Journal of Software: Evolution and Process

JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS
J. Softw. Evol. and Proc. 2015; **27**:195–220
Published online 3 March 2015 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/smr.1706

Dual analysis for recommending developers to resolve bugs

Xin Xia[1], David Lo[2], Xinyu Wang[1,*,†] and Bo Zhou[1]

[1]*College of Computer Science and Technology, Zhejiang University, Hangzhou, China*
[2]*School of Information Systems, Singapore Management University, Singapore, Singapore*

ABSTRACT

Bug resolution refers to the activity that developers perform to diagnose, fix, test, and document bugs during software development and maintenance. Given a bug report, we would like to recommend the set of bug resolvers that could potentially contribute their knowledge to fix it. We refer to this problem as *developer recommendation for bug resolution*. In this paper, we propose a new and accurate method named *DevRec* for the developer recommendation problem. DevRec is a composite method that performs two kinds of analysis: bug reports based analysis (BR-Based analysis) and developer based analysis (D-Based analysis). We evaluate our solution on five large bug report datasets including GNU Compiler Collection, OpenOffice, Mozilla, Netbeans, and Eclipse containing a total of 107,875 bug reports. We show that DevRec could achieve recall@5 and recall@10 scores of 0.4826–0.7989, and 0.6063–0.8924, respectively. The results show that DevRec on average improves recall@5 and recall@10 scores of Bugzie by 57.55% and 39.39%, outperforms DREX by 165.38% and 89.36%, and outperforms NonTraining by 212.39% and 168.01%, respectively. Moreover, we evaluate the stableness of DevRec with different parameters, and the results show that the performance of DevRec is stable for a wide range of parameters. Copyright © 2015 John Wiley & Sons, Ltd.

Received 16 December 2013; Revised 26 December 2014; Accepted 13 January 2015

KEY WORDS: developer recommendation; multi-label learning; topic model; composite

1. INTRODUCTION

Because of the complexity of software development, bugs are inevitable. Bug resolution, which is the diagnosis, fixing, testing, and documentation of bugs, is an important activity in software development and maintenance. Bug tracking systems, such as Bugzilla and JIRA, help developers manage bug reporting, bug resolution, and bug archiving processes [1]. However, despite the availability of bug tracking systems, bug resolution still faces a number of challenges. The large number of new bug reports submitted to bug tracking systems daily increases the burden of bug triagers. For Eclipse, it was reported in 2005 that the number of bug reports received daily (around 200 reports/day) are too many for developers to handle [2]. As there are many bug reports requiring resolution and potentially hundreds or even thousands of developers working on a large project, it is non-trivial to assign a bug report to the appropriate developers.

Although only one developer is recorded as the final bug fixer, bug resolution is fundamentally a collaborative process [1, 3, 4]. Various developers contribute their knowledge, ideas, and expertise to resolve bugs. Figure 1 shows a bug report from Eclipse with

*Correspondence to: Xinyu Wang, College of Computer Science and Technology, Zhejiang University, #38 Zheda Road, Hangzhou, China.
†E-mail: wangxinyu@zju.edu.cn

Bug 215252 - frequent "invalid thread access"

Status: RESOLVED FIXED
Reported: 2008-01-14 14:57 EST by Chris Recoskie
Modified: 2008-10-29 15:59 EDT (History)
Product: Platform
Component: SWT
CC List: 15 users (show)
Version: 3.4
Hardware: PC Windows XP
Importance: P3 normal (vote)
Target Milestone: 3.4 M6
Assigned To: Steve Northover

Chris Recoskie CLA 2008-01-14 14:57:06 EST Description

I'm not sure 100% where the problem lies with this (hard to say if it's SWT, ...

Steffen Pingel CLA 2008-01-16 14:13:03 EST Comment 5

These startup warnings are most likely unrelated to the problem you are experiencing (see

Mik Kersten CLA 2008-01-18 11:50:44 EST Comment 6

As per comment#5, to the best of our knowledge all Mylyn-related parts of the stack traces

Felipe Heidrich CLA 2008-01-21 10:54:22 EST Comment 7

*** Bug 215791 has been marked as a duplicate of this bug. ***

This problem only happens on IBM Java 6 VM.
Bug 210459 (dup of Bug 210459) has a swt jar with debug information.

Dirk Baeumer CLA 2008-01-29 05:29:02 EST Comment 11

I was running with a hacked version of SWT which printed out some debugging information when getting the following exception:

Dani Megert CLA 2008-01-29 07:16:31 EST Comment 12

>I was running with a hacked version of SWT

Boris Bokowski CLA 2008-02-01 00:22:56 EST Comment 16

I just got a new Thinkpad and this is happening to me too.

Steve Northover CLA 2008-02-01 17:30:57 EST Comment 17

At this point, we should consider hacking our code to try and trick the VM into working.

Figure 1. Bug report of Eclipse with BugID=215252 – some comments are omitted or truncated.

BugID=215252.[††] In the figure, we notice that there are many developers that contribute their knowledge and post comments to resolve the bug. The bug reporter is Chris Recoskie, who provides detailed information of the bug. Seven other people, Steffen Pingel, Mik Kersten, Felipe Heidrich, Dirk Baeumer, Dani Megert, Boris Bokowski, and Steve Northover participated in the resolution of this bug report and contribute their expertise to the bug resolution process. The nine developers are the bug resolvers of this report. Among the nine developers, Steve Northover is recorded as the fixer of the bug (as specified in the assigned to field of the bug report).

In this paper, we are interested in developing an automated technique that processes a new bug report and recommends a list of developers that are likely to resolve it. We refer to this problem as *developer recommendation for bug resolution* (or developer recommendation, for short) [3, 4]. This is a part of the bug triaging problem [5] that would only recommend the fixer of a new bug report. Since bug fixing is a collaborative activity, aside from the final bug fixer, other developers involved in the bug resolution process also play a major role.

We propose a technique named *DevRec* that performs two kinds of analysis: bug report based analysis (BR-Based) and developer based analysis (D-Based). The combination of these two components would improve the overall performance further (Section 4.3). In our BR-Based analysis, we first measure the distance among bug reports. Given a new bug report, we find other similar past bug reports and recommend developers based on the developers of these past similar bug reports. In our D-Based analysis, we measure the distance between a potential developer with a bug report. We characterize the distance between a developer and a bug report by considering the characteristics of bug reports that the developer helps to resolve before. Given a new bug report, we would find

[††]https://bugs.eclipse.org/bugs/show_bug.cgi?id=215252

DOI: 10.1002/smr

developers with smallest distance to the new bug report. DevRec combines BR-Based and D-Based analyses to assign a score to each potential developer. A list of top-k most suitable developers would then be output.

There are a number of recent studies that are related to ours. The state-of-the-art work on automated *bug triaging* is the study by Tamrawi *et al.* that propose a fuzzy set method named Bugzie to recommend fixers given a new bug report [5]. Wu *et al.* address *bug resolution* problem by proposing a k-nearest neighbor search method named DREX to recommend developers given a bug report [3]. These two algorithms are the most recent studies related to the developer recommendation problem. Both of them return a list of candidate developers that are the most relevant for a bug report. Thus, we use these algorithms as baselines that we would compare with. Moreover, considering that the aforementioned approaches need to train a model based on historical bug reports, we also compare our method with an approach named NonTraining, which does not require any training phase.

We evaluate our approach on five datasets from different software communities: GNU Compiler Collection (GCC) [6], OpenOffice [7], Mozilla [8], Netbeans [9], and Eclipse [10]. In total, we analyze 107,875 bug reports. We measure the performance of the approaches in terms of recall@k. For the five datasets, DevRec could achieve recall@5 and recall@10 scores of up to 0.7989 and 0.8924, respectively. DevRec on average improves recall@5 and recall@10 scores of Bugzie by 57.55% and 39.39%, respectively. DevRec outperforms DREX by improving the average recall@5 and recall@10 scores by 165.38% and 89.36%, respectively. DevRec also outperforms NonTraining by improving the average recall@5 and recall@10 scores by 212.39% and 168.01%, respectively. Moreover, we evaluate the stableness of DevRec with different parameters, and the results show that DevRec is a stable approach.

This paper extends our preliminary study, published as a research paper in a conference [11]. It extends the preliminary study in various ways: more comparisons of DevRec over other state-of-the-art approaches are added, three additional research questions that explore the stableness of DevRec under different sets of parameters are investigated, more detailed descriptions of datasets and DevRec are added, the related work section is expanded, and a qualitative analysis of DevRec is added.

The main contributions of this paper are:

1. We propose DevRec, which performs both BR-Based analysis and D-Based analysis, to solve the developer recommendation problem.
2. We experiment on a broad range of datasets containing a total of 107,875 bugs to demonstrate the effectiveness of DevRec. We show that DevRec outperforms Bugzie [5], DREX [3], and NonTraining on the developer recommendation problem by a substantial margin. Statistically, tests show that the improvements are significant.
3. We evaluate DevRec on different sets of parameters, and the results show DevRec is robust on different parameters, which means developers do not need to spend much effort and time to configure our tool.

The remainder of the paper is organized as follows: We present the empirical study and preliminary materials in Section 2. We present our approach DevRec in Section 3. We present our experiments in Section 4. We present related work in Section 5. We conclude and mention future work in Section 6.

2. EMPIRICAL STUDY AND PRELIMINARIES

In this section, we first provide an example of bug resolution process in Section 2.1. Then, we present a simple empirical study on our collected datasets to understand developer recommendation problem in Section 2.2. Next, we introduce the preliminary materials, that is, Euclidean distance metric, multi-label k-nearest neighbor (ML-kNN) [12], and topic modeling [13], which would be used in our proposed approach DevRec, presented in Section 3.

DOI: 10.1002/smr

2.1. An example

Figure 1 presents an example bug resolution process. We notice different people have different roles in the bug resolution process. After Chris Recoskie reported the bug, Steffen Pingel found that 'startup warnings are most likely unrelated to the problem'. Then, Mik Kersten provided more information for this bug; Felipe Heidrich marked another bug as a duplicated bug of this one and pointed out the running environment and platform of the bug. Next, Dirk Baeumer, Dani Megert, and Boris Bokowski continue to contribute their knowledge for fixing the bug. Finally, Steve Northover fixed this bug.

In this paper, we focus on recommending three types of bug resolvers: bug fixers, bug contributors, and bug triagers. Bug fixers refer to developers who finally fix the bug (e.g., Steve Northover in Figure 1). Bug contributors refer to developers who contribute their knowledge to fix the bug (e.g., Steffen Pingel, Mik Kersten, Felipe Heidrich, Dirk Baeumer, DaniMegert, and Boris Bokowski in Figure 1). Bug triagers refer to developers who manage bug reports, such as mark duplicate bug reports, mark blocking bug reports, and assign bug reports to fixers. A person may have more than one role, for example, a bug triager may also be a bug fixer or a bug contributor. For example, in Figure 1, Felipe Heidrich is one of the bug triagers because he/she not only marked the duplicate bug report but he also provided the runtime environment of the bug. Thus, he/she is also a bug contributor. Notice although bug triagers do not directly contribute their knowledge for fixing bugs, they also play an important role for bug resolution. For example, a bug triager may contribute his/her knowledge to find a suitable fixer, or after he/she marks bug reports to be a duplicate of one another, other developers could find more information from other related bugs [14, 15]. Because of the importance of bug triagers, in this paper, we also recommend triagers. In this paper, for simplicity reason, unless otherwise specified, a developer can be a bug fixer, a bug contributor, or a bug triager.

2.2. Empirical study

A typical bug report contains many fields, such as reporter, fixer, creation time, modification time, bug version, platform, CC list, and comment list. In this work, we collected five pieces of information from the bug report fields including bug summary, bug description, product affected by the bug, component affected by the bug, and developers participated in the bug resolution process (i.e., bug resolvers). The details of these pieces of information, illustrated based on the bug report shown in Figure 1, are presented Table I.

In this paper, we collected the following five datasets from different software development communities: GCC, OpenOffice, Mozilla, Netbeans, and Eclipse. Table II shows the statistics of the five datasets that we collected. The columns correspond to the project name (Project), the time period of collected bug reports (Time), the number of collected reports (# Reports), the number of unique bug resolvers (# Resolvers), the number of terms (i.e., words) in the bug reports (# Terms), the average number of bug resolvers per bug report (# Avg.Re.), the number of different products (# Pro.), and the number of different components (# Comp.), respectively. All bug reports and their data are downloaded from their bug tracking systems. We collected bug reports with status 'closed' and 'fixed'. For these reports, the set of bug resolvers has been identified. Unless the bug is reopened in the future, no additional resolver would be added. Note that the average number of

Table I. Collected information from bug reports.

Information	Details	Example
Summary	Brief description of a bug.	frequent 'invalid thread access'
Description	Detailed description of a bug.	I'm not sure 100% where the problem …
Product	Product affected by the bug.	Platform
Component	Component affected by the bug.	Standard Widget Toolkit
Developers	Bug resolvers, that is, developers that contribute to the bug resolution process excluding the reporter.	Steffen Pingel, Mik Kersten, Felipe Heidrich, Dirk Baeumer, Dani Megert, Boris Bokowski, Steve Northoverr, Olivier Thomann, and Grant Gayed

DOI: 10.1002/smr

Table II. Statistics of collected bug reports.

Project	Time	# Reports	# Resolvers	# Terms	# Avg.Re.	# Pro.	# Comp.
GCC	2008-01-01–2010-10-28	5,742	650	3,916	2.56	2	40
Openoffice	2007-03-01–2013-04-07	15,448	1,656	8,291	2.60	37	100
Mozilla	2009-6-23–2010-06-03	26,046	3,812	10,232	2.89	56	511
Netbeans	2008-01-01–2010-01-11	26,240	2,274	10,255	2.45	38	336
Eclipse	2005-01-01–2008-07-23	34,399	3,086	11,234	1.88	114	540

resolvers represents the activity degree of the development community; the higher the average number of developers is, the more active is the community. In our dataset, Mozilla is the most active community; for each bug report, on average, there are 2.89 developers that help to resolve the bug.

We identify bug resolvers by looking at the 'assigned to' field and the list of comments in the bug reports. A bug resolver can be a developer who participates or contributes idea in a bug discussion or a developer who contributes code to fix a bug. In this paper, we do not differentiate between them; we recommend all developers who contribute to the bug resolution activity. We notice that for many bug reports, the 'assigned to' fields are set to generic names that do not specify developers. In GCC, 45.29% of the bug reports are assigned to 'unassigned'; In OpenOffice, 12.99% of the bug reports are assigned to generic names such as 'issues', 'needsconfirm', and 'swneedsconfirm'; In Mozilla, 15.1% of the bug reports are assigned to 'nobody'; In Netbeans, 8.59% of the bug reports are assigned to 'issues'; In Eclipse, 20.12% of the bug reports are assigned to generic names like 'platform-runtime-inbox', 'webmaster', and 'AJDT-inbox'. Because these generic names are not actual developers, in this paper, we do not want to recommend them, and thus, they are excluded from our datasets.

Finally, we also explore the relationship between bug resolvers, bug reports, and the product and component fields. A software system contains many products, and each product may contain various components. For example, Eclipse has 114 products and 540 components. Columns #Product and #Component of Table II show the number of products and components for the five software projects. We show some products and components of Eclipse in Figure 2. Table III presents some statistics on the number of bug resolvers and bug reports per product. Table IV presents similar statistics on the number of bug resolvers and bug reports per component. The first two columns list the maximum and average number of bug resolvers per product (or per component) for each software project. There are variations in the maximum and average number of bug resolvers per product (or per component) across the five projects. The last column lists the proportion of bug reports for the top ten products or components with the most number of bug resolvers. We notice

Figure 2. Some products and components of Eclipse.

Table III. Relationship between number of bug resolvers and bug reports with the product field.

Project	Max	Avg	Top 10
GCC	634	331	100%
OpenOffice	607	104	74.97%
Mozilla	1641	143.52	70.57%
Netbeans	567	155.32	45.27%
Eclipse	707	50.85	54.32%

DOI: 10.1002/smr

Table IV. Relationship between number of bug resolvers and bug reports with the component field.

Project	Max	Avg	Top 10
GCC	264	49.93	85.38%
OpenOffice	818	49.62	76.07%
Mozilla	895	34.28	38.11%
Netbeans	515	32.16	31.48%
Eclipse	472	17.31	31.64%

that the number of bug reports per product (or per component) is skewed. Most of the bug reports are for the top ten products and components, and for these products and components, there are many bug resolvers. We notice, for example, for OpenOffice, 74.97% and 76.07% bug reports belong to the top ten products (out of 37 products) and top ten components (out of 100 components). For Eclipse, 54.32% and 31.64% bug reports belong to the top ten products (out of 114 products) and top ten components (out of 514 components). Thus, information of the product and the component that is affected by the bug is not sufficient to decide suitable developers to be involved in the bug resolution process.

2.3. Preliminaries

2.3.1. Bug report representation and Euclidean distance metric. A bug report b can be represented by a vector of feature values. A feature represents one characteristic of the bug report b. In this paper, we use four types of features, that is, term, topic, product, and component (Section 3.1). For example, in Figure 1, we extract words from the summary and description texts as term features, and we extract topic features from these words by using topic models [13]; we also use the product and component field values as the product and component feature values. Thus, a bug report b can be denoted as $(p_1, p_2, p_3, \ldots, p_n)$, where p_i, $i \in \{1, 2, \ldots n\}$, is the value of b's i^{th} feature.

Suppose that there are two different bug reports $b_1 = (p_1, p_2, \ldots, p_n)$ and $b_2 = (q_1, q_2, \ldots, q_n)$, then the Euclidean distance between b_1 and b_2 is defined by the following:

$$\text{Distance}(b_1, b_2) = \sqrt{(p_1 - q_1)^2 + \cdots + (p_n - q_n)^2} \tag{1}$$

2.3.2. Multi-label classification. The task of multi-label classification is to predict for each data instance, a set of labels that applies to it [16]. Standard classification only assigns one label to each data instance. However, in many settings, a data instance can be assigned by more than one label. For developer recommendation problem, each data instance (i.e., a bug report) can be assigned by multiple labels (i.e., developers).

Multi-label k-nearest neighbor is a state-of-the-art algorithm in the multi-label classification literature [12]. For a new instance X_{new}, ML-KNN processes its k-nearest neighbors $KNN(X_{new})$ in the training dataset. For a label d_l in the label set D, it computes the number of training data instances in $KNN(X_{new})$ with label d_l. We denote the number of data instances with label d_l as $C_{X_{new}}(d_l)$.

Based on the aforementioned count, ML-KNN computes the estimated probability of the new instance X_{new} to belong to label d_l (denoted as $H_1^l(X_{new})$) and the estimated probability of the new instance to NOT belong to label d_l (denoted as $H_0^l(X_{new})$). These two estimates do not necessarily sum up to 1. The aforementioned two estimated probabilities are computed for every label in the label space D. If H_1^l is larger than H_0^l, the label d_l would be assigned to X_{new}. More than one label satisfying the aforementioned two estimated probabilities could be assigned to X_{new}. Instead of outputting predicted labels for X_{new}, we modify ML-KNN to output a score that combines the two probability estimates for each label d_l as follows:

DOI: 10.1002/smr

$$Score^{ML-KNN}(X_{new}, d_l) = \frac{H_1^l(X_{new})}{H_0^l(X_{new}) + H_1^l(X_{new})}$$

This score is the *relative likelihood* of d_l to be assigned to X_{new}.

2.3.3. Topic modeling. A textual document of a particular topic is likely to contain a particular set of terms (i.e., words). For example, a document about a user interface bug is likely to contain terms such as window and button. A document can be a mixture of several topics. Topic modeling models this phenomenon. In our setting, a document is a bug report, and a topic is a higher-level concept corresponding to a distribution of terms. With topic modeling, given a new bug report, we extract a set of topics along with the probabilities that they appear in the bug report.

Latent Dirichlet allocation (LDA) is a well-known topic modeling technique [13]. LDA is a generative probabilistic model of a textual corpus (i.e., a set of textual documents) that takes as inputs: a training textual corpus and a number of parameters including the number of topics (K) considered. In the training phase, for each document s, we would get a *topic proportion vector* θ_s, which is a vector with K elements, and each element corresponds to a topic. The value of each element in θ_s is a real number from 0 to 1, which represents the proportion of the words in s belonging to the corresponding topic in s. After training, the LDA is used to predict the topic proportion vector θ_m for a new document m. By this, we map the terms in the document m into a topic proportion vector $\theta_{m,}$ which contains the probabilities of each topic to be present in the document.

After training, the LDA can be used to predict the topic for every term in a new document. For a new document m, considering K topics, we compute its topic vector z_m based on the topics assigned to its constituent terms as

$$\begin{aligned} z_m &= < t_1, \ldots, t_K >, \quad where \\ t_i &= \frac{\#\ words\ \ assigned\ \ the\ \ i^{th}\ \ topic\ \ in\ \ m}{\#\ words\ \ in\ \ m} \end{aligned} \tag{2}$$

By this, we map the terms in the document m into a topic vector that contains the probabilities of each topic to be present in the document.

3. DEVREC: A COMPOSITE OF BR-BASED AND D-BASED ANALYSIS

In this section, we propose our DevRec method, to solve the developer recommendation problem. This section includes three parts: In Section 3.1, we present BR-Based analysis. In Section 3.2, we present D-Based analysis. Finally, in Section 3.3, we present a composite of BR-Based and D-Based analyses that would result in DevRec.

3.1. BR-Based analysis

Bug reports based analysis takes in a new bug report BR^{new} whose resolvers (i.e., developers that contribute to bug resolution) are to be predicted and outputs a score for each potential resolver. BR-Based analysis finds the k-nearest bug reports to BR^{new} whose resolvers are known and based on these resolvers, recommend developers for BR^{new}. There are two things that we need to do to realize our BR-Based analysis. First, we need to find the k-nearest neighbors of BR^{new} (i.e., k-nearest bug reports to BR^{new}). Next, we need a machine learning technique that could infer the resolvers of BR^{new} from the resolvers of its k-nearest neighbors. We describe how we perform these two steps in the following subsections.

3.1.1. Finding k-nearest neighbors. To find k-nearest neighbors of BR^{new}, we first need to find a set of features that characterize bug reports. Next, we need a distance metric that measures the

DOI: 10.1002/smr

distance between one bug report to another. We use the following features to characterize a bug report:

1. (Terms) This is a multi-set of stemmed non-stop words [17] that appear in the summary and description of the bug report. Stop words are words that carry little meaning, for example, I, you, he, and she. We remove all such stop words. We also remove the numerical values. Stemming is the process of reducing a word to its root form, for example, both 'reading' and 'reads' can be stemmed to 'read'. Each of the words is a feature. The value of each feature is the number of times the corresponding word appears in a bug report.
2. (Topics) This is a set of topics that appears in the summary and description of the bug report. We make use of latent Dirichlet allocation described in Section 2, which reduces a document into a set of topics along with the probabilities of the document to belong to each of the topics in the set. Each of the topics is a feature. The value of each feature is the probability of the corresponding topic to belong to the bug report.
3. (Product) This is the product that is affected by the bug as recorded in the bug report. Each possible product is a binary feature. The value of each of these features is either 0 or 1 depending if the bug report is for the corresponding product or not.
4. (Component) This is the component that is affected by the bug as recorded in the bug report. Each possible component is a binary feature. The value of each of these features is either 0 or 1 depending if the bug report is for the corresponding component or not.

Each bug report would then be represented as a vector of feature values (aka. a feature vector), which contains all of the four feature types, that is, terms, topics, product, and component features. The distance between two bug reports could be calculated by simply computing the Euclidean distance of two vectors (Section 2). In our approach, for each of the four feature types, we assign the same weights. Based on this distance, we can find the k-nearest neighbors of a new bug report.

3.1.2. Infer resolvers of BR^{new}. Given the k-nearest neighbors, we would like to predict the resolvers of BR^{new} based on the resolvers of its k-nearest neighbors. We consider each developer as a class label, each bug report as a data point, and each bug report with known resolvers as a training data point. Under this setting, the problem is reduced to a multi-label classification problem: given a data point (i.e., a bug report), predict its labels (i.e., its resolvers).

We leverage the state-of-the-art work on multi-label learning namely ML-KNN proposed by Zhang and Zhou [12]. We have provided a short description of this approach in Section 2. The ML-KNN approach outputs the relative likelihood of a label to be assigned to a data point. After the application of this approach, we would have assigned for each potential developer d, a score that denotes the likelihood of this developer d to be a resolver of BR^{new}, denoted by $BRScore_{BR^{new}}(d)$.

3.2. D-Based analysis

For D-Based analysis, we model the affinity of a developer to a bug report. A developer might have resolved past bug reports before. This experience of the developer could be used to model the affinity of the developer to various features of a bug report. We consider four types of features: terms, topics, component, and product. Similar features are used by the BR-Based analysis. However, in D-Based analysis, rather than finding distances between bug reports, we measure distances between bug reports and developers. We call the distance between a developer and a term, a topic, a component, and a product in a bug report as *term affinity*, *topic affinity*, *component affinity*, and *product affinity*, respectively. We describe how the scores measuring the affinity of a term, topic, component, and product with a bug report could be computed in the following subsections.

3.2.1. Terms affinity score. We use the following formula to compute the term affinity score of a bug report b to a developer d:

DOI: 10.1002/smr

$$Terms_b(d) = 1 - \prod_{t \in b}\left(1 - \frac{n_{d,t}}{n_t + n_d - n_{d,t}}\right) \quad (3)$$

where t refers to the terms in b and n_d, n_t, and $n_{d,t}$ refer to the number of bug reports that a developer d has contributed to bug resolution activities; the number of reports term t appears and the number of reports resolved by developer d that contains term t. We characterize each developer by the top TC terms of the highest affinity scores. The default number of terms (i.e., TC) for each developer is set to 10. The aforementioned formula is based on [5].

3.2.2. Topics affinity score. In natural language processing, a topic represents a distribution of terms (or words), and a document (in our setting, a bug report) is a distribution of topics. We use LDA [13] to get the topic distribution for each bug report. Section 2 provides a description of LDA. Using LDA, we map the term space of the original document into the topic space. Each document or bug report corresponds to one topic vector where a topic vector is simply a set of mappings from topics to the probabilities of the corresponding document to belong to these topics.

Consider a set of topic vectors T corresponding to the set of all bug reports. Let T_d refers to the topic vectors corresponding to bug reports that developer d helps in the bug resolution process. Also, given a topic vector v, let $v[t]$ denotes the probability of the corresponding bug report to belong to topic t – it is an entry in the topic vector v corresponding to topic t. For a bug report b, we denote $b[t]$ as the probability of the bug report b to belong to topic t. The topic affinity score of b to a developer d is given by the following:

$$Topics_b(d) = 1 - \prod_{t \in b}\left(1 - \frac{\sum_{v \in T_d} v[t]}{\sum_{v' \in T} v'[t]} \times b[t]\right) \quad (4)$$

$t \in b$ denotes a topic contained in the bug report b. Informally put, the aforementioned formula would be very small if the bug reports that developer d helps in the bug resolution process share very little topics with the topics contained in bug report b. It would be large if they share a lot of common topics.

3.2.3. Product and component affinity scores. A developer d might be biased toward certain products and components. The definitions of product and component affinity score defined here are different from those of terms and topics affinity scores. This is so because each bug report has only one product and one component.

Consider a bug report collection B. Let B_d refers to bug reports where a developer d participated before. Also, given product p, let $b[p]$ denotes whether bug report b is for product p: $b[p] = 1$ if b is for product p and $b[p] = 0$ otherwise (notice that for all the products, only one product p has $b[p] = 1$). Also, let p_b denotes the value of the product field of bug report b. The product affinity score $Product_b(d)$ for bug report b and developer d is given by the following:

$$Product_b(d) = \frac{\sum_{b \in B_d} b[p_b]}{\sum_{b' \in B} b'[p_b]} \quad (5)$$

Similarly, given component c, let $b[c]$ denotes whether bug report b is for component c, $b[c] = 1$ if b is for component c, and $b[c] = 0$ otherwise (notice that for all the components, only one component c has $b[c] = 1$). Also, let c_b denotes the value of the component field of bug report b. The component affinity score $Component_b(d)$ for b to a developer d is given by the following:

$$Component_b(d) = \frac{\sum_{b \in B_d} b[c_b]}{\sum_{b' \in B} b'[c_b]} \quad (6)$$

Informally put, the aforementioned two scores would be very small if the bug report b does not share any product or component with past bug reports that developer d participated before. The two scores

DOI: 10.1002/smr

Table V. An example dataset with two topics, two products, three components, and two Developers. PART = Participate. X = Does not participate.

BugID	Topic 1	Topic 2	Prod.	Comp.	Dev 1 (D1)	Dev 2 (D2)
Train 1	0.1	0.9	P1	C1	PART	X
Train 2	0.8	0.2	P1	C2	PART	PART
Train 3	0	1	P2	C3	X	PART
Train 4	0.5	0.5	P1	C1	X	PART
Test 1	0.4	0.6	P1	C1	?	?

would be high if many bug reports that developer d participated before share the same product and component as bug report b.

3.2.4. An example. To illustrate topic affinity, product affinity, and component affinity scores, we take an example bug report dataset shown in Table V, which has two topics, two types of product, three types of component, and two developers. The bug report with identifier 'Test 1' is the bug report whose resolvers are to be predicted.

Developer 1 participated in two bug reports, 'Train 1' and 'Train 2'. Developer 2 participated in three bug reports, 'Train 2', 'Train 3', and 'Train 4'. The topic affinity score of developer $D1$ and bug report 'Test 1' can be computed as

$$Topics_{Test1}(D1) = 1 - \left(1 - \frac{0.1 + 0.8}{0.1 + 0.8 + 0.5} \times 0.4\right) \times \left(1 - \frac{0.9 + 0.2}{0.9 + 0.2 + 1 + 0.5} \times 0.6\right) = 0.4458$$

The value of the product field of bug report 'Test 1' is P1. In the training set, there are three bug reports with their product field set as P1 (i.e., 'Train 1', 'Train 2', and 'Train 4'). For 'Train 1' and 'Train 2', developer $D1$ participated in the bug resolution activity. The product affinity score of developer $D1$ and bug report 'Test 1' can be computed as follows:

$$Product_{Test1}(D1) = \frac{2}{1 + 1 + 1} = 0.67$$

The value of the component field of bug report 'Test 1' is C1. In the training set, there are two bug reports with their product fields set as C1 (i.e., 'Train 1' and 'Train 4'). For 'Train 4', Developer 1 participated in the bug resolution activity. The component affinity score of developer $D1$ and bug report 'Test 1' can be computed as

$$Component_{Test1}(D1) = \frac{1}{1 + 1} = 0.5$$

For developer *D2*, a similar analysis can be performed to compute the topic, product, and component affinity scores.

3.2.5. D-BASED SCORE

In the previous subsections, we define the affinity scores for term, topic, product, and component. Definition 1 defines a way to combine all of these scores into a single score referred to as D-Based score.

DOI: 10.1002/smr

Definition 1

(D-Based Score.) Consider a bug report b and a developer d. Let us denote its term affinity score, topic affinity score, product affinity score, and component affinity score as $Terms_b(d)$, $Topics_b(d)$, $Product_b(d)$, and $Component_b(d)$, respectively. The D-Based score for developer d and bug report b is given by the following:

$$\begin{aligned} DScore_b(d) = \beta_1 \times Terms_b(d) + \beta_2 \times Topics_b(d) + \\ \beta_3 \times Product_b(d) + \beta_4 \times Component_b(d) \end{aligned} \tag{7}$$

Where $\beta_1, \beta_2, \beta_3, \beta_4 \in [0, 1]$ represent the different contribution weights of the various affinity scores to the overall D-Based score.

Algorithm 1 *EstimateWeights*: Estimation of $\gamma_1, \gamma_2, \gamma_3, \gamma_4$, and γ_5 in *DevRec*

1: EstimateWeights($B, D, T, EC, ITER, SampleSize$**)**
2: Input:
3: B: Bug Report Collection
4: D: Developer Collection
5: T: Bug Report Topic Distribution
6: EC: Evaluation Criterion
7: $ITER$: Maximum Number of Iterations (Default Value = 10)
8: $SampleSize$: Sample Size
9: Output: $\gamma_1, \gamma_2, \gamma_3, \gamma_4, \gamma_5$
10: Method:
11: Build BR-Based Analysis component using B;
12: Build D-Based Analysis component using B;
13: Sample a small subset $Samp_B$ of B of size $SampleSize$;
14: for all bug report $b \in Samp_B$, and developer $d \in D$ **do**
15: Compute the BR-Based score, i.e., $BRScore_b(d)$
16: Compute the Terms Affinity score, i.e., $Terms_b(d)$
17: Compute the Topic Affinity score, i.e., $Topics_b(d)$
18: Compute the Product Affinity score, i.e., $Product_b(d)$
19: Compute the Component Affinity score, i.e., $Component_b(d)$
20: end for
21: while iteration times $iter < ITER$ **do**
22: **for all** i from 1 to 5 **do**
23: Choose $\gamma_i = Math.random()$
24: **end for**
25: **for all** i from 1 to 5 **do**
26: $\gamma_i^{best} = \gamma_i$
27: **repeat**
28: Compute the *DevRec* scores according to Equation 9
29: Evaluate the effectiveness of the combined model on $Samp_B$ and D based on EC
30: **If** EC score of γ_i is better than that of γ_i^{best} **then**
31: $\gamma_i^{best} = \gamma_i$
32: **end if**
33: Increase γ_i by 0.01
34: **Until** $\gamma_i \geq 1$
35: $\gamma_i = \gamma_i^{best}$
36: **end for**
37: end while
38: Return $\gamma_1, \gamma_2, \gamma_3, \gamma_4, \gamma_5$ which give the best result based on EC

DOI: 10.1002/smr

3.3. DevRec: a composite method

As shown in previous sections, we can get the BR-Based score and D-Based scores for each new bug report b. In this section, we propose DevRec, which is a composite method that combines both BR-Based analysis and D-Based analysis. A linear combination of the BR-Based and D-Based scores defined in Definition 2 is used to compute the final DevRec score.

Definition 2

(DevRec Score.) Consider a bug report b and a developer d. Let the BR-Based score and D-Based score be $BRScore_b(d)$ and $DScore_b(d)$, respectively. The DevRec score that computes the expert ranking score of developer d with respect to bug report b is given by the following:

$$DevRec_b(d) = \alpha_1 \times BRScore_b(d) + \alpha_2 \times DScore_b(d) \tag{8}$$

Where $\alpha_1, \alpha_2 \in [0, 1]$ represent the contribution weights of BRScore and DScore to the overall DevRec score. If we unfold $DevRec_b(d)$, we get the following:

$$\begin{aligned} DevRec_b(d) = {} & \gamma_1 \times BRScore_b(d) + \gamma_2 \times Terms_b(d) \\ & + \gamma_3 \times Topics_b(d) + \gamma_4 \times Product_b(d) + \gamma_5 \times Component_b(d) \end{aligned} \tag{9}$$

Where $\gamma_1, \gamma_2, \gamma_3, \gamma_4, \gamma_5 \in [0, 1]$.

To automatically produce good γ values for DevRec, we propose a sample-based greedy method. Because of the large size of bug report collection B, we do not use the whole collection to estimate gamma weights, instead, we randomly sample a small subset of B. In this paper, by default, we set the sample size as 10% of the number of bug reports in B.

Algorithm 3.3 presents the pseudocode to estimate good γ values. We first build the BR-Based analysis component and the D-Based analysis component using the whole bug report collection B (Lines 11 and 12). After we sample a small subset $Samp_B$ from B (Line 13), we compute the BR-based score, term affinity score, topic affinity score, component affinity score, and product affinity score for each bug report in $Samp_B$ and each developer in the whole developer collection D (Lines 14–20). Next, we iterate the whole process of choosing good γ values *ITER* times (Line 22). For each iteration, we first randomly assign a value between 0 to 1 to each γ_i, for $1 \leq i \leq 5$ (Lines 22–24). Next, for each γ_i, we fix the values of γ_p where $1 \leq p \leq 5$ and $p \neq i$, and we increase γ_i incrementally by 0.01 at a time and compute the *EC* scores (Lines 25–36). By default, we set the input criterion *EC* as Recall@k [18, 19] (Definition 3). Algorithm 3.3 would return $\gamma_1, \gamma_2, \gamma_3, \gamma_4, \gamma_5$, which give the best result based on *EC*.

4. EXPERIMENTS

We evaluate our DevRec method on the collected datasets described in Table II. We compare our method with Bugzie [5] and DREX [3]. The experimental environment is a Windows 7 64-bit, Intel (R) Xeon(R) 2.53 GHz server with 24 GB RAM.

4.1. Experimental setup

For each bug report, we extract its bug ID, bug summary and description information, bug product, bug component and bug resolvers. We extract the stemmed non-stop terms (i.e., words) from the summary and description information. We do some pre-processing for the bug report collections similar to DREX [3]: For the small-scale bug report collections, such as GCC and OpenOffice, we delete the terms that appear less than 10 times; while for large-scale bug report collections, such as Mozilla, Netbeans, and Eclipse, we delete the terms that appear less than 20 times. For each of the five bug report collections, we remove developers who

DOI: 10.1002/smr

appear less than 10 times. Because they are not active, recommending these developers does not help much in bug resolution.

To simulate the usage of our approach in practice, we use the same longitudinal data setup described in [5, 20]. The bug reports extracted from each bug repository are sorted in chronological order of creation time and then divided into 11 non-overlapping frames (or windows) of equal sizes. The validation process proceeds as follows: First, we train using bug reports in frame 0 and test the bug reports in frame 1. Then, we train using bug reports in frame 0 and frame 1 and use the similar way to test the bug reports in frame 2 and so on. In the final fold, we train using bug reports in frame 0–9 and test the bug reports in frame 10. In the training data, we have, for each report, both the features that characterize the bug report and the set of resolvers. We use these to train DevRec, Bugzie, and DREX. In the test data, for each bug report, we use the features that characterize the bug report to predict the set of resolvers. We use the resolvers recorded in the bug repository as the ground truth.

DevRec accepts an evaluation criteria *EC* as a parameter. In this work, we consider two evaluation criteria to compare DevRec with Bugzie and DREX: recall@5 and recall@10. These measures are well-known information retrieval measures [17], and recall@10 has also been used to evaluate DREX [3]. DevRec uses LDA that accepts a number of parameters. For LDA, we set the maximum number of iterations to 500 and the hyperparameters α and β to $50/T$ and 0.01, respectively, where T is the number of topics. By default, we set the number of topics T to 5% of the number of distinct terms (i.e., words) in the training data. We use JGibbsLDA[§] as the LDA implementation. We use percentages rather than a fixed number as the amount of training data varies for different datasets and different test frames (following longitudinal study setup [5, 20] described in Section 4.1). If there are more distinct terms, there are likely to be more topics. For the BR-Based analysis of DevRec, by default, we set the number of neighbors to 15.

Bugzie was first proposed for the bug assignment problem. However, since it can rank each developer based on the developer's suitability to a bug report, we can use it for our problem too. For Bugzie, there are two parameters: the developer cache size and the number of descriptive terms. We use 100% developer cache size and set the number of descriptive terms to 10. These settings have been shown to result in the best performance [5]. DREX was proposed to address the same problem as ours. We compare our approach to the simple frequency variants of DREX, which has been shown to result in the best performance [3]. We set the number of neighbors of DREX to 15 (the same as that of DevRec). Notice all of DevRec, Bugzie, and DREX require a training phase, which a model is built based on the historical bug reports.

To reduce the effort due to the training, we also propose a method named NonTraining. NonTraining still needs a number of historical bug reports but does not need to build any machine learning model based on the historical bug reports. Considering that there is no machine learning model built on the historical bug reports, we refer to this approach as NonTraining. To recommend the developers for a new bug report, NonTraining first finds the top-k most similar bug reports in the training data based on our proposed four types of features, that is, terms, topics, products, and component. Based on developers who resolve these bug reports, NonTraining then recommends developers by using frequency counting. We set top 15 similar bug reports as we do for DevRec to make a fair comparison, and we use Euclidean distance as the similarity metric.

We evaluate the performance of our DevRec with two metrics, that is, recall@k and precision@k. The definitions of recall@k and precision@k are as follows:

Definition 3

(Recall@k and Precision@k.) Suppose that there are m bug reports. For each bug report b_i, let the set of its actual bug resolvers be D_i. We recommend the set of top-k developers P_i for b_i according to our method. The recall@k and precision@k for the m bug reports are given by the following:

$$\text{Recall@k} = \frac{1}{m}\sum_{i=1}^{m}\frac{|P_i \cap D_i|}{|D_i|} \tag{10}$$

[§]http://jgibblda.sourceforge.net/

DOI: 10.1002/smr

$$\text{Precision@k} = \frac{1}{m}\sum_{i=1}^{m}\frac{|P_i \cap D_i|}{|P_i|} \tag{11}$$

For example, suppose there are two bug reports and three developers participated in the bug resolution process. For bug report 1, the resolvers are {1,2,3}, and for bug report 2, the resolvers are {1}. The top 2 predicted developers are {1,3} and{2,3} for bug reports 1 and 2, respectively. Then the recall@2 and precision@2 are

$$\text{Recall@2} = \frac{1}{2}\left(\frac{|\{1,3\}\cap\{1,2,3\}|}{|\{1,2,3\}|} + \frac{|\{2,3\}\cap\{1\}|}{|\{1\}|}\right) = \frac{1}{3} \tag{12}$$

$$\text{Precision@2} = \frac{1}{2}\left(\frac{|\{1,3\}\cap\{1,2,3\}|}{|\{1,3\}|} + \frac{|\{2,3\}\cap\{1\}|}{|\{2,3\}|}\right) = \frac{1}{2} \tag{13}$$

We are interested to answer the following research questions:

RQ1 *How effective is our proposed DevRec? How much improvement could our proposed approach gain over Bugzie, DREX, and NonTraining?*

Tamrawi *et al.* propose a fuzzy set method named Bugzie to recommend fixers given a new bug report [5]. Wu *et al.* address bug resolution problem by proposing a k-nearest neighbor search method named DREX to recommend developers given a bug report [3]. In this research question, we investigate the extent our approach (DevRec) outperforms these state-of-the-art approaches. We also compare our approach with NonTraining, which directly recommends developers without a training phase. Answer to this research question would shed light to whether our proposed approach advances existing state-of-the-art approaches. To answer this research question, we compare the Recall@5, Recall@10, Precision@5, and Precision@10 scores of DevRec with those of Bugzie, DREX, and NonTraining. For each bug report, we could compute its Recall@5, Recall@10, Precision@5, and Precision@10 scores by using DevRec, Bugzie, DREX, and NonTraining, respectively. Thus, for each technique and each metric, we have a list of scores. We then apply the Wilcoxon signed-rank test [21] on these lists of scores to test whether the improvements of DevRec over Bugzie, DREX, and NonTraining are significant (at $\alpha = 0.05$).

RQ2 *What is the performance of the BR-based component and D-based component*?

DevRec has two components (i.e., BR-based and D-based components); we would like to investigate the performance of each of them. We want to see whether the combination of the two components results in better or poorer performance. To answer this research question, we compare the Recall@5, Recall@10, Precision@5, and Precision@10 scores of DevRec with those of BR-based component and D-based component. We also apply Wilcoxon signed-rank test [21] to test whether the improvements of DevRec over BR-based and D-based components are significant.

RQ3 *What is the effect of varying the number of topics to the performance of DevRec*?

Latent Dirichlet allocation generates topics from a set of documents; however, the number of topics needs to be manually specified. Previous study shows that different numbers of topics would affect the performance of LDA in several software engineering tasks [22]. Thus, in this research question, we would like to investigate how the performance of DevRec varies for various numbers of topics. Ideally, since users do not know how to choose the best number of topics for a new dataset, the performance of DevRec should be relatively stable for different numbers of topics, as long as they are within a reasonable range. For the other research questions, by default, the number of topics is

DOI: 10.1002/smr

set to be 5% of the number of distinct terms in our training dataset. To answer this research question, we vary the number of topics from 1%, 3%, 5%, 7%, 9%, and 11% of the number of distinct terms in our training data.

RQ4 *What is the effect of varying the number of neighbors to the performance of DevRec*?

In the BR-based component, for a new bug report, DevRec would find its k-nearest neighbors. The value of k needs to be manually specified. In this research question, we would like to investigate whether different numbers of k would affect the stableness of DevRec. Ideally, because users do not know how to choose the best value of k for a new dataset, the performance of DevRec should be relatively stable for different numbers of neighbors k, as long as they are within a reasonable range. For the other research questions, by default, k is set to be 15. To answer this research question, we vary the number of neighbors k from 5, 10, 15, 20, and 25.

RQ5 *What is the effect of varying the number of terms for each developer to the performance of DevRec*?

In the term affinity score in D-based component, we characterize each developer by the top *TC* terms of the highest affinity scores. In this research question, we would like to investigate whether different number of terms *TC* would affect the stableness of DevRec. Ideally, because users do not know how to choose the best value of *TC* for a new dataset, the performance of DevRec should be relatively stable for different number of terms *TC*, as long as they are within a reasonable range. For the other research questions, by default, the number of terms (i.e., *TC*) for each developer is set to 10. To answer this research question, we vary the *TC* from 5, 10, 15, 20, and 25.

4.2. RQ1: performance of DevRec

In this section, we compare DevRec with other state-of-the-art methods, namely Bugzie and DREX. Table VI presents the recall@5, recall@10, precision@5, and precision@10 of DevRec compared with Bugzie and DREX and the improvement of DevRec over Bugzie (Impro.B.) and over DREX (Impro.D.), respectively. The statistically significant improvements are marked in bold. The p-values for all the Wilcoxon signed-rank tests are less than $2.2e^{-16}$, which indicate that the results are statistically significant. The recall@5 and recall@10 of DevRec vary from 48.26% to 79.89% and from 60.63% to 89.24%, respectively. The precision@5 and precision@10 of DevRec vary from 21.00% to 31.96% and 13.31% to 18.59%, respectively.

From Table VI, the improvement of our method over Bugzie is substantial and statistically significant. DevRec outperforms Bugzie by 57.55% and 39.39% for average recall@5 and recall@10, respectively. For the Eclipse dataset, DevRec achieves the highest improvement of 108.79% and 78.55% over Bugzie for recall@5 and recall@10, respectively. We notice that the result shown in Table VI is different from the result presented in [5] because of several reasons: First, the problem considered there is different from ours. In [5], it addresses bug triaging problem, that is, one bug report has only one fixer. In this work, we address the developer recommendation problem, that is, one bug report has multiple bug resolvers. Second, we drop generic names, for example, nobody, issues, unassigned, as they do not identify particular developers.

From Table VI, the improvement of our method over DREX is substantial and statistically significant. DevRec outperforms DREX by 165.38% and 89.36% for average recall@5 and recall@10, respectively. For the Mozilla dataset, DevRec achieves the highest improvement of 426.08% and 136.67% over DREX for recall@5 and recall@10, respectively.

From Table VI, the improvement of our method over NonTraining is substantial and statistically significant. DevRec outperforms NonTraining by 212.39% and 168.01% for average recall@5 and recall@10, respectively. For the Eclipse dataset, DevRec achieves the highest improvement of 481.86% and 357.88% over NonTraining for recall@5 and recall@10, respectively. Comparing DevRec, Bugzie, DREX, and NonTraining, we notice in general that NonTraining has the lowest recall@k and precision@k values. This is the case because NonTraining does not build any machine learning model based on the historical bug reports.

DOI: 10.1002/smr

Table VI. Recall@5, Recall@10, Precision@5, and Precision@10 of DevRec compared with Bugzie and DREX, and the improvement of DevRec over Bugzie (Impro.B.) and over DREX (Impro.D.). The last row shows the average Recall@5, Recall@10, Precision@5, and Precision@10 scores of DevRec, Bugzie, and DREX, and the average improvement. Statistically significant improvements are highlighted in bold.

2*Projects	Recall@5						
	DevRec	Bugzie	Impro.Bugzie	DREX	Impro.DREX	NonTraining	Impro.Non
GCC	56.33%	47.43%	**18.77%**	52.17%	**7.97%**	39.95%	**41.00%**
OpenOffice	48.26%	40.42%	**19.38%**	16.95%	**184.46%**	17.56%	**174.83%**
Mozilla	55.92%	32.14%	**73.98%**	10.63%	**426.08%**	16.56%	**237.68%**
Netbeans	70.73%	42.40%	**66.81%**	48.53%	**45.74%**	31.22%	**126.55%**
Eclipse	79.89%	38.26%	**108.79%**	30.45%	**162.44%**	13.73%	**481.86%**
Average	62.22%	40.13%	**57.55%**	31.75%	**165.38%**	23.80%	**212.39%**
2*Projects	Recall@10						
	DevRec	Bugzie	Impro.Bugzie	DREX	Impro.DREX	NonTraining	Impro.Non
GCC	70.72%	67.24%	**5.17%**	64.94%	**8.90%**	51.88%	**36.31%**
OpenOffice	60.63%	53.64%	**13.05%**	25.11%	**141.48%**	26.67%	**127.33%**
Mozilla	67.55%	44.16%	**52.96%**	28.54%	**136.67%**	22.41%	**201.43%**
Netbeans	80.21%	54.48%	**47.24%**	56.29%	**42.50%**	36.95%	**117.08%**
Eclipse	89.24%	49.98%	**78.55%**	41.08%	**117.28%**	19.49%	**357.88%**
Average	73.67%	53.90%	**39.39%**	43.19%	**89.36%**	31.48%	**168.01%**
2*Projects	Precision@5						
	DevRec	Bugzie	Impro.Bugzie	DREX	Impro.DREX	NonTraining	Impro.Non
GCC	24.53%	20.77%	**15.33%**	23.14%	**6.01%**	17.69%	**38.67%**
OpenOffice	21.00%	17.40%	**17.14%**	8.52%	**146.48%**	8.00%	**162.50%**
Mozilla	24.71%	14.56%	**41.08%**	6.64%	**272.14%**	7.61%	**224.70%**
Netbeans	31.96%	19.53%	**38.89%**	22.87%	**39.75%**	15.20%	**110.26%**
Eclipse	25.09%	12.12%	**51.69%**	10.38%	**141.71%**	4.43%	**466.37%**
Average	25.46%	16.88%	**32.83%**	14.31%	**121.22%**	10.59%	**200.50%**
2*Projects	Precision@10						
	DevRec	Bugzie	Impro.Bugzie	DREX	Impro.DREX	NonTraining	Impro.Non
GCC	15.99%	15.14%	**5.32%**	14.93%	**7.10%**	11.61%	**37.73%**
OpenOffice	13.31%	11.92%	**10.44%**	6.18%	**115.37%**	5.87%	**126.75%**
Mozilla	15.46%	10.43%	**32.54%**	7.23%	**113.83%**	5.31%	**191.15%**
Netbeans	18.59%	12.68%	**31.79%**	13.55%	**37.20%**	9.03%	**105.87%**
Eclipse	14.31%	8.16%	**42.98%**	6.92%	**106.79%**	3.18%	**350.00%**
Average	15.53%	11.67%	**24.62%**	9.76%	**76.06%**	7.00%	**162.30%**

Notice that the values of precision might seem low. However, notice that the number of bug resolvers per bug report is low. Thus, the optimal precision@k value is low. For example, in Eclipse, the average number of bug resolvers per bug report is 1.88. If we recommend top 10 developers, the best precision@10 would be around 0.188. The precision@10 of DevRec for the Eclipse dataset is 0.1431, which is close to the optimal value. From Table VI, the improvement of our method over Bugzie is substantial and statistically significant. DevRec outperforms Bugzie by 32.83% and 24.61% for average precision@5 and precision@10, respectively. From Table VI, the improvement of our method over DREX is also substantial and statistically significant. DevRec outperforms DREX by 121.22% and 76.06% for average precision@5 and precision@10, respectively. From Table VI, the improvement of our method over NonTraining is also substantial and statistically significant. DevRec outperforms NonTraining by 200.50% and 162.30% for average precision@5 and precision@10, respectively. The results show that clearly DevRec outperforms Bugzie, DREX, and NonTraining, which are the state-of-the-art techniques.

4.3. RQ2: performance of BR-based and D-based components

Table VII presents the recall@5, recall@10, precision@5, and precision@10 scores of DevRec compared with those of BR-based and D-based components. DevRec outperforms the BR-based component by 5.17% and 4.83% for average recall@5 and recall@10, respectively. DevRec outperforms the D-based component by 6.07% and 2.99% for average recall@5 and recall@10,

DOI: 10.1002/smr

Table VII. Recall@5, Recall@10, Precision@5, and Precision@10 of DevRec compared with BR-based (BR.) component and D-based (D.) component, and the improvement of DevRec over BR-based (BR.) component (Impro.BR.) and over D-based (D.) component (Impro.D.). Statistically significant improvements are highlighted in bold.

2*Projects	Recall@5				
	DevRec	BR-Based.	Impro.BR.	D-Based.	Impro.D.
GCC	56.33%	48.20%	**16.87%**	55.24%	**1.97%**
OpenOffice	48.26%	46.70%	**3.34%**	47.83%	0.90%
Mozilla	55.92%	54.87%	**1.91%**	50.53%	**10.67%**
Netbeans	70.73%	69.74%	**2.24%**	63.83%	**11.70%**
Eclipse	79.89%	78.73%	**1.47%**	76.02%	**5.09%**
Average	62.22%	59.65%	**5.17%**	58.69%	**6.07%**
2*Projects	Recall@10				
	DevRec	BR-Based.	Impro.BR.	D-Based.	Impro.D.
GCC	70.72%	64.90%	**8.97%**	69.52%	**1.73%**
OpenOffice	60.63%	57.28%	**5.85%**	60.59%	0.07%
Mozilla	67.55%	65.67%	**2.86%**	63.13%	**7.00%**
Netbeans	80.21%	78.73%	**2.65%**	77.94%	**3.70%**
Eclipse	89.24%	85.95%	**3.93%**	87.08%	**2.48%**
Average	73.67%	70.51%	**4.83%**	71.65%	**2.99%**
2*Projects	Precision@5				
	DevRec	BR-Based.	Impro.BR.	D-Based.	Impro.D.
GCC	24.53%	24.17%	1.49%	20.93%	**17.20%**
OpenOffice	21.00%	20.28%	**3.55%**	20.80%	0.96%
Mozilla	24.71%	24.14%	**2.36%**	22.29%	**10.86%**
Netbeans	31.96%	31.45%	1.62%	29.39%	**8.74%**
Eclipse	25.09%	24.60%	**1.99%**	23.65%	**6.09%**
Average	25.46%	24.93%	**2.00%**	23.41%	**8.77%**
2*Projects	Precision@10				
	DevRec	BR-Based.	Impro.BR.	D-Based.	Impro.D.
GCC	15.99%	15.63%	2.30%	14.54%	**9.07%**
OpenOffice	13.31%	12.78%	**4.15%**	13.27%	0.30%
Mozilla	15.46%	15.08%	**2.52%**	14.60%	**5.56%**
Netbeans	18.59%	18.17%	2.31%	18.10%	**2.64%**
Eclipse	14.31%	13.77%	**3.92%**	14.01%	**2.10%**
Average	15.53%	15.09%	**3.04%**	14.90%	**3.93%**

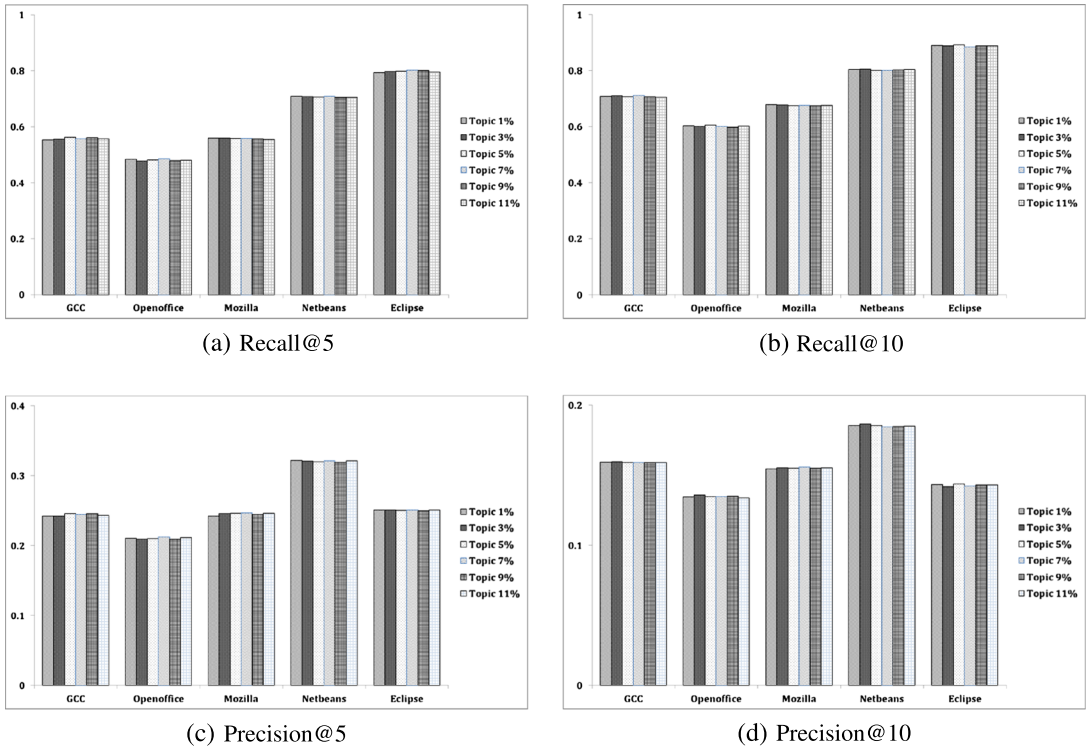
respectively. DevRec outperforms the BR-based component by 2.00% and 3.04% for average precision@5 and precision@10, respectively. DevRec outperforms the D-based component by 8.77% and 3.93% for average precision@5 and precision@10, respectively. The results show that it is beneficial to combine the BR-based and D-based components. Statistical tests show that DevRec significantly improves recall@5 (recall@10) scores of BR-based component on all of the five projects and four out of the five projects, respectively. DevRec also significantly improves recall@5 and recall@10 scores of the D-based component on four out of the five projects. Furthermore, DevRec significantly improves the precision@5 and precision@10 scores of BR-based component on three out of the five projects and those of D-based component on four out of the five projects.

4.4. RQ3: effect of varying the number of topics

Figure 3 presents the recall@5, recall@10, precision@5, and precision@10 of DevRec for various numbers of topics for GCC, OpenOffice, Mozilla, Netbeans, and Eclipse, respectively. It can be seen that the performance of DevRec over the different numbers of topics only varies slightly. Thus, DevRec is robust to different numbers of topics, within a reasonable range.

To answer why DevRec is stable for various numbers of topics, we manually check the term distributions under each topic when we vary the number of topics from 1% to 11% of the number of distinct terms in our training data. We find that even when we choose the number of topics as 1% of

DOI: 10.1002/smr



(a) Recall@5 (b) Recall@10

(c) Precision@5 (d) Precision@10

Figure 3. Recall@5, Recall@10, Precision@5, and Precision@10 for different numbers of topics (1–11% of the number of distinct terms in the training data).

the number of distinct terms, the terms under many topics show relatively good semantic relationships. Table VIII presents the top 10 terms under several topics learned from our Eclipse dataset when we set the number of topic as 1% of the number of distinct terms. Notice that these terms show good semantic relationships. For example, the terms in Topic 1 describe various components of the Eclipse user interface (e.g., button, table, documentation, and image); the terms in Topic 2 describe various operations (e.g., click, import, and export); the terms in Topic 3 describe various programming languages (e.g., C, Java, HTML, and Javascript), and the terms in Topic 4 are related to debugging (e.g., stack, trace, exception). Our findings suggest that the number of topics does not matter as long as the topics capture a reasonable abstraction of a set of terms.

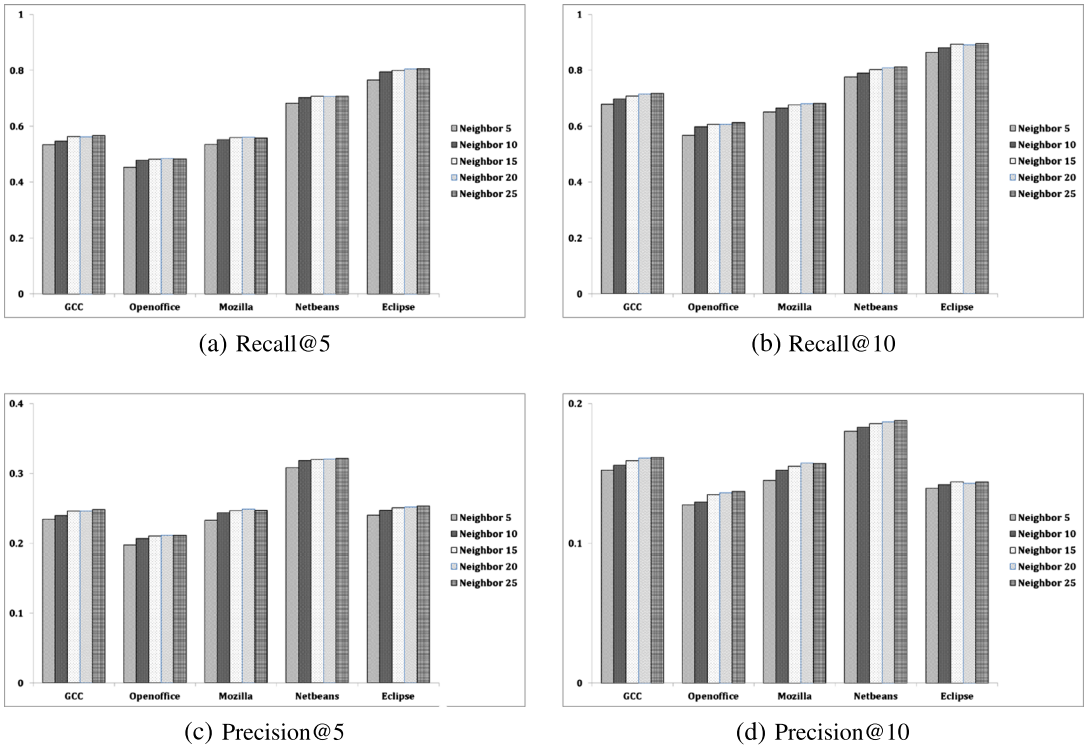
4.5. RQ4: effect of varying the number of neighbors

We vary the number of neighbors k from 5 to 25 to investigate the sensitivity of DevRec on this parameter. Figure 4 presents the experiment results for GCC, OpenOffice, Mozilla, Netbeans, and Eclipse. The results show that the performance of DevRec is stable for various numbers of neighbors.

Table VIII. Top 10 terms under four topics in Eclipse when the topic number is 1% of the number of distinct terms.

Topic 1	Topic 2	Topic 3	Topic 4
Button	Click	Java	Exception
UI	Save	C	Stack
Explorer	Import	J2EE	Trace
Table	Export	Code	Debug
URL	Edit	Language	Org
Interface	Insert	HTML	Eclipse
Image	Open	JavaScript	Reproduce
Document	New	JSON	Step
Package	Print	Framework	Error

DOI: 10.1002/smr



(a) Recall@5 (b) Recall@10

(c) Precision@5 (d) Precision@10

Figure 4. Recall@5, Recall@10, Precision@5, and Precision@10 for different numbers of neighbors (from 5 to 25).

To investigate why DevRec is stable for various numbers of neighbors, we check the list of top k neighbors of several bug reports in our dataset for k = 5–25. We find that in many cases, the set of main resolvers involved in the top 5 most similar bug reports is similar to the set of main resolvers involved in the top 25 most similar bug reports. Our findings suggest that as long as the ground truth resolvers are also actively involved in at least k similar bug reports before, the exact value of k does not matter much.

4.6. RQ5: effect of varying the number of terms

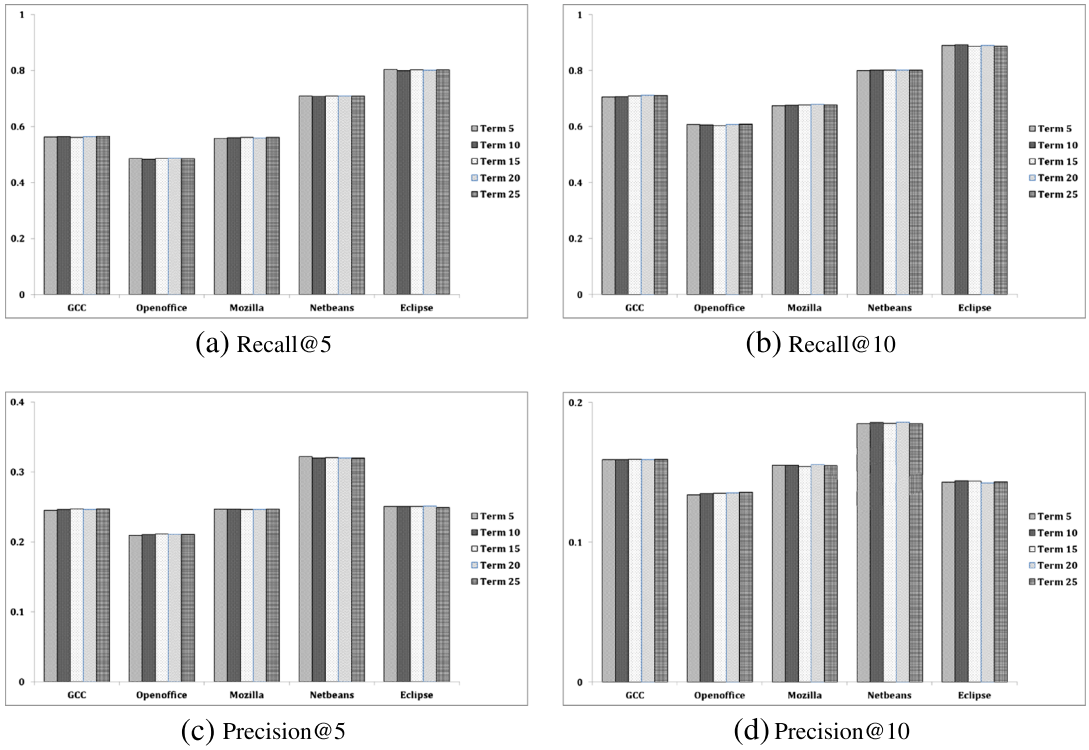
Because our DevRec uses terms affinity score as shown in Section 3.2.1, we set the descriptive term number for each developer from 5 to 25 to validate the stability of DevRec. Figure 5 presents the experiment results for GCC, OpenOffice, Mozilla, Netbeans, and Eclipse. The results show that the performance of DevRec is stable for various numbers of terms, and the differences between different numbers of terms are slight.

One rationale why the performance of DevRec remains stable for various numbers of terms is due to the fact that DevRec combines different analyses (i.e., BR-Based and D-Based analyses) and the number of terms is only one parameter that affects a small part of the D-Based analysis (i.e., term affinity score). Also, the weights of various scores in DevRec are fine-tuned based on its performance on a training data. Scores that lead to poorer performance are given lower weights. This might offset any adverse effect that is caused by varying the number of terms.

4.7. Discussion

4.7.1. Effectiveness of different components. In this paper, we automatically identify good γ weights for DevRec following Algorithm 1. The γ weights would be optimized (and thus are different) for different datasets and training frames in our longitudinal data setup. Table IX presents the average weights of the term, topic, product, and component affinity scores in the D-based component and

DOI: 10.1002/smr



(a) Recall@5 (b) Recall@10

(c) Precision@5 (d) Precision@10

Figure 5. Recall@5, Recall@10, Precision@5, and Precision@10 for different numbers of terms (from 5 to 25).

Table IX. Average weights for the term, topic, product, and component affinity scores in D-based components and the BR-based components.

Projects	Term	Topic	Product	Component	BR-based
GCC	0.3934	0.0100	0.5150	0.2651	0.7392
OpenOffice	0.4539	0.0100	0.3714	0.3695	0.6775
Mozilla	0.1268	0.0100	0.4862	0.4697	0.7323
Netbeans	0.2779	0.0100	0.5013	0.3966	0.6086
Eclipse	0.3815	0.1000	0.5150	0.6836	0.5025

the average weight of the BR-based component for each dataset. We compute the average weights across the 10 folds of the longitudinal data setup. We note that the weights for different datasets are different. For example, for GCC, BR-based component has the highest weight; and for Eclipse, the component affinity score has the highest weight.

4.7.2. Mean average precision. Considering our developer recommendation is based on the ranking scores of each developers; we also investigate the performance of our DevRec with one more evaluation metric, mean average precision (MAP) [23], which is a single-figure evaluation metric to measure the quality of information retrieval. In our case, we consider a bug report as a 'query' and the developers who join the bug resolution process as the 'relevant results (i.e., instances)'. In information retrieval, a query q could have multiple relevant results, thus the average precision (AveP) for the query q is

$$AveP_q = \frac{\sum_{k=1}^{M} P(k) \times rel(k)}{Number\ of\ relevant\ instances} \tag{14}$$

DOI: 10.1002/smr

In the aforementioned equation, k is the rank, M is the number of instances (i.e., total number of developers) retrieved, and $rel(k)$ indicates whether the instance (aka. developer) in the rank k is relevant or not. $P(k)$ is the precision at the given cut-off rank k and is computed as follows:

$$p(k) = \frac{Number\ of\ relevant\ instances\ in\ top\ k\ position}{k} \tag{15}$$

Mean average precision for a set of queries (aka. bug reports) is the mean of the average precision values for all queries. Considering a total of Q queries, MAP is computed as follows:

$$MAP = \frac{\sum_{q=1}^{Q} AveP(q)}{Q} \tag{16}$$

In developer recommendation, a bug can be resolved by multiple developers. We use MAP to measure the average performance of DevRec for recommending the developers. The higher the MAP value, the better the performance of our DevRec.

Table X presents the MAP scores of DevRec compared with Bugzie and DREX and the improvement of DevRec over Bugzie (Impro.B.) and over DREX (Impro.D.), respectively. The statistically significant improvements are highlighted in bold. The MAP scores of DevRec vary from 0.4041 to 0.6377. From Table X, the improvements of our method over Bugzie, DREX, and NonTraining are substantial and statistically significant. On average across five projects, DevRec improves the MAP scores of Bugzie, DREX, and NonTraining by 61.50%, 205.86%, and 158.03%, respectively.

4.7.3. Qualitative analysis. Here, we want to perform a qualitative analysis on why our new types of features (i.e., topics, products, and components) worked in DevRec and why we need to combine BR-based component and D-based component.

4.7.3.1. Features. Figures 6, 7, and 8 present three bug reports from Eclipse. These three bug reports share some commonalities: they are in the same product JDT and component CORE, and some of the bug resolvers are the same. For example, bug reports #102780 and #230944 both have Olivier Thomann, Frederic Fusier, Eric Jodet, and Kent Johnson as resolvers. Also, these three bug reports all describe bugs about graphical user interface problems in Eclipse, that is, they share some latent commonalities. However, the textual description of these three bug reports are different, thus, if we use Bugzie or DREX, which are only based on the terms in bug reports, and recommend resolvers for bug report #230944, Bugzie or DREX cannot effectively leverage information from bug reports #102780 and #148859, which causes inaccurate recommendation. Our DevRec uses topic model to detect the latent commonalities behind the textual description of bugs and also considers the product and component information, which results in better recommendations.

4.7.3.2. Combination. In our dataset, when we recommend resolvers for bug report #230944 just using BR-based component, the top 5 developers are Eric Jodet, Frederic Fusier, Olivier Thomann,

Table X. MAP scores of DevRec compared with Bugzie and DREX and the improvement of DevRec over Bugzie (Impro.B.) and over DREX (Impro.D.). The last row shows the average Recall@5, Recall@10, Precision@5, and Precision@10 scores of DevRec, Bugzie, and DREX and the average improvement. Statistically significant improvements are highlighted in bold.

Projects	DevRec	Bugzie	Impro.Bugzie	DREX	Impro.DREX	NonTraining	Impro.Non
GCC	0.4488	0.3561	**26.03%**	0.3509	**27.90%**	0.3132	**43.30%**
OpenOffice	0.4041	0.3300	**22.45%**	0.0941	**329.44%**	0.1457	**177.35%**
Mozilla	0.4616	0.2640	**74.85%**	0.0432	**968.52%**	0.1372	**236.44%**
Netbeans	0.6002	0.3357	**78.79%**	0.2878	**108.55%**	0.2779	**115.98%**
Eclipse	0.6377	0.2946	**116.46%**	0.0585	**990.09%**	0.1152	**453.56%**
Average	0.5105	0.3161	**61.50%**	0.1669	**205.86%**	0.1978	**158.03%**

DOI: 10.1002/smr

Summary: CodeFormatter does not format Javadoc
Description: We are using the code formatter (org.eclipse.jdt.core.formatter.CodeFormatter) from within a code generating plugin and everything works fine except it does NOT format comments and Javadoc according to the preferences setting in " Java Code Style Code Formatter" (Edit profile, tab "comments").
Product: JDT
Component: CORE
Resolvers: Frederic Fusier, Olivier Thomann, Benno Baumgartner, Jerome Lanneluc, Eric Jodet, Kent Johnson, Dani Megert

Figure 6. Bug report #102780 of Eclipse.

Summary: Package Explorer only shows default package after import
Description: To reproduce this, import an ear with a J2EE Application Client module in it (with source). The import works fine by putting the Java files at the right location.
However, in the J2EE and Java perspectives, the Project Explorer only shows the default package although there are more packages present in the project (visible in Resource view). You can take a look at my screen captures.
There are currently no workaround other than restarting the workbench, then the missing packages show up.
Product: JDT
Component: CORE
Resolvers: Frederic Fusier, Michael D. Elder, Tim deBoer, Martin Aeschlimann, Yen Lu, Jerome Lanneluc, Philipe Mulet, Angel Vera, Olivier Thomann

Figure 7. Bug report #148859 of Eclipse.

Summary: Formatter does not respect /*-
Description: Steps To Reproduce:
1. Create a block comment with whitespace formatting that starts with /*-
2. Run the Eclipse [built-in] formatter
3. Whitespace will be stripped out, rendering comments that contain lists or tree diagrams unreadable
...
Product: JDT
Component: CORE
Resolvers: Olivier Thomann, Frederic Fusier, Eric Jodet, Steven Schwab, Kent Johnson

Figure 8. Bug report #230944 of Eclipse.

Philipe Mulet, and Markus Keller, that is, Recall@5 = 3/5=0.6. And when we just use D-based component, the top 5 developers are Frederic Fusier, Olivier Thomann, Kent Johnson, Jerome Lanneluc, and Michael D. Elder, that is, Recall@5 = 3/5=0.6. DevRec combines BR-based and D-based components and produces the following top 5 developers: Frederic Fusier, Olivier Thomann, Eric Jodet, Kent Johnson, and Jerome Lanneluc, that is, Recall@5 = 4/5 =0.8. Thus, the combination of these two components utilizes the advantages of each component and achieves a better performance.

4.8. Threats to validity

Threats to internal validity relates to errors in our experiments. We have double checked our datasets and experiments, still there could be errors that we did not notice. Notice that although a bug triager does not directly contribute knowledge for bug fixing, they also play an important role for bug resolution. For example, a bug triager may contribute his/her knowledge to find a suitable fixer, or after he/she marks duplicate or blocking bug reports, the other developers could find more information from other related bug reports [14, 15]. Moreover, a bug triager could also be a bug contributor or a bug fixer too (Figure 1). In this work, we assume bug triagers, bug contributors, and bug fixers are equally important and we do not differentiate them in our recommendation process. In the future, we plan to develop a new technique that would recommend developers along with their likely roles.

DOI: 10.1002/smr

Threats to external validity relates to the generalizability of our results. We have analyzed 107,875 bug reports from five large software systems. Also, in this paper, we recommend three types of bug resolvers: bug fixer, bug contributor, and bug triager.

Threats to construct validity refers to the suitability of our evaluation measures. We use recall@k and precision@k, which are also used by past studies to evaluate the effectiveness of developer recommendation [3, 4], and also many other software engineering studies [19]. Thus, we believe there is little threat to construct validity.

5. RELATED WORK

In this section, we briefly review several studies related to our paper. To our best knowledge, DREX [3], which will be introduced in Section 5.1, is the most related work to our paper. We highlight some of bug triaging studies especially Bugzie [5] in Section 5.2. In Section 5.3, we present other related studies on bug report management.

5.1. DREX

The most related work to our paper is DREX [3], which recommends developers for bug resolution. The main idea behind DREX is that the k-nearest neighbors of a bug report can help to recommend developers for the bug report. In DREX, for a new bug report, it first finds the new bug report's k-nearest neighbors based on the textual description of historical bug reports. And based on the neighbors' information, it uses simple frequency counting, and some other social network analysis, such as degree, in-degree, out-degree, betweeness, closeness, and PageRank to recommend potential developers for bug resolution.

Our approach DevRec is different from DREX in several ways: (1) We perform not only BR-Based analysis but also D-Based analysis; (2) For the BR-Based analysis, we make use of multiple features that are not only terms but also topics, product, and component; (3) Also, for the BR-Based analysis, we make use of the state-of-the-art work on multi-label classification namely ML-KNN; and (4) We consider a larger dataset consisting of more than 100,000 bug reports from five projects to evaluate our approach and compares it with DREX and Bugzie. We show that our approach outperforms DREX by a substantial margin.

5.2. Bug assignment

Bug assignment refers to the task of finding the most appropriate developer to fix the bug [5, 20, 24, 25, 26, 27]. From the machine learning perspective, the problem can be mapped to *single-label* learning problem, where each bug report is assigned to only one developer. The title, description, and summary fields of bug reports are extracted to train classifiers. Anvik *et al.* and Cubranic *et al.* use machine learning technologies such as Naive Bayes, support vector machine, and C4.8 for bug assignment [24, 25]. Tamrawi *et al.* propose a method called Bugzie, which is based on the fuzzy set theory [5]. It caches the most descriptive terms that characterize each developer and uses them to measure the suitability of a developer to a new bug report. Baysal *et al.* represent the expertise of a developer by analyzing the history of bugs previously resolved by the developer and apply vector space model to recommend experts for fixing bugs [28]. Guo *et al.* perform an empirical study of bug reassignment phenomenon, and they find that bug reassignment could help to determine the best person to fix a bug [29]. Jeong *et al.* investigate bug reassignment in Eclipse and Mozilla and propose a graph model based on Markov chain to improve bug triaging performance [26]. Bhattacharya *et al.* reduce tossing path lengths and improve the accuracy of the approach by Jeong *et al.* further [20]. Lin *et al.* study bug assignment using bug reports that are written in Chinese, and they find that text data are useful to assign bug reports [30]. Yang *et al.* propose an approach that uses topic model to identify past historical bug reports that are similar to the new bug report and recommends fixers of the past historical bug reports to new bug report [31].

In this work, different from the aforementioned studies, we consider bug fixing as a collaborative effort. The aforementioned studies focused on recommending the developer that is assigned to fix

DOI: 10.1002/smr

the bug (aka. bug fixer). However, more than one people often work together to resolve a bug. In this work, we would like to recommend everyone that contribute to the bug resolution process (the bug resolvers). Notice both our work and the aforementioned studies would also recommend a list of people (i.e.,top-n people). However, among these n people, the aforementioned studies only have at most one developer who is considered to have fixed the bug, while in our work, we could have multiple (one or more) people who are involved in the bug resolution process.

Anvik and Murphy point out that bug assignment is one of the activities of a bug triager [27]. Moreover, a bug triager would also make other decisions, for example, mark duplicate bug reports, categorize bug reports by components or products, and check for reproducibility and validity [27]. From this perspective, both bug assignment and developer recommendation for bug resolution are in the bug triaging process, and they have different targets: bug assignment aims to find the most appropriate developer to fix the bug and developer recommendation aims to find developers who could help in the bug resolution process in various ways.

Recently, other information sources aside from bug reports (e.g., commits and source code) have been used to recommend appropriate developers. Matter *et al.* propose a tool named Develect that models developer expertise based on the vocabulary used in their source code contributions and recommends bug fixers based on the similarity between the words in a bug report and those in the vocabulary of the fixers [32]. Kagdi *et al.* use feature location technology to find program units (e.g., files or classes) that are related to a change request (i.e., bug report or feature request) and then mine commits in version control repositories that modify those program units to recommend appropriate developers [33]. Linares-Vásquez *et al.* propose a method to recommend developers by also employing feature location to find relevant files; however, rather than analyzing version control repositories, they recommend developers by looking to the list of authors at the header comments of the relevant files [34]. Kevic *et al.* recommend developers to a bug report by finding other similar bug reports, recovering the files that are changed to fix the previous similar bug reports, and analyzing developers that changed those files [35]. Shokripour *et al.* propose a two-phased location-based approach to assign bug reports to developers [36]. In their approach, they first determine bug location information by utilizing a noun extraction process on multiple information sources, and then a simple term weighting scheme is used to recommend a list of potential bug fixers. Different from the aforementioned studies, in this work, we only analyze information available in bug reports. Hossen *et al.* propose iMacPro, which considers the authors and maintenances in source code and commit logs to recommend developers for a change request [37]. Hossen *et al.* propose iMacPro that amalgates source code authors, maintainers, and change proneness to assign a developer to a change request [37]. iMacPro first identifies source code files that are relevant to a change request using Latent Semantic Indexing. Next, the relevant files are ranked based on their change proneness. Finally, the authors and maintainers of these files are ranked and recommended to the change request. Often there is an issue in linking bug reports to commits that fix the bug [38, 39]. Still, it would be interesting to combine our approach with the aforementioned approaches when the links between bug reports and relevant commits are well maintained.

5.3. Other studies on bug report management

A number of studies have been proposed to automatically detect duplicated bug reports [40–43]. Hiew proposes the problem of duplicated bug report detection [44]. For a new bug report, Hiew computes the distance between the new bug report and existing bug reports. Runeson *et al.* measure the similarity of two bug reports by using various similarity metrics based on the terms appearing in the bug reports [40]. Wang *et al.* propose a duplicate bug report detection approach that considers execution traces in addition to terms appearing in the bug reports [41]. Our work is orthogonal to the aforementioned studies. Sun *et al.* propose a discriminative model-based approach, which leverages support vector machine, to identify bug reports that are duplicate of one another [43]. In a latter work, Sun *et al.* extend BM25 for duplicate bug report detection [42]. Our work is orthogonal to the aforementioned studies; also, the set of bug reports that we experimented on is non-duplicate bug reports – for each set of bug reports whose members are duplicate of one another, we only keep the master report.

DOI: 10.1002/smr

A number of studies have been proposed to predict the severity labels of bug reports [45, 46]. Severis, proposed by Menzies and Marcus, performs multi-class classification to predict the five severity labels of bug reports in NASA [45]. Their work is extended by Lamkanfi *et al.*, which predict two severity labels (severe versus not severe) of bug reports in a number of Bugzilla bug tracking systems of open source programs [47]. In a latter work, Lamkanfi *et al.* further investigate the performance of a number of classification methods to predict the severity of bug reports [48]. Tian *et al.* predict fine-grained severity labels by using extended BM25 and nearest neighbor classification [46]. Our work complements the aforementioned studies; after the severity of a bug report could be determined, our approach could be employed to recommend a suitable developer to fix the bug.

Huang *et al.* propose a machine learning approach that predicts the categories of bug reports based on their impact; the category labels include: capability, security, performance, reliability, requirement, and usability [49]. Gegick *et al.* perform text mining to recover security bug reports [50].

6. CONCLUSION AND FUTURE WORK

In this paper, we propose a new method DevRec to automatically recommend developers for bug resolution. We consider two kinds of analysis: bug report based analysis (BR-Based analysis) and developer based analysis (D-Based analysis). DevRec takes the advantage of both BR-Based and D-Based analyses and composes them together. The experiment results show that, compared with the other state-of-the-art approaches, DevRec achieves the best performance. The results show that DevRec on average improves recall@5 and recall@10 scores of Bugzie by 57.55% and 39.39%, outperforms DREX by 165.38% and 89.36%, and outperforms NonTraining by 212.39% and 168.01%, respectively. Moreover, we evaluate the stableness of DevRec with different parameters, and the results show that DevRec is a stable approach.

In the future, we plan to improve the effectiveness of DevRec further (for example, integrate the LDA-GA method proposed by Panichella *et al.* [22] or employ other text mining solutions, e.g., [51]). We also plan to experiment with even more bug reports from more projects and perform a case study by surveying developers of both open source and industrial projects to study the viability of our proposed approach.

ACKNOWLEDGEMENTS

This research was supported by the National Basic Research Program of China (the 973 Program) under grant 2015CB352201, and National Key Technology R&D Program of the Ministry of Science and Technology of China under grant 2014BAH24F02. We would like to thank Jafar M. Al-Kofahi for providing information about Bugzie. The datasets and source code of DevRec can be downloaded from https://www.dropbox.com/s/43ohauo0dfwufvx/DevRec.7z?m.

REFERENCES

1. Bertram D, Voida A, Greenberg S, Walker R. Communication, collaboration, and bugs: the social nature of issue tracking in small, collocated teams, In CSCW 2010.
2. Anvik J, Hiew L, Murphy G. Coping with an open bug repository, In ETX 2005.
3. Wu W, Zhang W, Yang Y, Wang Q. DREX: Developer recommendation with k-nearest-neighbor search and expertise ranking, In APSEC 2011.
4. Xie X, Zhang W, Yang Y, Wang Q. Dretom: Developer recommendation based on topic models for bug resolution, In PROMISE 2012.
5. Tamrawi A, Nguyen T, Al-Kofahi J, Nguyen T. Fuzzy set and cache-based approach for bug triaging, In ESEC/FSE 2011.
6. Gcc bug tracking system, http://gcc.gnu.org/bugzilla/.
7. Openoffice bug tracking system, https://issues.apache.org/ooo/.
8. Mozilla bug tracking system.
9. Netbeans bug tracking system, http://netbeans.org/bugzilla/.
10. Eclipse bug tracking system, https://bugs.eclipse.org/bugs/.
11. Xia X, Lo D, Wang X, Zhou B. Accurate developer recommendation for bug resolution, In *Reverse engineering* (*WCRE*), *2013 20th working conference on*. IEEE, 2013; 72–81.
12. Zhang M, Zhou Z. Ml-knn: A lazy learning approach to multi-label learning, Pattern Recognition, 2007.
13. Blei D, Ng A, Jordan M. Latent dirichlet allocation, *Journal of Machine Learning Research* 2003.

DOI: 10.1002/smr

14. Bettenburg N, Premraj R, Zimmermann T, Kim S. Duplicate bug reports considered harmfu really? In *Software maintenance*, *2008. ICSM 2008. IEEE international conference on*. IEEE, 2008; 337–345.
15. Valdivia Garcia H, Shihab E. Characterizing and predicting blocking bugs in open source projects, In Proceedings of the 11th working conference on mining software repositories.ACM, 2014; 72–81.
16. H. J., Kamber M. Data Mining: Concepts and Techniques, 2006.
17. Manning C, Raghavan P, Schutze H. Introduction to Information Retrieval, 2008.
18. Tsoumakas G, Katakis I. Multi-label classification: An overview, IJDWM, 2007.
19. Xia X, Lo D, Wang X, Zhou B. Tag recommendation in software information sites, In MSR 2013.
20. Bhattacharya P, Neamtiu I. Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging, In ICSM 2010.
21. Wilcoxon F. Individual comparisons by ranking methods, *Biometrics Bulletin* 1945; 80–83,
22. Panichella A, Dit B, Oliveto R, Penta MD, Poshyvanyk D, Lucia AD. How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms, In ICSE 2013.
23. Baeza-Yates R, Ribeiro-Neto B, et al. Modern Information Retrieval ACM press: *New York*, 1999, vol. **463**.
24. Anvik J, Hiew L, Murphy G. Who should fix this bug? In ICSE 2006.
25. Čubranić D. Automatic bug triage using text categorization, In SEKE.
26. Jeong G, Kim S, Zimmermann T. Improving bug triage with bug tossing graphs, In ESEC/FSE 2009.
27. Anvik J, Murphy GC. Reducing the effort of bug report triage: Recommenders for development-oriented decisions, TOSEM, 2011.
28. Baysal O, Godfrey MW, Cohen R. A bug you like: A framework for automated assignment of bugs, In *Program comprehension*, *2009. ICPC'09. IEEE 17th international conference on*. IEEE, 2009; 297–298.
29. Guo PJ, Zimmermann T, Nagappan N, Murphy B. Not my bug! and other reasons for software bug report reassignments, In CSCW 2011.
30. Lin Z, Shu F, Yang Y, Hu C, Wang Q. An empirical study on bug assignment automation using chinese bug data, In *Empirical software engineering and measurement*, *2009. ESEM 2009. 3rd international symposium on*. IEEE, 2009; 451–455.
31. Yang G, Zhang T, Lee B. Towards semi-automatic bug triage and severity prediction based on topic model and multi-feature of bug reports, In *Computer software and applications conference* (*COMPSAC*), *2014 IEEE 38th annual*. IEEE, 2014; 97–106.
32. Matter D, Kuhn A, Nierstrasz O. Assigning bug reports using a vocabulary-based expertise model of developers, In MSR 2009.
33. Kagdi H, Gethers M, Poshyvanyk D, Hammad M. Assigning Change Requests to Software Developers. Journal of Software: Evolution and Process 2012.
34. Linares-Vásquez M, Hossen K, Dang H, Kagdi H, Gethers M, Poshyvanyk D. Triaging incoming change requests: Bug or commit history, or code authorship? In ICSM, 2012.
35. Kevic K, Müller SC, Fritz T, Gall HC. Collaborative bug triaging using textual similarities and change set analysis, In CHASE 2013.
36. Shokripour R, Anvik J, Kasirun ZM, Zamani S. Why so complicated? simple term filtering and weighting for location-based bug report assignment recommendation, In Proceedings of the 10th working conference on mining software repositories.IEEE Press, 2013; 2–11.
37. Hossen K, Kagdi HH, Poshyvanyk D. Amalgamating source code authors, maintainers, and change proneness to triage change requests. In ICPC, 2014; 130–141.
38. Tian Y, Lawall JL, Lo D. Identifying linux bug fixing patches, In ICSE, 2012.
39. Bissyandé TF, Thung F, Wang S, Lo D, Jiang L, Réveillère L. Empirical evaluation of bug linking, In CSMR, 2013.
40. Runeson P, Alexandersson M, Nyholm O. Detection of duplicate defect reports using natural language processing, In ICSE, 2007.
41. Wang X, Zhang L, Xie T, Anvik J, Sun J. An Approach to Detecting Duplicate Bug Reports using Natural Language and Execution Information, In ICSE, 2008.
42. Sun C, Lo D, Khoo S-C, Jiang J. Towards more accurate retrieval of duplicate bug reports, In ASE, 2011.
43. Sun C, Lo D, Wang X, Jiang J, Khoo S-C. A discriminative model approach for accurate duplicate bug report retrieval, In ICSE, 2010.
44. Hiew L. Assisted detection of duplicate bug reports, Ph.D. dissertation, The University Of British Columbia, 2006.
45. Menzies T, Marcus A. Automated severity assessment of software defect reports, In ICSM, 2008.
46. Tian Y, Lo D, Sun C. Information retrieval based nearest neighbor classification for fine-grained bug severity prediction, In WCRE 2012.
47. Lamkanfi A, Demeyer S, Giger E, Goethals B. Predicting the severity of a reported bug, In MSR, 2010.
48. Lamkanfi A, Demeyer S, Soetens Q, Verdonck T. Comparing mining algorithms for predicting the severity of a reported bug, In CSMR, 2011.
49. Huang L, Ng V, Persing I, Geng R, Bai X, Tian J. AutoODC: Automated generation of orthogonal defect classifications, In ASE, 2011.
50. Gegick M, Rotella P, Xie T. Identifying security bug reports via text mining: An industrial case study, In MSR, 2010.
51. Wang X, Lo D, Jiang J, Zhang L, Mei H. Extracting paraphrases of technical terms from noisy parallel software corpora, In ACL/IJCNLP, 2009.

DOI: 10.1002/smr