10-2013

Accurate Developer Recommendation for Bug Resolution

Xin XIA

David LO
Singapore Management University, davidlo@smu.edu.sg

Xinyu WANG

Bo ZHOU

Accurate Developer Recommendation for Bug Resolution

Xin Xia*‡, David Lo†, Xinyu Wang*, and Bo Zhou*§
*College of Computer Science and Technology, Zhejiang University, China
†School of Information Systems, Singapore Management University, Singapore
xxkidd@zju.edu.cn, davidlo@smu.edu.sg, {wangxinyu, bzhou}@zju.edu.cn

***Abstract*—Bug resolution refers to the activity that developers perform to diagnose, fix, test, and document bugs during software development and maintenance. It is a collaborative activity among developers who contribute their knowledge, ideas, and expertise to resolve bugs. 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 which performs two kinds of analysis: bug reports based analysis (BR-Based analysis), and developer based analysis (D-Based analysis). In the BR-Based analysis, we characterize a new bug report based on past bug reports that are similar to it. Appropriate developers of the new bug report are found by investigating the developers of similar bug reports appearing in the past. In the D-Based analysis, we compute the affinity of each developer to a bug report based on the characteristics of bug reports that have been fixed by the developer before. This affinity is then used to find a set of developers that are "close" to a new bug report.

We evaluate our solution on 5 large bug report datasets including GCC, 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. We also compare *DevRec* with other state-of-art methods, such as Bugzie and DREX. The results show that *DevRec* on average improves recall@5 and recall@10 scores of Bugzie by 57.55% and 39.39% respectively. *DevRec* also outperforms DREX by improving the average recall@5 and recall@10 scores by 165.38% and 89.36%, respectively.

***Index Terms*—Developer Recommendation, Multi-label Learning, Topic Model, Composite Method**

I. Introduction

Due to 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 BugID=215252[1]. 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. Nine other people, `Steffen Pingel`, `Mik Kersten`, `Felipe Heidrich`, `Dirk Baeumer`, `Dani Megert`, `Boris Bokowski`, `Steve Northover`, `Olivier Thomann`, and `Grant Gayed`, 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 an extended version 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 (see Section IV-C). 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 char-

‡The work was done while the author was visiting Singapore Management University.

§Corresponding author.

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

acterize 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 developers with smallest distance to the new bug report. *DevRec* combines BR-based and D-based analysis 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.

We evaluate our approach on 5 datasets from different software communities: GCC[2], OpenOffice[3], Mozilla[4], Netbeans[5], and Eclipse[6]. In total we analyze 107,875 bug reports. We measure the performance of the approaches in terms of recall@k. For the 5 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* also outperforms DREX by improving the average recall@5 and recall@10 scores by 165.38% and 89.36%, respectively.

The main contributions of this paper are:

1) We propose *DevRec*, which performs both bug report based analysis and developer 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 outperform Bugzie [5] and DREX [3] on the *developer recommendation* problem by a substantial margin.

The remainder of the paper is organized as follows. We present the empirical study and preliminary materials in Section II. We present our approach *DevRec* in Section III. We present our experiments in Section IV. We present related work in Section V. We conclude and mention future work in Section VI.

II. Empirical Study and Preliminaries

In this section, we first present a simple empirical study on our collected datasets to understand *developer recommendation* problem in Section II-A. Next, we introduce the preliminary materials, i.e., Euclidean distance metric, ML-kNN [6], and topic modeling [7], that would be used in our proposed approach *DevRec* presented in Section III.

[2]http://gcc.gnu.org/bugzilla/
[3]https://issues.apache.org/ooo/
[4]https://bugzilla.mozilla.org/
[5]http://netbeans.org/bugzilla/
[6]https://bugs.eclipse.org/bugs/

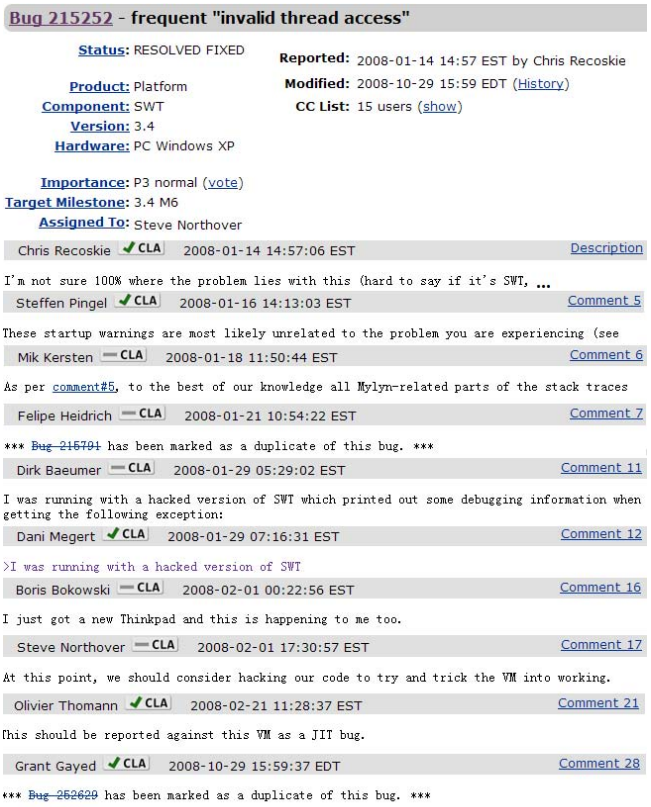

Fig. 1. Bug Report of Eclipse with BugID=215252 - some comments are omitted or truncated.

TABLE I
Collected Information from Bug Reports

Info.	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.	SWT
Developers	Bug resolvers, i.e., developers that contribute to the bug resolution process excluding the reporter.	Steffen Pingel, Mik Kersten, Felipe Heidrich, Dirk Baeumer, Dani Megert, Boris Bokowski, Steve Northover, Olivier Thomann, and Grant Gayed

A. Empirical Study

A typical bug report contains many fields, such as reporter, fixer, creation time, modification time, bug version, platform, CC list, comment list, etc. In this work, we collected 5 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, is presented Table I.

TABLE II
STATISTICS OF COLLECTED BUG REPORTS.

Project	Time	# Reports	# Resolvers	# Terms	# Avg.Resolvers	# Product	# Component
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	2008-01-01–2009-07-23	34,399	3,086	11,234	1.88	114	540

In this paper, we collected 5 datasets from different software development communities: GCC, OpenOffice, Mozilla, Netbeans, and Eclipse. Table II shows the statistics of the 5 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.Resolvers), the number of different products (# Products), and the number of different components (# Component), respectively. All bug reports and their data are download from their bug tracking systems. We collected bug reports with status "closed" and "fixed". For these reports, the set of bug resolvers have been identified. Unless the bug is reopened in the future, no additional resolver would be added. Note that the average number of 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 which 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". Since 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 5 software projects. 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-10 products or components with the most number of bug resolvers. We notice that the number of bug reports per product (or per component) is skewed. Most of the bug reports are for the top-10 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 10 products (out of 37 products) and top 10 components (out of 100 components). For Eclipse, 54.32% and 31.64% bug reports belong to the top 10 products (out of 114 products) and top 10 components (out of 514 components). Thus, information of the product and the component that are affected by the bug is not sufficient to decide suitable developers to be involved in the bug resolution process.

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%

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%

B. Preliminaries

1) Bug Report Representation & 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 4 types of features, i.e., term, topic, product, and component (see Section III-A). 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 [7]; 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 $(p_1, p_2, p_3, ..., p_n)$, where p_i, $i \in \{1, 2, ...n\}$, is the value of b's i^{th} feature.

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

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

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 [8]. Standard classification only assigns one label to each data instance. However, in many settings a data instance can be assigned by more than one labels. For developer recommendation problem, each data instance (i.e., a bug report) can be assigned multiple labels (i.e., developers).

ML-KNN is a state-of-the-art algorithm in the multi-label classification literature [6]. 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 above 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 above 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 labels satisfying the above 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:

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

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

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, button, etc. 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 the they appear in the bug report.

Latent Dirichlet Allocation (LDA) is a well-known topic modeling technique [7]. 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 zero to one, 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 θ_m which contains the probabilities of each topic to be present in the document.

III. 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 III-A, we present BR-Based analysis. In Section III-B, we present D-Based Analysis. Finally, in Section III-C we present a composite of BR-Based and D-Based analysis that would result in *DevRec*.

A. BR-Based Analysis

BR-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.

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 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 (see [9]) that appear in the summary and description of the bug report. Stop words are words that carry little meaning, e.g., I, you, he, she, etc. We remove all such stop words. Stemming is the process of reducing a word to its root form, e.g., 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 appear in the summary and description of the bug report. We make use of Latent Dirichlet Allocation described in Section II 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 4 feature types, i.e., terms, topics, product, and component features. The distance between two bug reports could be calculated by simply computing the Euclidean distance of two vectors (see Section II). Based on this distance, we can find the k-nearest neighbors of a new bug report.

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 [6]. We have provided a short description of this approach in Section II. 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 $BRScore_{BR^{new}}(d)$.

B. 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 4 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.

1) Terms Affinity Score: We use the following formula to compute the term affinity score of a bug report b to a developer d:

$$Terms_b(d) = 1 - \prod_{t \in b}(1 - \frac{n_{d,t}}{n_t + n_d - n_{d,t}}) \qquad (2)$$

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 in bug resolution activities, the number of reports term t appears and the number of reports resolved by developer d that contain 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 above formula is based on [5].

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 Latent Dirichlet Allocation (LDA) [7] to get the topic distribution for each bug report. Section II 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 a 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 refer to the topic vectors corresponding to bug reports that developer d helps in the bug resolution process. Also, given a topic vector v, let $v[t]$ denote 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:

$$Topics_b(d) = 1 - \prod_{t \in b}(1 - \frac{\sum_{v \in T_d} v[t]}{\sum_{v' \in T} v'[t]} \times b[t]) \qquad (3)$$

$t \in b$ denotes a topic contained in the bug report b. Informally put, the above 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) Product and Component Affinity Scores: A developer d might be biased towards 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 since 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:

$$Product_b(d) = \frac{\sum_{b \in B_d} b[p_b]}{\sum_{b' \in B} b'[p_b]} \qquad (4)$$

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 denote the value of the component field of bug report b. The component

TABLE V
AN EXAMPLE DATASET WITH 2 TOPICS, 2 PRODUCTS, 3 COMPONENTS AND 2 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	?	?

affinity score $Component_b(d)$ for b to a developer d is given by:

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

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

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 2 topics, 2 types of product, 3 types of component and 2 developers. The bug report with identifier "Test 1" is the bug report whose resolvers are to be predicted.

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

$$\begin{aligned} Topics_{Test1}(D1) &= 1 - (1 - \frac{0.1 + 0.8}{0.1 + 0.8 + 0.5}) \times \\ &\quad (1 - \frac{0.9 + 0.2}{0.9 + 0.2 + 1 + 0.5}) = 0.1511 \end{aligned}$$

The value of the product field of bug report "Test 1" is P1. In the training set, there are three bug report 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:

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

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.

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.

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:

$$\begin{aligned} DScore_b(d) = \beta_1 \times Terms_b(d) + \beta_2 \times Topics_b(d) + \\ \beta_3 \times Product_b(d) + \beta_4 \times Component_b(d) \end{aligned} \quad (6)$$

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.

C. DevRec: A Composite Method

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

EstimateWeights($B, D, T, EC, ITER, SampleSize$)
Input:
B: Bug Report Collection
D: Developer Collection
T: Bug Report Topic Distribution
EC: Evaluation Criterion
$ITER$: Maximum Number of Iterations (Default Value = 10)
$SampleSize$: Sample Size
Output:$\gamma_1, \gamma_2, \gamma_3, \gamma_4, \gamma_5$
Method:
Build BR-Based Analysis component using B;
Build D-Based Analysis component using B;
Sample a small subset $Samp_B$ of B of size $SampleSize$;
for all bug report $b \in Samp_B$, and developer $d \in D$ **do**
    Compute the BR-Based score , i.e., $BRScore_b(d)$
    Compute the Terms Affinity score, i.e., $Terms_b(d)$
    Compute the Topic Affinity score, i.e., $Topics_b(d)$
    Compute the Product Affinity score, i.e., $Product_b(d)$
    Compute the Component Affinity score, i.e., $Component_b(d)$
end for
while iteration times $iter < ITER$ **do**
    for all i from 1 to 5 **do**
        Choose $\gamma_i = Math.random()$
    end for
    for all i from 1 to 5 **do**
        $\gamma_i^{best} = \gamma_i$
        repeat
            Compute the *DevRec* scores according to Equation (8)
            Evaluate the effectiveness of the combined model on $Samp_B$ and D based on EC
            if EC score of γ_i is better than that of γ_i^{best} **then**
                $\gamma_i^{best} = \gamma_i$
            end if
            Increase γ_i by 0.01
        until $\gamma_i \geq 1$
        $\gamma_i = \gamma_i^{best}$
    end for
end while
Return $\gamma_1, \gamma_2, \gamma_3, \gamma_4, \gamma_5$ which give the best result based on EC

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:

$$DevRec_b(d) = \alpha_1 \times BRScore_b(d) + \alpha_2 \times DScore_b(d) \quad (7)$$

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:

$$\begin{aligned} DevRec_b(d) \quad = \quad & \gamma_1 \times BRScore_b(d) + \gamma_2 \times Terms_b(d) + \\ & \gamma_3 \times Topics_b(d) + \gamma_4 \times Product_b(d) \\ & + \gamma_5 \times Component_b(d) \end{aligned} \quad (8)$$

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. Due to 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 the sample size as 10% of the number of bug reports in B.

Algorithm 1 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 [10], [11] (see Definition 3). Algorithm 1 would return $\gamma_1, \gamma_2, \gamma_3, \gamma_4, \gamma_5$ which give the best result based on EC.

IV. 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) 2.53GHz server with 24GB RAM.

A. Experiment 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 which appear less than 10 times; while for large-scale bug report collections, such as Mozilla, Netbeans, and Eclipse, we delete the terms which appear less than 20 times. For each of the 5 bug report collections, we remove developers who appear less than 10 times. Since they are not active, recommending these developers do 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], [12]. 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 [9] and recall@10 has also been used to evaluate DREX [3]. *DevRec* uses LDA which 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[7] 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], [12] described in Section IV-A). 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 triaging 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*).

[7] http://jgibblda.sourceforge.net/

TABLE VI
RECALL@5 AND RECALL@10 OF *DevRec* AND BUGZIE, AND THE IMPROVEMENT OF *DevRec* OVER BUGZIE (IMPROVE.). THE LAST ROW SHOWS THE AVERAGE RECALL@5 AND RECALL@10 SCORES OF *DevRec* AND BUGZIE, AND THE AVERAGE IMPROVEMENT.

	Recall@5			Recall@10		
Projects	***DevRec***	**Bugzie**	**Improve.**	***DevRec***	**Bugzie**	**Improve.**
GCC	0.5633	0.4743	**18.77%**	0.7072	0.6724	**5.17%**
OpenOffice	0.4826	0.4042	**19.38%**	0.6063	0.5364	**13.05%**
Mozilla	0.5592	0.3214	**73.98%**	0.6755	0.4416	**52.96%**
Netbeans	0.7073	0.4240	**66.81%**	0.8021	0.5448	**47.24%**
Eclipse	0.7989	0.3826	**108.79%**	0.8924	0.4998	**78.55%**
Average.	0.6222	0.4013	**57.55%**	0.7367	0.5390	**39.39%**

TABLE VII
RECALL@5 AND RECALL@10 OF *DevRec* AND DREX, AND THE IMPROVEMENT OF *DevRec* OVER DREX (IMPROVE.)

	Recall@5			Recall@10		
Projects	***DevRec***	**DREX**	**Improve.**	***DevRec***	**DREX**	**Improve.**
GCC	0.5633	0.5217	**7.97%**	0.7072	0.6494	**8.90%**
OpenOffice	0.4826	0.1695	**184.69%**	0.6063	0.2511	**141.48%**
Mozilla	0.5592	0.1063	**426.08%**	0.6755	0.2854	**136.67%**
Netbeans	0.7073	0.4853	**45.74%**	0.8021	0.5629	**42.50%**
Eclipse	0.7989	0.3045	**162.44%**	0.8924	0.4108	**117.28%**
Average.	0.6222	0.3175	**165.38%**	0.7367	0.4319	**89.36%**

We evaluate the performance of our *DevRec* with two metrics, i.e., 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:

$$Recall@k = \frac{1}{m}\sum_{i=1}^{m}\frac{|P_i \cap D_i|}{|D_i|} \quad (9)$$

$$Precision@k = \frac{1}{m}\sum_{i=1}^{m}\frac{|P_i \cap D_i|}{|P_i|} \quad (10)$$

We are interested to answer the following research questions:

RQ1 How is the performance of *DevRec* compared to those of Bugzie and DREX?

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

B. RQ1: Performance of DevRec

In this section, we compare *DevRec* with other state-of-art methods, namely Bugzie and DREX. Table VI compares recall@5 and recall@10 of *DevRec* and Bugzie. Table VII compares recall@5 and recall@10 of *DevRec* and DREX. The recall@5 and recall@10 of *DevRec* vary from 0.4826-0.7989, and 0.6063-0.8924, respectively.

From Table VI, the improvement of our method over Bugzie is substantial. DevRec outperforms Bugzie by 57.55% and 39.39% for average recall@5, and recall@10, respectively. In 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] due to several reasons: first, the problem considered there is different from ours. In [5], it addresses bug triaging problem, i.e., one bug report has only one fixer. In this work, we address the developer recommendation problem, i.e., one bug report has multiple bug resolvers. Second, we drop generic names, e.g., nobody, issues, unassigned, as they do not identify particular developers. From Table VII, the improvement of our method over DREX is substantial. *DevRec* outperforms DREX by 165.38% and 89.36% for average recall@5 and recall@10, respectively. In the Mozilla dataset, *DevRec* achieves the highest improvement of 426.08% and 136.67% over DREX for recall@5, and recall@10, respectively.

TABLE VIII
PRECISION@5 AND PRECISION@10 OF *DevRec* AND BUGZIE.

	Precision@5			Precision@10		
Projects	***DevRec***	**Bugzie**	**Improve.**	***DevRec***	**Bugzie**	**Improve.**
GCC	0.2453	0.2077	**15.33%**	0.1599	0.1514	**5.32%**
OpenOffice	0.2100	0.1740	**17.14%**	0.1331	0.1192	**10.44%**
Mozilla	0.2471	0.1456	**41.08%**	0.1546	0.1043	**32.54%**
Netbeans	0.3196	0.1953	**38.89%**	0.1859	0.1268	**31.79%**
Eclipse	0.2509	0.1212	**51.69%**	0.1431	0.0816	**42.98%**
Average.	0.2546	0.1688	**32.83%**	0.1553	0.1167	**24.61%**

TABLE IX
PRECISION@5 AND PRECISION@10 OF *DevRec* AND DREX.

	Precision@5			Precision@10		
Projects	***DevRec***	**DREX**	**Improve.**	***DevRec***	**DREX**	**Improve.**
GCC	0.2453	0.2314	**6.01%**	0.1599	0.1493	**7.10%**
OpenOffice	0.2100	0.0852	**146.48%**	0.1331	0.0618	**115.37%**
Mozilla	0.2471	0.0664	**272.14%**	0.1546	0.0723	**113.83%**
Netbeans	0.3196	0.2287	**39.75%**	0.1859	0.1355	**37.20%**
Eclipse	0.2509	0.1038	**141.71%**	0.1431	0.0692	**106.79%**
Average.	0.2546	0.1431	**121.22%**	0.1553	0.0976	**76.06%**

Tables VIII and IX compare the precision@5 and precision@10 of *DevRec*, Bugzie, and DREX. The precision@5 and precision@10 of *DevRec* vary from 0.2100-0.3196, and 0.1331-0.1859, respectively. This numbers 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 is 1.88. If we recommend top-10 developers, the best precision@10 would around 0.188. The precision@10 of *DevRec* for the Eclipse dataset is 0.1431, which is close to the optimal value. From Table VIII, the improvement of our method over Bugzie is substantial. DevRec outperforms Bugzie by 32.83% and 24.61% for average precision@5, and precision@10, respectively. From Table IX, the improvement of our method over DREX is also substantial. *DevRec* outperforms DREX by 121.22% and 76.06% for average precision@5, and precision@10, respectively. The results show that clearly DevRec outperforms Bugzie and DREX which are the state-of-the-art techniques.

C. RQ2: Performance of BR-based and D-based Components

DevRec has two components (i.e., BR-based and D-based components), in this section, we investigate the performance of each of them. We want to see if the combination of the two components results in better or poorer performance. Table X and XI present the recall@5 and recall@10 scores of *DevRec* compared with those of BR-based and D-based component.

TABLE X
RECALL@5 AND RECALL@10 OF *DevRec* AND BR-BASED COMPONENT.

	Recall@5			Recall@10		
Projects	***DevRec***	**BR.**	**Improve.**	***DevRec***	**BR.**	**Improve.**
GCC	0.5633	0.4820	**16.87%**	0.7072	0.6490	**8.97%**
OpenOffice	0.4826	0.4670	**3.34%**	0.6063	0.5728	**5.85%**
Mozilla	0.5592	0.5487	**1.91%**	0.6755	0.6567	**2.86%**
Netbeans	0.7130	0.6974	**2.24%**	0.8082	0.7873	**2.65%**
Eclipse	0.7989	0.7873	**1.47%**	0.8924	0.8595	**3.93%**
Average.	0.6234	0.5965	**5.17%**	0.7379	0.7051	**4.83%**

TABLE XI
RECALL@5 AND RECALL@10 OF *DevRec* AND D-BASED COMPONENT.

	Recall@5			Recall@10		
Projects	***DevRec***	**D.**	**Improve.**	***DevRec***	**D.**	**Improve.**
GCC	0.5633	0.5524	**1.97%**	0.7072	0.6952	**1.73%**
OpenOffice	0.4826	0.4783	**0.90%**	0.6063	0.6059	**0.07%**
Mozilla	0.5592	0.5053	**10.67%**	0.6755	0.6313	**7.00%**
Netbeans	0.7130	0.6383	**11.70%**	0.8082	0.7794	**3.70%**
Eclipse	0.7989	0.7602	**5.09%**	0.8924	0.8708	**2.48%**
Average.	0.6234	0.5869	**6.07%**	0.7379	0.7165	**2.99%**

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, respectively. Table XII and XIII present the precision@5 and precision@10 scores of *DevRec* compared with those of BR-based and D-based component. *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.

D. Discussion and Threats to Validity

In this paper, we automatically identify good γ values for *DevRec* following Algorithm 1. The γ values would be optimized (and thus are different) for different datasets and different training frames in our longitudinal data setup.

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. Threats to external validity relates to the generalizability of our results. We have analyzed 107,875 bug reports from 5 large software systems. In the future, we plan to reduce this threat further by analyzing more bug reports from more software systems. 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 [11]. Thus, we believe there is little threat to construct validity.

TABLE XII
PRECISION@5 AND PRECISION@10 OF *DevRec* AND BR-BASED COMPONENT.

	Precision@5			Precision@10		
Projects	***DevRec***	**BR.**	**Improve.**	***DevRec***	**BR.**	**Improve.**
GCC	0.2453	0.2417	**1.49%**	0.1599	0.1563	**2.30%**
OpenOffice	0.2100	0.2028	**3.55%**	0.1331	0.1278	**4.15%**
Mozilla	0.2471	0.2414	**2.36%**	0.1546	0.1508	**2.52%**
Netbeans	0.3196	0.3145	**1.62%**	0.1859	0.1817	**2.31%**
Eclipse	0.2509	0.2460	**1.99%**	0.1431	0.1377	**3.92%**
Average.	0.2546	0.2493	**2.00%**	0.1553	0.1509	**3.04%**

TABLE XIII
PRECISION@5 AND PRECISION@10 OF *DevRec* AND D-BASED COMPONENT.

	Precision@5			Precision@10		
Projects	***DevRec***	**D.**	**Improve.**	***DevRec***	**D.**	**Improve.**
GCC	0.2453	0.2093	**17.20%**	0.1599	0.1454	**9.07%**
OpenOffice	0.2100	0.2080	**0.96%**	0.1331	0.1327	**0.30%**
Mozilla	0.2471	0.2229	**10.86%**	0.1546	0.1460	**5.56%**
Netbeans	0.3196	0.2939	**8.74%**	0.1859	0.1810	**2.64%**
Eclipse	0.2509	0.2365	**6.09%**	0.1431	0.1401	**2.10%**
Average.	0.2546	0.2341	**8.77%**	0.1553	0.1490	**3.93%**

V. RELATED WORK

In this section, we briefly review DREX, studies on bug triaging, and other studies on bug report management.

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. 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 which 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, 4) We consider a larger dataset consisting of more than 100,000 bug reports from 5 projects to evaluate our approach and compares it with DREX and Bugzie. We show that our approach outperform DREX by a substantial margin.

Bug Triaging. Bug triaging refers to the task of finding the most appropriate developer to fix the bug [13], [14], [5], [15], [12]. 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. Anvik et al. and Cubranic et al. use machine learning technologies such as Naive Bayes, SVM, and C4.8 for bug triaging [13], [14]. Tamrawi et al. [5] propose a method called Bugzie, which is based on the fuzzy set theory. 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. Jeong et al. investigate bug reassignment in Eclipse and Mozilla, and propose a graph model based on Markov chain to improve bug triaging performance [15]. Bhattacharya et al. reduce tossing path lengths and improve the accuracy of the approach by Jeong et al. further [12]. In this work, different from the above mentioned studies, we consider bug fixing as a collaborative effort. More than one developer often work together to resolve a bug. Thus, rather than recommending only

the developer that is assigned to fix the bug (aka. the fixer), we would like to recommend all developers that contribute to the bug resolution process (the bug resolvers). Our problem (i.e., developer recommendation for bug resolution) is thus an extension of the bug triaging problem.

Recently, other information sources aside from bug reports (e.g., commits and source code) have been used to recommend appropriate developers. Kagdi et al. uses 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 [16]. 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 [17]. 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 [18]. Different from the above studies, in this work we only analyze information available in bug reports. Often there is an issue in linking bug reports to commits that fix the bug [19], [20]. Still, it would be interesting to combine our approach with the above mentioned approaches when the links between bug reports and relevant commits are well maintained.

Other Studies on Bug Report Management. A number of studies have been proposed to automatically detect duplicated bug reports [21], [22]. A number of studies have been proposed to predict the severity labels of bug reports [23], [24]. A number of other studies locate source code relevant to a bug report [25].

VI. 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 analysis, and compose them together. The experiment results show that, compared with other state-of-the-art approaches, *DevRec* achieves the best performance. *DevRec* improves the average recall@5 and recall@10 scores of Bugzie by 57.55% and 39.39%, respectively. *DevRec* also outperforms DREX by improving the average recall@5 and recall@10 scores by 165.38% and 89.36%, respectively.

In the future, we plan to improve the effectiveness of *DevRec* further (for example, integrate the LDA-GA method proposed by Panichella et al. [26], or employ other text mining solutions, e.g., [27]). We also plan to experiment with even more bug reports from more projects.

Acknowledgment

This research is sponsored in part by NSFC Program (No.61103032) and National Key Technology R&D Program of the Ministry of Science and Technology of China (No2013BAH01B03). 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] D. Bertram, A. Voida, S. Greenberg, and R. Walker, "Communication, collaboration, and bugs: the social nature of issue tracking in small, collocated teams," in *CSCW 2010*.
[2] J. Anvik, L. Hiew, and G. Murphy, "Coping with an open bug repository," in *ETX 2005*.
[3] W. Wu, W. Zhang, Y. Yang, and Q. Wang, "DREX: Developer recommendation with k-nearest-neighbor search and expertise ranking," in *APSEC 2011*.
[4] X. Xie, W. Zhang, Y. Yang, and Q. Wang, "Dretom: Developer recommendation based on topic models for bug resolution," in *PROMISE 2012*.
[5] A. Tamrawi, T. Nguyen, J. Al-Kofahi, and T. Nguyen, "Fuzzy set and cache-based approach for bug triaging," in *ESEC/FSE 2011*.
[6] M. Zhang and Z. Zhou, "Ml-knn: A lazy learning approach to multi-label learning," *Pattern Recognition*, 2007.
[7] D. Blei, A. Ng, and M. Jordan, "Latent dirichlet allocation," *Journal of Machine Learning Research*, 2003.
[8] H. J. and M. Kamber, *Data Mining: Concepts and Techniques*, 2006.
[9] C. Manning, P. Raghavan, and H. Schutze, *Introduction to information retrieval*, 2008.
[10] G. Tsoumakas and I. Katakis, "Multi-label classification: An overview," *IJDWM*, 2007.
[11] X. Xia, D. Lo, X. Wang, and B. Zhou, "Tag recommendation in software information sites," in *MSR 2013*.
[12] P. Bhattacharya and I. Neamtiu, "Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging," in *ICSM 2010*.
[13] J. Anvik, L. Hiew, and G. Murphy, "Who should fix this bug?" in *ICSE 2006*.
[14] D. Čubranić, "Automatic bug triage using text categorization," in *SEKE*.
[15] G. Jeong, S. Kim, and T. Zimmermann, "Improving bug triage with bug tossing graphs," in *ESEC/FSE 2009*.
[16] H. Kagdi, M. Gethers, D. Poshyvanyk, and M. Hammad, "Assigning change requests to software developers," *Journal of Software: Evolution and Process*, 2012.
[17] M. Linares-Vásquez, K. Hossen, H. Dang, H. Kagdi, M. Gethers, and D. Poshyvanyk, "Triaging incoming change requests: Bug or commit history, or code authorship?" in *ICSM*, 2012.
[18] K. Kevic, S. C. Müller, T. Fritz, and H. C. Gall, "Collaborative bug triaging using textual similarities and change set analysis," in *CHASE 2013*.
[19] Y. Tian, J. L. Lawall, and D. Lo, "Identifying linux bug fixing patches," in *ICSE*, 2012.
[20] T. F. Bissyandé, F. Thung, S. Wang, D. Lo, L. Jiang, and L. Réveillère, "Empirical evaluation of bug linking," in *CSMR*, 2013.
[21] C. Sun, D. Lo, S.-C. Khoo, and J. Jiang, "Towards more accurate retrieval of duplicate bug reports," in *ASE*, 2011.
[22] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo, "A discriminative model approach for accurate duplicate bug report retrieval," in *ICSE*, 2010.
[23] T. Menzies and A. Marcus, "Automated severity assessment of software defect reports," in *ICSM*, 2008.
[24] Y. Tian, D. Lo, and C. Sun, "Information retrieval based nearest neighbor classification for fine-grained bug severity prediction," in *WCRE 2012*.
[25] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports," in *ICSE*, 2012.
[26] A. Panichella, B. Dit, R. Oliveto, M. D. Penta, D. Poshyvanyk, and A. D. Lucia, "How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms," in *ICSE 2013*.
[27] X. Wang, D. Lo, J. Jiang, L. Zhang, and H. Mei, "Extracting paraphrases of technical terms from noisy parallel software corpora," in *ACL/IJCNLP*, 2009.