let a rate $z_{d,e}$ be the number of bug reports on a field value $e \in E$ commented by a developer d. Especially, $z_{d,e} = 0$ indicates no comments by d on e.

For historical bug reports, based on the rates of each developer on each bug field value, we define the similarity between two developers as Pearson correlation coefficient, which is a typical method to measure the linear dependency between two variables in collaborative filtering [22]. In our work, Pearson correlation coefficient of two developers is defined as the covariance of their rates divided by the product of the standard deviations. Given a set of historical bug reports, the similarity between two developers a and b is defined as

$$\mathrm{sim}(a,b) = \frac{\sum_{e\in E}(z_{a,e} - \overline{z_a})(z_{b,e} - \overline{z_b})}{\sqrt{\sum_{e\in E}(z_{a,e} - \overline{z_a})}\sqrt{\sum_{e\in E}(z_{b,e} - \overline{z_b})}}, \tag{1}$$

where $\overline{z_a}$ and $\overline{z_b}$ denote the average rates by a and b over historical bug field values in E. The value of $\mathrm{sim}(a,b)$ is between $+1$ and -1 (inclusive). A value $+1$ denotes a perfect positive correlation for the rates by a and b while -1 denotes a perfect negative correlation. Intuitively, a positive similarity indicates that two developers tend to make similar numbers of comments on the same bugs; a negative similarity indicates that two developers tend to make different numbers of comments.

Recommendation. Let V be a set of θ most similar developers to $d \in D$ and θ is an input parameter. For each $v \in V$, we define the predicted rate of a developer d as

$$s_{d,r} = \overline{z_d} + \frac{\sum_{v\in V}\mathrm{sim}(d,v) \times \sum_{e\in E_r}(z_{v,e} - \overline{z_v})}{\sum_{v\in V}|\mathrm{sim}(d,v)|}, \tag{2}$$

where E_r is a set of all the bug field values of r. Thus, given a bug report and a set of its bug field values, the predicted rates of developers on this bug report can be calculated. A candidate developer with a higher predicted rate is more possible to comment on this bug report.

4.4 Capability ranking

Handling a bug report needs a set of specific skills [10]. A developer is competent to handle specific bugs if he/she has historically handled similar bugs. Existing work in bug triage has proposed content-based capability [10] or expertise [23] to model how competent a developer is. In contrast to training classifiers in machine-learning based approaches, capability or expertise based approaches directly compare developers based on historical data. In this paper, we follow the work by Tamrawi et al. [10], which uses capability to denote the degree of a capable developer with respect to specific bugs.

Capability ranking is based on the fact that developers are capable to fix bugs according to their expertise. We measure the capability between a developer and a bug report to determine whether this developer is able to comment on the bug. In our work, we employ capability ranking to rank developers by measuring the capability rather than train classifiers in machine learning.

Similar to collaborative filtering, capability ranking also consists of two main steps. In modeling, the content of each bug report is viewed as a set of terms. The capability between each developer and each

Table 5 Measurement for the capability

Capability, $c_{d,t}$	Capability, $c_{d,r}$
Jaccard index, $c_{d,t} = \frac{n_{d,t}}{n_d+n_t-n_{d,t}}$	Sum representation, $c_{d,r} = \sum_{t\in r} c_{d,t}$
Ochiai coefficient, $c_{d,t} = \frac{n_{d,t}}{\sqrt{n_d n_t}}$	Product representation, $c_{d,r} = 1 - \prod_{t\in r}(1 - c_{d,t})$
Dice coefficient, $c_{d,t} = \frac{2n_{d,t}}{n_d+n_t}$	

term is generated based on historical bug reports. A "term" in our approach is a word after preprocessing (see Subsection 5.1). In recommendation, given a new bug report, we calculate the capability between each developer and this bug report by counting the capability of each term in the bug report. Then we rank developers based on their capabilities on the bug report.

Modeling. In our work, we extract the summary and the description in bug reports as the bug content. The capability of developers can be defined by different methods. For example, Tamrawi et al. [10] show that Jaccard index and a product representation (combined as a fuzzy set approach) are effective to measure the capabilities of developers. In this paper, besides Jaccard index in [10], we employ two other methods, i.e., Ochiai coefficient and Dice coefficient, to measure the capability between a developer and a term. Ochiai coefficient and Dice coefficient are two measurements of correlation in information retrieval, which were first introduced in biology [24] and ecology [25], respectively. Given a developer d and a term t in historical bug reports, we use n_d, n_t, and $n_{d,t}$ to denote the number of bug reports that d has commented on, the number of bug reports that contain t, and the number of bug reports for both. Table 5 lists the three methods of measuring capability $c_{d,t}$ between a developer d and a term t.

Recommendation. Besides the product representation in [10], we employ another method (called a sum representation in our work) to measure the capability between a developer and a bug report. Given a bug report r and a developer $d \in D$, we obtain each term t in bug content of r and calculate the capability $c_{d,r}$ based on $c_{d,t}$. The two methods of calculating $c_{d,r}$ can be found in Table 5. Then, developers with high capabilities to the given bug report are recommended to make comments.

The measurements of the capability in Table 5 can form six combinations. In Subsection 5.2, we will compare these six combinations for capability ranking and show that Ochiai coefficient with the sum representation is the most effective one.

4.5 Approach integration

In Subsections 4.3 and 4.4, we design a collaboration ranking approach and a capability ranking approach by leveraging developer collaboration and bug content, respectively. For a given bug report r, either approach can assign one score (a predicted rate $s_{d,r}$ or a capability $c_{d,r}$) to each candidate developer $d \in D$.

In real-world bug tracking systems, both developer collaboration and developer capability are helpful to identify relevant developers. Thus, we consider integrating collaboration ranking and capability ranking. First, we respectively normalize the predicted rate $s_{d,r}$ and the capability $c_{d,r}$ to the range 0 to 1. Then, we combine the values of $s_{d,r}$ and $c_{d,r}$ to generate a final score $f_{d,r}$. The final score is defined as $f_{d,r} = (1-\omega)s_{d,r}+\omega c_{d,r}$, where $0 \leqslant \omega \leqslant 1$. Actually, ω is a weight factor to determine the contribution of $s_{d,r}$ and $c_{d,r}$ in the final score. Thus, $f_{d,r}$ is a weighted average of $s_{d,r}$ and $c_{d,r}$. This integrated approach takes advantages of both collaboration and capability ranking. Based on $f_{d,r}$, we rank all the developers in D and recommend them to make comments on r.

5 Experiments and results

We examine the results of our approach based on data sets in four open source projects. Our approach ReComm is implemented in Java 1.7.

Table 6 Average scale of data sets in ten rounds

Average scale of 10 rounds	Average scale in training sets			Average scale in test sets		
	#Bug	#Commenter	#Comment	#Bug	#Commenter	#Comment
Eclipse IDE	5198.3	224.4	40484.9	914.2	145.8	7567.1
JDT	2513.0	73.7	18076.3	462.4	50.1	3596.2
Firefox	3767.3	337.9	52430.7	676.3	234.8	10721.4
Thunderbird	1019.4	122.9	12860.8	216.8	81.9	2848.3

5.1 Data preparation and evaluation measures

To evaluate the performance of commenter recommendation, we employ a 10-round incremental framework for experiments [26]. First, we chronologically sort all the bug reports in a project; then, these bug reports are cut into 11 equally-sized folds. We form 10 rounds evaluation with these 11 folds. In the ith round, the first i folds are formed as a training set and the $(i+1)$th fold is used as a test set[7]. We run one approach on each round and calculate the average results of all the 10 rounds. In each round, inactive developers who comment on less than five bug reports are removed since such inactive developers make little contribution to bug handling. Table 6 shows the average scale of 10 rounds for training sets and test sets.

As shown in Subsection 4.1, the commenter recommendation is a multi-label recommendation problem. This leads to a complicated evaluation. In our work, we employ Recall@k [6] as evaluation measures, where k equals to 5 or 10. The value range for each evaluation measure is from 0 to 1. A value equals to 1 denotes that a recommendation approach generates perfectly matched results to the truth in the test set.

Recall@k aims to measure how many of actual developers (that make comments) are hit by the top k recommended developers [6,20]. Meanwhile, another evaluation measure, Precision@k is also considered; Precision@k aims to measure how many of the top k recommended developers are correct. Given a test set R of m bug reports, let T_r be the set of developers in truth for a bug report $r \in R$. Recall@k and Precision@k are defined as

$$\text{Recall@}k = \frac{1}{|R|}\sum_{r\in R}\frac{|P_r \cap T_r|}{|T_r|}, \tag{3}$$

$$\text{Precision@}k = \frac{1}{|R|}\sum_{r\in R}\frac{|P_r \cap T_r|}{k}\ , \tag{4}$$

where P_r denotes a set of top-k developers based on a recommendation approach. Note that in Precision@k, the k value in the denominator is the same for all bug reports. Since in commenter recommendation, the number of commenters that are involved in each bug is different (Table 3), it is better to use Recall@k to consider the variable number of commenters. Hence, in this paper, we employ Recall@k as the major evaluation measure and list Precision@k as reference. Based on Recall@k, all the top k developers by recommendation are compared with developers in truth.

5.2 Performance in capability ranking

In Subsection 4.4, we propose three methods to measure the capability between a developer and a term, namely Jaccard index, Ochiai coefficient, and Dice coefficient. We also propose two methods to count the capability between a developer and a bug report, namely a sum representation and a product representation. These methods (see Table 5) can be formed into six combinations. In this section, we take Firefox as an example to examine the results of these six combinations of measurement methods.

Figure 6 shows the results of two evaluation measures, Recall@5 and Recall@10, on the data set of Firefox. We can find out that either Recall@5 or Recall@10 does not change much. The changes of Recall@5 and Recall@10 are less than 2.0%. Based on Figure 6, Ochiai-Sum achieves the best results on Recall@5 while Dice-Sum and Jaccard-Product achieve the best results on Recall@10.

7) We use training sets and test sets to simplify the statement, although our approach ReComm does not train classifiers.

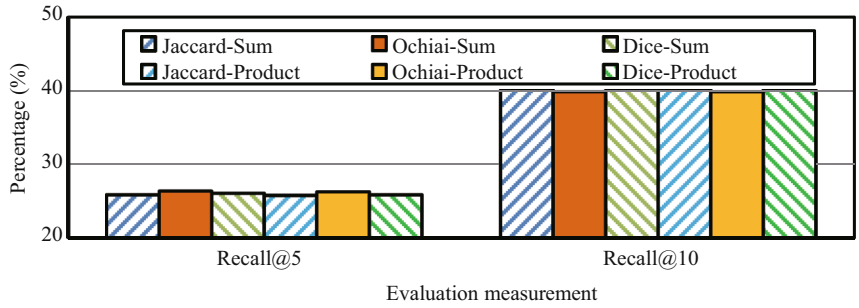

Figure 6 (Color online) Measurement comparison in capability ranking on Firefox based on Recall@5 and Recall@10. Each result is an average of results in ten rounds.

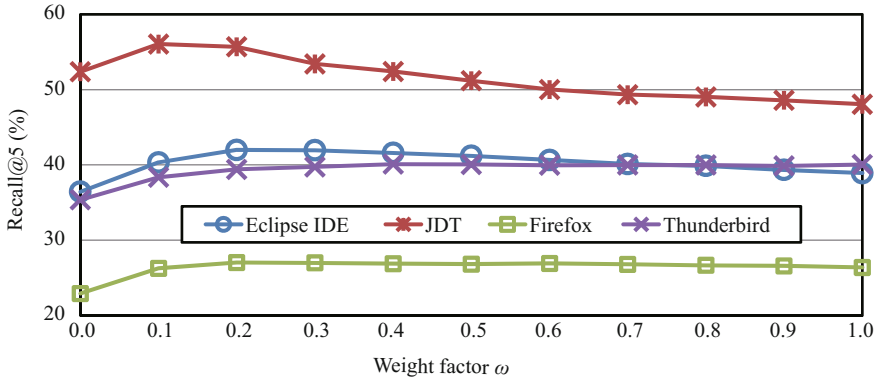

Figure 7 (Color online) Performance of Recall@5 in the integrated approach when changing the weight factor on four projects. Each result is an average of results in ten rounds.

5.3 Performance of the integrated approach

In this paper, our approach to commenter recommendation is the integration of collaboration and capability ranking. In Subsection 4.5, we employ a weight factor ω to combine two ranking approaches.

We investigate the performance of the integrated approach based on the value of the weight factor, ω. Figure 7 presents Recall@5 on four projects. This figure shows that most of results with ω from 0.1 to 0.4 are better than those with ω equals to 0 or 1. In other words, the integration of two filtering approaches can obtain good results by leveraging both developer collaboration and bug content. When ω changes from 0 to 1, Recall@5 on four projects varies around 10%. This result show that the integrated approach is sensitive to the value of the weight factor ω .

As shown in Figure 7, to obtain the best results of Recall@5, the values of ω for Eclipse IDE, JDT, Firefox, and Thunderbird are 0.1, 0.2, 0.4, and 0.3, respectively. The average of these four values is 0.25. In the following experiments, we identically set $\omega = 0.3$ for all the projects to avoid changing ω among projects.

5.4 Comparison with existing approaches

We compare our commenter recommendation approach ReComm on the data sets of four projects with four existing approaches: Drex-Frequency, Drex-Outdegree, Bugzie, and Dretom.

Drex, Developer Recommendation with k-nearest-neighbor search and EXpertise ranking, by Wu et al. [27] is originally proposed for commenter recommendation. In Drex, bug reports are converted into term vectors via a vector space model based on tf-idf weighting; then a k-nearest-neighbor classifier is trained to detect similar bugs. Among different criteria, Drex obtains the best performance if related developers are ranked in frequency or out-degree (called Drex-Frequency and Drex-Outdegree for short) in a developer social network [27].

Table 7 Recall@5 and Recall@10 of five approaches on four projects

Project	Recall@5								
	ReComm	Drex-Frequency		Drex-Outdegree		Bugzie		Dretom	
	Value	Value	Impr (%)	Value	Impr (%)	Value	Impr (%)	Value	Impr (%)
Eclipse IDE	41.95	37.36	**12.28**	36.24	**15.74**	17.08	**145.65**	14.70	**185.39**
JDT	53.41	52.63	**1.48**	50.92	**4.89**	35.01	**52.55**	41.48	**28.77**
Firefox	26.99	22.95	**17.61**	22.03	**22.50**	14.87	**81.48**	14.89	**81.21**
Thunderbird	39.74	32.48	**22.37**	32.18	**23.51**	23.77	**67.21**	30.79	**29.06**
Project	Recall@10								
	ReComm	Drex-Frequency		Drex-Outdegree		Bugzie		Dretom	
	Value	Value	Impr (%)	Value	Impr (%)	Value	Impr (%)	Value	Impr (%)
Eclipse IDE	56.41	47.98	**17.58**	47.75	**18.14**	30.39	**85.66**	24.45	**130.69**
JDT	74.53	72.72	**2.49**	72.36	**3.01**	60.82	**22.54**	62.64	**19.00**
Firefox	40.63	35.56	**14.25**	35.95	**13.00**	25.02	**62.39**	24.99	**62.57**
Thunderbird	57.75	50.68	**13.96**	51.93	**11.21**	39.81	**45.05**	45.23	**27.67**

To our knowledge, besides Drex, no other approaches are directly proposed for commenter recommendation. We compare our approach with two other related approaches. Bugzie, Bug Triage based on Fuzzy Set and Cache, by Tamrawi et al. [10], is originally proposed for bug triage. This approach models developer expertise with the fuzzy set theory and optimizes the approach with a cache based method.

Dretom, Developer REcommendation based on TOpic Models, by Xie et al. [28] is originally proposed for bug resolution. In Dretom, a topic model is trained based on historical bug reports; then developers are associated with topics via bug reports. Given a new bug, Dretom calculates probabilities for all the topics and rank developers based on these topics [28]. According to [10, 27, 28], we retune the parameters on data sets. Note that both Bugzie and Dretom are not originally designed for commenter recommendation. Then the data in commenter recommendation may hurt their performance because of the adaption of them to this new problem. Our experiment will use these algorithms as references, without deeply exploring their computational ability in commenter recommendation.

In our proposed approach, ReComm, we set $\omega = 0.3$ according to the tuning experiment in Subsection 5.3; we use Ochiai coefficient and the sum representation in capability ranking, according to the experiment in Subsection 5.2. For θ, we manually tune the value of θ and find out that collaborative filtering is insensitive to θ. In the following experiments, we set $\theta = 11$ as a default value.

In Table 7, we compare average results in 10 rounds for five approaches on four projects with Recall@5 and Recall@10. Besides directly presenting results, we evaluate the percentage of improvement between ReComm and other approaches. The term "Impr%" between ReComm and an approach X is defined as $\text{Impr}\% = \frac{p_{\text{ReComm}} - p_X}{p_X} \times 100\%$ where p_{ReComm} and p_X denote the recall values of ReComm and X, respectively.

As shown in Table 7, ReComm achieves the best results out of five total approaches. In most of projects, ReComm can improve over 10% of the recall values. One exception is JDT. That is, ReComm on JDT is less than 5% than Drex-Frequency and Drex-Outdegree. The major reason is JDT is not in a large scale, comparing with other projects. According to Table 6, the average number of commenters in training sets and test sets are 73 and 50, respectively. Thus, each of ReComm, Drex-Frequency, and Drex-Outdegree can obtain a good result.

Among the four projects, the recall values in Firefox are worse than the other three projects. For example, Recall@10 by ReComm in Firefox is much lower than that in any other projects. Based on our analysis in Subsections 3.1 and 3.2, we consider two possible reasons. One is Firefox has the highest similarity between the sets of commenters and fixers among the four projects (Table 2); the other is Firefox has the largest number of commenters (Table 2) and the largest average number of comments for one bug report (Table 3). The similarity between commenters and fixers may lead the complex relationship among developers while the large numbers of commenters and comments in Firefox may lead to the difficulty of recommendation.

Table 8 Precision@5 and Precision@10 of five approaches on four projects

Project	Precision@5								
	ReComm	Drex-Frequency		Drex-Outdegree		Bugzie		Dretom	
	Value	Value	Impr (%)	Value	Impr (%)	Value	Impr (%)	Value	Impr (%)
Eclipse IDE	16.91	16.79	**0.70**	15.27	**10.74**	9.80	**72.59**	7.97	**112.22**
JDT	25.78	25.61	**0.69**	24.78	**4.04**	20.52	**25.62**	23.75	**8.57**
Firefox	21.17	20.87	**1.41**	20.21	**4.74**	14.44	**46.63**	14.18	**49.23**
Thunderbird	24.11	22.48	**7.21**	24.16	**−0.22**	18.41	**30.94**	22.49	**7.20**
Project	Precision@10								
	ReComm	Drex-Frequency		Drex-Outdegree		Bugzie		Dretom	
	Value	Value	Impr (%)	Value	Impr (%)	Value	Impr (%)	Value	Impr (%)
Eclipse IDE	11.82	10.23	**15.54**	10.17	**16.22**	8.69	**35.99**	6.68	**76.82**
JDT	17.94	18.03	**−0.52**	17.94	**−0.01**	17.78	**0.87**	18.09	**−0.83**
Firefox	16.34	15.16	**7.83**	15.00	**8.92**	12.24	**33.50**	11.97	**36.48**
Thunderbird	18.66	16.43	**13.63**	17.05	**9.49**	15.80	**18.09**	17.06	**9.43**

In Table 8, we show the comparison of Precision@k values on four projects. ReComm can achieve better precision values in most of comparisons. Note that for one project, the value of Precision@5 should be no less than the value of Precision@10 because the definition of Precision@k is based on the total number k of recommended developers. However, the improvement on precision values is lower than that on recall values. In the project Thunderbird, ReComm on Precision@5 is worse than Drex-Outdegree by 0.22%; in JDT, ReComm on Precision@10 is worse than Drex-Frequency, Drex-Outdegree, and Dretom by 0.52%, 0.01%, and 0.83%, respectively. One possible reason is that Precision@k is not designed to compare the number of recommended developers and the number of actual developers that make bug comments; the difference between the number of recommended developers and the number of actual developers may lead to the current result. For example, for one bug that is commented by 8 developers, a Precison@10 value is not suitable since the maximum of correctly recommended developer is 8 (a precision value of 0.8, less than 1). We leave this as one threat to our work.

ReComm can obtain better results on most of projects than other approaches under evaluation. This derives from the design of combining ranking based on developer collaboration and bug content. With developer collaboration, commenters who have similar behaviors of commenting will be identified; with bug content, developers who have similar expertise will be identified. Then developers based on both collaboration and bug content can be selected as candidates in the recommendation. These selected developers are ranked according to their combined scores as the final solution. Hence, the combination of these rankings will benefit future commenter recommendation.

Summary. Experimental results show that our approach ReComm can effectively recommend developers for commenting on bugs. This approach integrates collaborative filtering and capability ranking to rank developers by leveraging both developer collaboration and bug content. ReComm can obtain better results than the other four approaches on most of projects. In three out of four projects, the Recall@10 value of ReComm is over 50%.

6 Threats to validity

In this paper, we propose an recommendation approach on bug commenting. We list three potential threats to the validity in our work as follows.

For the analysis of bug comments and commenters, we pay much effort to explore the numerical characteristics based on the data in bug tracking systems. We design a series of experiments to analyze bug commenters and comments. However, further investigation should be conducted. For example, what are the differences between the comments made by a bug fixer and the comments made by an ordinary commenter? How does source code change after new comments [29]? This question should be empirically answered. It is helpful to manually examine the content of comments made by the bug fixers. We will

investigate such issues in our future work.

In collaboration ranking, we recommend a developer to a bug, if his/her collaborative developers make comments on the same bug. However, this approach is limited by the number of known developers, who have already collaboratively made comments on historical bugs. Such limitation is a typical problem in collaborative filtering, called data sparsity [22], e.g., the sparsity of historical comments by developers in our work. Another typical problem in recommendation systems is cold start [21], which is caused by insufficient known information. To weaken the impact of data sparsity and cold start in commenter recommendation, it is helpful to design a new recommendation algorithm to augment the results by collaborative filtering.

In experiment, we follow existing work (e.g., [6]) to leverage Recall@k as the major evaluation measure; then we use Precision@k as reference. The major reason for this choice is that the deviation of the number of comments is high (as shown in Table 3); Recall@k can measure the correctly recommended developers based on the different number of actual commented developers. Hence, Recall@k is chosen as the major measure, rather than Precision@k. However, we have not provide any quantified evidence for this choice. A further discussion about the measures for multi-label recommendation would be helpful to this issue.

7 Related work

7.1 Bug comments

Bug report summarization selects representative sentences in the bug description and the comments to support the understanding of bug reports. Rastkar et al. [30] address bug report summarization by recognizing the similarity between bug commenting and email threads. They collect 24 features to generate an extractive summary for each bug report. By estimating the attention of developers, Lotufo et al. [7] model the reading process of bug comments to create summaries.

Besides bug report summarization, bug comments are employed to analyze the developer behavior in software development. Hong et al. [17] build a social network with bug comments to discover sub-communities of developers in open source projects. In the above work, bug comments provide a platform to support the communication among developers.

In this paper, we propose an integrated approach to solve the problem of commenter recommendation. To our knowledge, three pieces of existing work have focused on this problem with content-based recommendation, i.e., Drex by Wu et al. [27], Dretom by Xie et al. [28], and recent work of DevRec by Xia et al. [31]. The differences between our work and existing work ([27, 28, 31]) are as follows. First, the approach in our work combines collaborative filtering with capability ranking to leverage both developer collaboration and bug content while existing work focuses on bug content. We believe the collaboration among developers is important to identify relevant developers. Second, we conduct experiments on five-year bug reports in four open source projects. Experimental results show that our work achieves much better results than exiting work.

7.2 Recommendation for the crowd

A widely-used approach to modeling the collaboration is social network analysis. Bird et al. [32] discover the latent structure among developer collaboration in an email list database and find out that developer communities arise spontaneously Pinzger et al. [11] build a developer-module network to model the collaboration and predict module failures with social network metrics. Meneely & Williams [5] empirically investigate the reliability of such metrics for software failures. To face with bug data, Hu et al. [15] build an predictive model for predicting developers who can fix bugs; Xuan et al. [33] investigate the collaboration on bug reports and transfer the collaboration on bug data to requirements data, which are used to solve requirements optimization with heuristics [34]. Recent work by Mao et al. [35] and Wang et al. [36] deeply investigate the collaboration in the crowd and developer recommendation.

In contrast to the above work, in this paper, we empirically analyze the developer collaboration based on bug commenting and model such collaboration with collaborative filtering. To our knowledge, existing work by Park et al. [37] has employed collaborative filtering on bug reports to optimize the time cost of fixing bugs. In our work, we use collaborative filtering on comments to predict the behavior of commenting.

Capability or expertise is a kind of measurement to quantify the specific knowledge of a developer in software development. Anvik and Murphy [9] explored the implementation expertise in check-in logs and bug reports. Matter et al. [23] used vocabulary-based expertise to assign bug reports to relevant developers. Focusing on the same problem, Tamrawi et al. [10] proposed the Jaccard-index based method to model the capability of developers. To distinguish experts and non-experts, Huo et al. [38] conducted an empirical study of the effects of expert knowledge on developer recommendation and feature location. For further understanding developer capability, Meyer et al. [39] designed a questionnaire to analyze the productivity of developers while Fritz et al. [40] used a psycho-physiological tracking system to measure the task difficulty for developers. For characterizing bug reports, Shihab et al. [41] used developer capability to predict re-opened bugs; Xia et al. [42] made a further improvement on re-opened bug prediction.

In our work, we follow [10] to model the capability but propose Ochiai coefficient instead. Experiments show that Ochiai coefficient can achieve better results. To our knowledge, this is one of the first work that applies Ochiai coefficient to model the capability on bug reports.

8 Conclusion

In this paper, we describe a ranking approach on bug commenting using five-year bug reports of four open source projects. We investigate the characteristics of comments, commenters, and the recommendation of commenters for bug reports. The analysis results in our work can help developers better understand the process of bug commenting. Meanwhile, our approach to commenter recommendation can effectively recommend multiple developers for commenting on bugs. Automatic commenter recommendation can be integrated into open source projects to facilitate collaborative development.

In future work, we plan to design a new mechanism to further model the developer collaboration in bug commenting. For example, we consider that historical sequences of comments are helpful to indicate comments in the future. Another future work is to explore the changes of developer collaboration on bug commenting and to further understand the factors of such changes.

Acknowledgements This work was supported by National Natural Science Foundation of China (Grant Nos. 61502345, 61403057, 61370144, 61272089) and New Century Excellent Talents in University (Grant No. NCET-13-0073).

Conflict of interest The authors declare that they have no conflict of interest.

References

1 Wu W, Tsai W-T, Li W. An evaluation framework for software crowdsourcing. Front Comput Sci, 2013, 7: 694–709
2 Zimmermann T, Premraj R, Bettenburg N, et al. What makes a good bug report? IEEE Trans Softw Eng, 2010, 36: 618–643
3 Xuan J F, Martinez M, DeMarco F, et al. Nopol: automatic repair of conditional statement bugs in java programs. IEEE Trans Softw Eng, 2017, 43: 34–55
4 Anvik J, Hiew L, Murphy G C. Who should fix this bug? In: Proceedings of the 28th International Conference on Software Engineering. New York: ACM, 2006. 361–370
5 Meneely A, Williams L. Socio-technical developer networks: should we trust our measurements? In: Proceedings of the 33rd International Conference on Software Engineering. New York: ACM, 2011. 281–290
6 Zhou J, Zhang H, Lo D. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In: Proceedings of the 34th International Conference on Software Engineering. Washington:

IEEE, 2012. 14–24
7 Lotufo R, Malik Z, Czarnecki K. Modelling the hurried bug report reading process to summarize bug reports. In: Proceedings of the 28th IEEE International Conference on Software Maintenance. Washington: IEEE, 2012. 430–439
8 Gao J, Bai X, Tsai W-T, et al. Mobile application testing: a tutorial. IEEE Comput, 2014, 47: 46–55
9 Anvik J, Murphy G C. Determining implementation expertise from bug reports. In: Proceedings of the 4th International Workshop on Mining Software Repositories. Washington: IEEE, 2007. 2
10 Tamrawi A, Nguyen T T, Al-Kofahi J M, et al. Fuzzy set and cache-based approach for bug triaging. In: Proceedings of the 19th ACM SIGSOFT Symposium on Foundations of Software Engineering and the 13th European Software Engineering Conference. New York: ACM, 2011. 365–375
11 Pinzger M, Nagappan N, Murphy B. Can developer-module networks predict failures? In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering. New York: ACM, 2008. 2–12
12 Wang X, Zhang L, Xie T, et al. An approach to detecting duplicate bug reports using natural language and execution information. In: Proceedings of the 30th International Conference on Software Engineering. New York: ACM, 2008. 461–470
13 Sun C, Lo D, Khoo S C, et al. Towards more accurate retrieval of duplicate bug reports. In: Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering. Washington: IEEE, 2011. 253–262
14 Shihab E, Ihara A, Kamei Y, et al. Predicting re-opened bugs: a case study on the eclipse project. In: Proceedings of the 17th Working Conference on Reverse Engineering. Washington: IEEE, 2010. 249–258
15 Hu H, Zhang H, Xuan J, et al. Effective bug triage based on historical bug-fix information. In: Proceedings of IEEE 25th International Symposium on Software Reliability Engineering. Washington: IEEE, 2014. 122–132
16 Xuan J, Jiang H, Hu Y, et al. Towards effective bug triage with software data reduction techniques. IEEE Trans Knowl Data Eng, 2015, 27: 264–280
17 Hong Q, Kim S, Cheung S C, et al. Understanding a developer social network and its evolution. In: Proceedings of the 27th IEEE International Conference on Software Maintenance. Washington: IEEE, 2011. 323–332
18 Han J, Kamber M. Data Mining: Concepts and Techniques. San Francisco: Morgan Kaufmann, 2001. 71–72
19 Jiang H, Ren Z, Xuan J, et al. Extracting elite pairwise constraints for clustering. Neurocomputing, 2013, 99: 124–133
20 Herlocker J L, Konstan J A, Terveen L G, et al. Evaluating collaborative filtering recommender systems. ACM Trans Inf Syst, 2004, 22: 5–53
21 Sarwar B, Karypis G, Konstan J, et al. Item-based collaborative filtering recommendation algorithms. In: Proceedings of the 10th International Conference on World Wide Web. New York: ACM, 2001. 285–295
22 Adomavicius G, Tuzhilin A. Toward the next generation of recommender systems: a survey of the state-of-the-art and possible extensions. IEEE Trans Knowl Data Eng, 2005, 17: 734–749
23 Matter D, Kuhn A, Nierstrasz O. Assigning bug reports using a vocabulary-based expertise model of developers. In: Proceedings of the 6th IEEE International Working Conference on Mining Software Repositories. Washington: IEEE, 2009. 131–140
24 Ochiai A. Zoogeographic studies on the soleoid fishes found in japan and its neighbouring regions. Bull Jpn Soc Sci Fish, 1957, 22: 526–530
25 Dice L R. Measures of the amount of ecologic association between species. Ecology, 1945, 26: 297–302
26 Bettenburg N, Premraj R, Zimmermann T, et al. Duplicate bug reports considered harmful ... really? In: Proceedings of IEEE International Conference on Software Maintenance. Washington: IEEE, 2008. 337–345
27 Wu W, Zhang W, Yang Y, et al. Drex: developer recommendation with k-nearest-neighbor search and expertise ranking. In: Proceedings of the 18th Asia Pacific Software Engineering Conference. Washington: IEEE, 2011. 389–396
28 Xie X, Zhang W, Yang Y, et al. Dretom: developer recommendation based on topic models for bug resolution. In: Proceedings of the 8th International Conference on Predictive Models in Software Engineering. New York: ACM, 2012. 19–28
29 Xuan J F, Cornu B, Martinez M, et al. B-refactoring: automatic test code refactoring to improve dynamic analysis. Inf Softw Tech, 2016, 76: 65–80
30 Rastkar S, Murphy G C, Murray G. Summarizing software artifacts: a case study of bug reports. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering. New York: ACM, 2010. 505–514
31 Xia X, Lo D, Wang X, et al. Dual analysis for recommending developers to resolve bugs. J Softw Evol Process, 2015, 27: 195–220
32 Bird C, Pattison D, D'Souza R, et al. Latent social structure in open source projects. In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering. New York: ACM, 2008. 24–35
33 Xuan J, Jiang H, Ren Z, et al. Solving the large scale next release problem with a backbone-based multilevel algorithm. IEEE Trans Softw Eng, 2012, 38: 1195–1212

34 Jiang H, Xuan J, Ren Z. Approximate backbone based multilevel algorithm for next release problem. In: Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation. New York: ACM, 2010. 1333–1340

35 Mao K, Yang Y, Wang Q, et al. Developer recommendation for crowdsourced software development tasks. In: Proceedings of IEEE Symposium on Service-Oriented System Engineering. Washington: IEEE, 2015. 347–356

36 Wang S, Zhang W, Wang Q. Fixercache: unsupervised caching active developers for diverse bug triage. In: Proceedings of ACM-IEEE International Symposium on Empirical Software Engineering and Measurement. New York: ACM, 2014. 1–10

37 Park J-W, Lee M-W, Kim J, et al. Costriage: a cost-aware triage algorithm for bug reporting systems. In: Proceedings of the 25th AAAI Conference on Artificial Intelligence. Palo Alto: AAAI Press, 2011. 139–144

38 Huo D, Ding T, McMillan C, et al. An empirical study of the effects of expert knowledge on bug reports. In: Proceedings of IEEE International Conference on Software Maintenance and Evolution. Washington: IEEE, 2014. 1–10

39 Meyer A N, Fritz T, Murphy G C, et al. Software developers' perceptions of productivity. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. New York: ACM, 2014. 19–29

40 Fritz T, Begel A, Müller S C, et al. Using psycho-physiological measures to assess task difficulty in software development. In: Proceedings of the 36th International Conference on Software Engineering. New York: ACM, 2014. 402–413

41 Shihab E, Ihara A, Kamei Y, et al. Studying re-opened bugs in open source software. Empir Softw Eng, 2013, 18: 1005–1042

42 Xia X, Lo D, Shihab E, et al. Automatic, high accuracy prediction of reopened bugs. Autom Softw Eng, 2015, 22: 75–109