IET Software

Research Article

IET Journals
The Institution of Engineering and Technology

Empirical study on developer factors affecting tossing path length of bug reports

ISSN 1751-8806
Received on 24th August 2017
Revised 9th February 2018
Accepted on 9th March 2018
E-First on 6th April 2018
doi: 10.1049/iet-sen.2017.0159
www.ietdl.org

Hongrun Wu[1], Haiyang Liu[1], Yutao Ma[1] ✉

[1]*State Key Laboratory of Software Engineering, Wuhan University, Wuhan 430072, People's Republic of China*
✉ *E-mail: ytma@whu.edu.cn*

Abstract: Bug reassignment (also known bug tossing) is a common activity in the life cycle of bug reports, and it increases the cost of time and labour to fix bugs in software projects. In large-scale projects, about 6–10% of bug reports are tossed at least three times. However, the nature of repeatedly-tossed bug reports was usually overlooked in previous works. This study focuses on developer features from four aspects, namely network centrality, developer workspace, developer expertise, and transmissibility of developers, to investigate which factors affect the tossing path length (TPL). By using statistical methods, this study finds that working theme, product, component, and degree centrality are key impact factors affecting the change of TPL. The four key features are then simplified to three core features, namely working theme, product, and component, which contribute about 90% of the variance of TPL. Finally, the two feature groups mentioned above are applied in six machine learning algorithms to predict potential developers for bug reports from Eclipse and Mozilla, and the results validate the effectiveness of the feature groups for developer recommendation. Hence, this study provides an easy-to-use feature selection method to train quality developer recommenders for automatic bug triage in an efficient way.

1 Introduction

Software bug resolution is an important part of software development because it is highly related to development and maintenance costs. The process of assigning bug reports to appropriate developers is known as bug triage [1]. Ideally, each new bug report will be efficiently assigned to a suitable developer, but in fact, an assigned bug report may have to be reassigned if the developer cannot fix this bug for a variety of reasons. The reassignment will repeat until a developer fixes the bug. This process of bug reassignment is called bug tossing [2]. Some empirical studies have shown that, on average, it takes about 40 days to assign a bug report to the first developer in the Eclipse project, and it then takes additional 100 days or more to reassign the bug report to the second developer [3, 4].

During the bug triaging process, improper assignments and tosses (referred to as reassignments) can obviously increase the number of repeatedly-tossed bug reports. The goal of improving the efficiency of resolving bugs lies in two aspects: recommending appropriate developers for a given bug report, and reducing the number of tosses (i.e. tossing path length (TPL)). Many approaches based on machine learning (ML) have been proposed to improve the accuracy of recommending matchable developers [1, 5–13], in which a classifier (also known as a classification model) is built by using some features of bug reports such as title, comment, and description. Besides, a few approaches that combine ML and tossing graph (TG) have also been proposed to reduce TPL as well as to improve the prediction accuracy [4, 14, 15]. Although the results of these approaches are promising, the reasons why a few bug reports are repeatedly tossed remain unknown.

To answer the question mentioned above, in this study, we focus on developer factors affecting the number of tosses during the bug triaging process. That is to say, the goal of our work is to identify the key impact factors that shape repeatedly-tossed bug reports, so as to facilitate the process of feature engineering for classifier construction, which is fundamental to the application of ML in bug triage. We analyse the commonly-used factors in previous studies from the perspectives of developer and bug report, and we define the factors that affect the relationship between developer and bug triage as developer features. To this end, we consider a total of 16 factors closely relevant to developer features, and they are extracted from four aspects: network centrality, workspace, expertise, and transmissibility. The network centralities [16], including degree, in-degree, out-degree, betweenness, directed betweenness, closeness, in-closeness, and out-closeness are defined based on graph theory. The developer workspace, including product and component, is referred to as the project workspace in which a developer works. The developer expertise, including fixing probability and working theme, is defined by all the bug reports that a developer has dealt with. The transmissibility includes tossing probability and the identity of a developer acting as a fixer or tosser. The empirical results on Eclipse and Mozilla and technical contributions of this study are summarised as follows.

i. By using a multivariable regression model, four factors (i.e. degree centrality, product, component, and working theme) are identified as key impact factors from nine factors which have high correlations with TPL. The four factors contribute more than 95% of the variance of TPL.
ii. The set of main impact factors is further reduced to a minimal subset of core factors (i.e. product, component, and working theme) by calculating the variance inflation factor (VIF). The three factors contribute about 90% of the variance of TPL.
iii. By using six common ML algorithms, the developer classifiers trained with the two feature sets mentioned above perform better than those classifiers built using the commonly-used features of bug reports and developers described in previous studies. Therefore, our work provides an easy-to-use feature selection method to train quality developer classifiers for automatic bug triage efficiently.

In the rest of this paper, Section 2 introduces the work related to our study. Section 3 presents the measures of TPL and the 16 developer factors. Section 4 introduces the experimental setup, and Section 5 outlines research questions and presents the results of the research questions as well as their implications for practice and research. In Section 6, we discuss some potential threats to the validity of our work. Finally, Section 7 concludes this paper and presents our future work.

2 Related work

Since the early work of Perry and Stieg [17], many researchers have investigated effective software quality assurance approaches from the perspective of bug triage. Generally speaking, there are three mainstream types of approaches, namely information retrieval (IR) based, ML based, and graph theory-based approaches [13, 18, 19]. In the following subsections, we will give a brief introduction to each of the three types of approaches.

2.1 IR-based approaches

Some IR approaches were proposed to recommend appropriate developers for a new bug report [20–25]. For example, Canfora and Cerulo presented an IR-based method that identified candidate developers using the textual description of a new change request as a query [20]. Due to the difficulty in mining bug repositories without sufficient bug fixing records, Shokripour *et al.* collected the necessary data from the version-control repositories using some information extraction methods [22], so as to obtain better results. In the work of Nagwani and Verma [21], they extracted frequent terms from the textual information of bug reports and used term similarity to identify appropriate developers for newly reported bugs. Recently, Xia *et al.* recommended developers by calculating the affinity score of each developer to possible bug reports [24, 25].

2.2 ML-based approaches

In addition to the works based on IR, researchers have also proposed various ML-based methods with the goal of assigning bug reports automatically and accurately [1, 5–13]. Cubraknic and Murphy first used the Naive Bayes (NB) classification algorithm to semi-automate the process of bug assignment [5]. Then, Anvik *et al.* improved the approach proposed by Cubraknic *et al.*, and they utilised the support vector machine (SVM) algorithm to recommend a set of appropriate developers to a bug manager [1]. Baysal *et al.* presented a theoretical framework-based SVM for automatic bug assignment, which considered developer expertise and workloads [8]. Lin *et al.* reported a case study of automatic bug assignment which used Chinese text and other non-text information of bug reports [7]. In [6, 10, 13], several comparative analyses of supervised ML algorithms for automatic bug triage were presented in detail to guide the selection of these algorithms.

2.3 TG-based approaches

Unlike the methods mentioned above, a few researchers found a different direction based on the theory of TG to improve the prediction accuracy of triaging bug reports. Jeong *et al.* proposed a TG model based on Markov chains [4], and the proposed model reduced tossing steps by up to 72%. Then, Bhattacharya *et al.* improved the prediction accuracy and reduced TPL by employing the combination of ML classifiers and a specific TG [14]. Zhang and Lee assigned bug reports automatically using concept profile and network analysis [26]. Wang *et al.* presented the heterogeneous developer network model *DevNet*, a framework for representing and analysing developer collaboration in bug repositories [27]. Park *et al.* developed an automatic triaging system considering both accuracy and cost and proposed a model *CosTriage* to characterise user-specific experiences and estimate the cost of each bug category [28]. Recently, Zhang *et al.* proposed a model *KSAP*, which used historical bug reports and heterogeneous networks of bug repositories to improve the performance of automatic bug assignment [29]. In addition, some hybrid methods combining ML and TG have attracted much attention in recent years.

2.4 Summary

The textual features of bug reports have been widely used in bug triage. Also, some previous studies investigated efficient bug assignment methods based on developer features. For example, the early work, by Jeong *et al.*, took into consideration the tossing probability between developers [4]. Besides, the interval of the last activity of a developer was also considered to filter out inactive (or retired) developers [14, 15]. To the best of our knowledge, developer attributes, such as expertise, interest, and working partners, are also used in [30–35].

Most of the approaches mentioned above used the features of bug reports and developers, and the selections of such features are essential for improving the prediction accuracy. In Table 1, we show an overview of the principal features of bug reports used in previous studies, and the bold lines are used to separate the IR-, ML-, and TG-based work. The marker '*x*' denotes that a feature is used in the corresponding paper. It can be seen from this table that the features in the first five columns are often used in automatic bug triage, suggesting that the vast majority of these approaches extract textual information (i.e. summary, description, and comments) and the information of source location (i.e. product and component). Similarly, in Table 2 we also present an overview of the main features of developers used in previous studies. Surprisingly, developer features used vary from one paper to another, including fixing probability, tossing probability, degree, betweenness, and closeness centrality, implying that there is yet no consensus on widely-recognised developer features.

Unlike those previous studies that focused on new methods for bug triage, in this work, we explore the main reasons why some bug reports are repeatedly tossed from a new perspective of developer feature. Our work acts as a feature selector for ML-based approaches. Thus, we can efficiently train ML classifiers using the core factors obtained in this work to predict suitable developers for given bug reports.

3 Measures and calculation methods

3.1 Tossing path

For the detailed descriptions of bug reports and bug life cycle, please refer to [4, 40]. First of all, we present the concepts of tossing path and TPL.

A tossing path is defined as a set of finite tossing steps among developers in the bug tossing process [4]. In Fig. 1, the tossing path starts from the first assigned developer d_1, and d_1 tosses out this bug report to d_2 if he/she does not fix the bug. The reassignment from d_1 to d_2 ($d_1 \rightarrow d_2$) is the first tossing step, $d_2 \rightarrow d_3$ for the second tossing step, and so forth. The path ends until developer d_l fixes the bug.

A tossing path must have one fixer; otherwise, our work does not consider all the bug reports without any fixer. The number of tossing steps $(l-1)$ is defined as the tossing length of a tossing path $K = d_1 \rightarrow d_2 \rightarrow d_3, \ldots, \rightarrow d_l$. If the tossing path of a bug report has only one element, i.e. $K = d_1 | d_1 = d_l$, this suggests that the first assigned developer who receives the bug report finally fixes it. Based on all available tossing paths, we can create a developer collaboration network, where each vertex represents a developer and each edge indicates a tossing step between two developers.

3.2 Measures of developer factors

In this section, we detail the definitions of the four types of developer factors used in this study, namely network centrality, workspace, expertise, and transmissibility.

3.2.1 Network centralities: In the analysis of networks, three main indicators of centrality [16], namely degree, betweenness, and closeness, are used to measure the importance of vertices, while the importance of an edge is usually measured by edge betweenness centrality which reflects the communicating ability of the edge in a network. The definitions of these measures are given in (1)–(10).

Degree centrality: The degree centrality of a vertex (i.e. developer) d_i is defined in the following equation [16]:

$$S_D(d_i) = \sum_{j=1}^{n} \delta(d_i, d_j), \tag{1}$$

Table 1 Bug report features used in previous studies. The symbol × indicates that a given feature was used in the corresponding paper

Reference	Summary	Description	Comment	Product	Component	Platform	Version	Type	Phase	Priority	Submitter	Activity
Cubraknic and Murphy [5]	×	×	—	—	—	—	—	—	—	—	—	—
Anvik *et al.* [1]	×	×	—	—	—	—	—	—	—	—	—	—
Ahsan *et al.* [6]	×	×	—	×	×	×	—	—	—	—	—	—
Lin *et al.* [7]	×	×	—	—	×	—	—	×	×	×	×	—
Baysal *et al.* [8]	×	×	—	—	—	—	—	—	—	—	—	—
Helming *et al.* [9]	×	×	—	—	—	—	—	—	—	—	—	—
Anvik and Murphy [10]	×	×	—	—	—	—	—	—	—	—	—	—
Alenezi *et al.* [11]	×	—	—	—	—	—	—	—	—	—	—	—
Shokripour *et al.* [12]	×	×	×	×	×	—	—	—	—	—	—	—
Jonsson *et al.* [13]	×	×	—	—	—	—	×	×	—	×	—	—
Canfora and Cerulo [20]	×	×	×	—	—	—	—	—	—	—	—	—
Matter *et al.* [31]	—	×	—	—	—	—	—	—	—	—	—	—
Xuan *et al.* [36]	×	×	—	—	—	—	—	—	—	—	—	—
Xie *et al.* [32]	×	×	—	—	—	—	—	—	—	—	—	—
Kagdi *et al.* [23]	—	—	×	×	—	—	—	—	—	—	—	—
Linaresvasquez *et al.* [33]	×	×	×	—	—	—	—	—	—	—	—	—
Nagwani and Verma [21]	×	×	×	—	—	—	—	—	—	×	—	—
Shokripour *et al.* [22]	×	×	×	—	—	—	—	—	—	—	—	—
Naguib *et al.* [30]	×	×	—	—	×	—	—	—	—	—	—	—
Xia *et al.* [24]	×	×	×	×	×	—	—	—	—	—	—	—
Park et al. [28]	×	×	—	—	—	×	×	—	×	—	—	—
Xia *et al.* [25]	×	×	×	×	×	—	—	—	—	—	—	—
Jeong *et al.* [4]	×	×	—	—	—	—	—	—	—	—	—	—
Chen *et al.* [37]	×	×	—	×	×	—	—	—	—	—	—	—
Wu *et al.* [34]	—	×	×	—	—	—	—	—	—	—	—	—
Bhattacharya *et al.* [15]	×	×	—	—	×	×	—	—	—	—	—	×
Zhang and Lee [26]	—	×	×	—	—	—	—	—	—	—	—	—
Wang *et al.* [27]	—	—	×	×	×	—	—	—	—	—	—	—
Zhang *et al.* [29]	—	—	×	×	×	—	—	—	—	—	—	—

Table 2 Developer features used in previous studies

Reference	Degree	Betweenness	Closeness	Pagerank	Expertise	Interest	FixingProb	TossingProb	Interval	Cost	Role	AuthorInfo
Pinzger *et al.* [38]	×	×	×	—	—	—	—	×	—	—	—	—
Jeong *et al.* [4]	—	—	—	—	—	—	—	×	—	—	—	—
Zanetti *et al.* [35]	×	×	×	—	—	—	—	—	—	—	—	—
Guo *et al.* [39]	—	—	—	—	—	—	×	—	—	—	—	—
Bhattacharya and Neamtiu [14]	—	—	×	—	—	—	—	×	×	—	—	—
Wu *et al.* [34]	×	×	×	×	—	—	—	—	—	—	—	—
Bhattacharya *et al.* [15]	—	—	—	—	—	—	—	×	×	—	—	—
Xie *et al.* [32]	—	—	—	—	×	×	×	—	—	—	—	—
Linaresvasquez *et al.* [33]	—	—	—	—	—	—	—	—	—	—	—	×
Zhang and Lee [26]	—	—	—	—	—	—	×	—	—	×	—	—
Naguib *et al.* [30]	—	—	—	—	×	—	—	—	—	—	×	—
Park *et al.* [28]	—	—	—	—	—	—	—	—	—	×	—	—
Baysal *et al.* [8]	—	—	—	—	×	—	—	—	—	—	—	—

where $\delta(d_i, d_j)$ is 1 when a link exists between d_i and d_j; otherwise, it is 0. Here, n is the number of vertexes in the network.

For directed networks, the degree is measured by in-degree and out-degree, which are defined in (2) and (3), respectively [16]. Here, $\delta(d_i \leftarrow (\rightarrow) d_j)$ is 1 if d_j has a link to (from) d_i

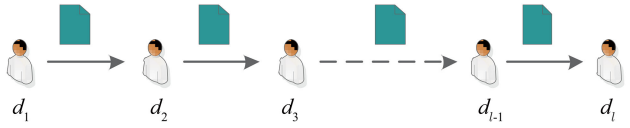

Fig. 1 *Illustration of the tossing path of a bug report*

$$S_{\mathrm{ID}}(d_i) = \sum_{j=1}^{n} \delta(d_i \leftarrow d_j), \tag{2}$$

$$S_{\mathrm{OD}}(d_i) = \sum_{j=1}^{n} \delta(d_i \rightarrow d_j). \tag{3}$$

Closeness centrality: In undirected networks, the closeness of vertex d_i is shown in the following equation:

$$S_{\mathrm{C}}(d_i) = \frac{\sum_{d_j \neq d_i} (1/s(d_j, d_i))}{n-1}, \tag{4}$$

where $s(d_i, d_j)$ is the total distance of the shortest path [41] between d_i and d_j.

For directed networks, in-closeness and out-closeness are presented in (5) and (6), respectively. Here, $s(d_i \leftarrow d_j)$ is the length of the shortest path from d_j to d_i, and vice versa for $s(d_i \rightarrow d_j)$

$$S_{\mathrm{IC}}(d_i) = \frac{\sum_{d_j \neq d_i} (1/s(d_i \leftarrow d_j))}{n-1}, \tag{5}$$

$$S_{\mathrm{OC}}(d_i) = \frac{\sum_{d_j \neq d_i} (1/s(d_i \rightarrow d_j))}{n-1}. \tag{6}$$

Node betweenness centrality: In undirected networks, node betweenness is defined in (7), where $h_{d_t d_j}$ is the number of the shortest paths between vertexes d_i and d_j and $h_{d_t d_j}(d_i)$ is the number of those paths [41] also passing through d_i

$$S_{\mathrm{UB}}(d_t) = \frac{\sum_{(1/s(d_i \rightarrow d_j))} (h_{d_t d_j}(d_t)/h_{d_t d_j})}{(n-1)(n-2)/2}. \tag{7}$$

The node betweenness centrality in directed networks can be written as (8), where the shortest paths are calculated based on directed paths [16]

$$S_{\mathrm{DB}}(d_t) = \frac{\sum_{d_t \neq d_i \neq d_j} (h_{d_t d_j}(d_t)/h_{d_t d_j})}{(n-1)(n-2)}. \tag{8}$$

Edge betweenness centrality: The definitions of the edge betweenness [16] in undirected and directed networks are formulated in (9) and (10), respectively. Note that $h_{d_t d_j}(e)$ is the number of the shortest paths that vertexes d_i and d_j pass through edge e.

$$S_{\mathrm{UH}}(e) = \frac{\sum_{d_i \neq d_j} (h_{d_t d_j}(e)/h_{d_t d_j})}{n(n-1)/2}, \tag{9}$$

$$S_{\mathrm{DH}}(e) = \frac{\sum_{d_i \neq d_j} (h_{d_t d_j}(e)/h_{d_t d_j})}{n(n-1)}. \tag{10}$$

3.2.2 Developer workspace: In previous studies [1, 12], the location of bugs, e.g. component, is a key reference for the recommendation of suitable developers, and the developers who are responsible for a given component (or usually fix bugs from the component) may provide practical information about bug fixing. In this study, developer workspace, including component and product, indicates that developers usually fix bugs from location-specific components or products.

Working component: The developer component is determined based on all the components of the bugs in which a developer has ever participated. The probability of a developer d_i working in component c is formulated in (11). Here, B_{d_i} is the set of bug reports in which d_i has ever participated in the bug triaging process, and $\delta(b, c)$ represents whether bug report b comes from c. If b is from c, $\delta(b, c) = 1$; otherwise, $\delta(b, c) = 0$. The working component of developer d_i is then formulated in (12)

$$P(d_i, c) = \frac{\sum_{b \in B_{d_i}} \delta(b, c)}{|B_{d_i}|}, \tag{11}$$

$$d_i[\hat{c}] = \arg \max_{c \in c^*} P(d_i, c), \tag{12}$$

where c^* denotes all the components in which d_i has ever participated. A developer may fix many bugs, and these bugs are probably from different components. Therefore, the developer may work on multiple components.

Working product: Similarly, the working product of developer d_i is defined by the following equation:

$$d_i[\hat{p}] = \arg \max_{p \in p^*} \frac{\sum_{b \in B_{d_i}} \delta(b, p)}{|B_{d_i}|}, \tag{13}$$

where p^* is the set of products in which d_i has been involved and $\delta(b, p)$ represents whether bug report b is from product p.

3.2.3 Developer expertise: Developer expertise represents a developer's proficiency in processing some specific types of bugs. Two types of developer expertise are considered in this study, i.e. working theme and fixing probability.

Working theme: The working theme of a developer is a topic to which most of the bug reports fixed by the developer belong, and it is defined according to the topics of historical bug reports in which the developer has been involved. The probability of developer d_i working on theme t is shown in (14), where $\theta(b, t)$, generated from an latent Dirichlet allocation (LDA) model [42], is the probability that the corresponding bug report b belongs to topic t, and B is the set of bug reports

$$P(d_i, t) = \frac{\sum_{b \in B_{d_i}} \theta(b, t)}{\sum_{t \in T} \sum_{b \in B_{d_i}} \theta(b, t)}. \tag{14}$$

A developer may have at least one working theme because a bug report may have a few topics. Here, $\hat{t}$ is defined as the working theme of developer d_i (see (15)), which is determined by using the most likely topic of d_i

$$d_i[\hat{t}] = \arg \max_{t \in t^*} P(d_i, t), \tag{15}$$

where t^* is the set of working themes of d_i.

Fixing probability: The fixing probability of a developer denotes his/her ability to fix bug reports, which is the ratio of the number of bugs that have been resolved to the total number of received bug reports

$$P(d_i) = \frac{f_{d_i}}{m_{d_i}}, \tag{16}$$

where m_{d_i} is the number of bug reports assigned to developer d_i and f_{d_i} is the number of bugs fixed by d_i.

3.2.4 Transmissibility: In this study, transmissibility means that developers play the role as a dispatcher in the bug tossing process, and it includes two features: tossing probability and tosser identity.

Tossing probability: The tossing probability, the probability that a developer tosses out a bug report to another developer, has been

Table 3 Normalisation of the four types of features

Type	Name	Normalised equation
network centrality	in-degree	$N_{\mathrm{ID}}(l)=\frac{\sum_{k=1}^{K}(\sum_{i=1,d_i\in Q_k}^{l+1}S_{\mathrm{ID}}(d_i)/l+1)}{K}$ (r1)
	out-degree	$N_{\mathrm{OD}}(l)=\frac{\sum_{k=1}^{K}(\sum_{i=1,d_i\in Q_k}^{l+1}S_{\mathrm{OD}}(d_i)/l+1)}{K}$ (r2)
	degree	$N_{\mathrm{D}}(l)=\frac{\sum_{k=1}^{K}(\sum_{i=1,d_i\in Q_k}^{l+1}S_{\mathrm{D}}(d_i)/l+1)}{K}$ (r3)
	in-closeness	$N_{\mathrm{IC}}(l)=\frac{\sum_{k=1}^{K}(\sum_{i=1,d_i\in Q_k}^{l+1}S_{\mathrm{IC}}(d_i)/l+1)}{K}$ (r4)
	out-closeness	$N_{\mathrm{OC}}(l)=\frac{\sum_{k=1}^{K}(\sum_{i=1,d_i\in Q_k}^{l+1}S_{\mathrm{OC}}(d_i)/l+1)}{K}$ (r5)
	closeness	$N_{\mathrm{C}}(l)=\frac{\sum_{k=1}^{K}(\sum_{i=1,d_i\in Q_k}^{l+1}S_{\mathrm{C}}(d_i)/l+1)}{K}$ (r6)
	directed node betweenness	$N_{\mathrm{DB}}(l)=\frac{\sum_{k=1}^{K}(\sum_{i=1,d_i\in Q_k}^{l+1}S_{\mathrm{DB}}(d_i)/l+1)}{K}$ (r7)
	undirected node betweenness	$N_{\mathrm{UB}}(l)=\frac{\sum_{k=1}^{K}(\sum_{i=1,d_i\in Q_k}^{l+1}S_{\mathrm{UB}}(d_i)/l+1)}{K}$ (r8)
	directed edge betweenness	$N_{\mathrm{DH}}(l)=\frac{\sum_{k=1}^{K}(\sum_{i=1,j=i+1\&d_i,d_j\in Q_k}^{i\le l,j\le l+1}S_{\mathrm{DH}}(d_i,d_j)/l)}{K}$ (r9)
	undirected edge betweenness	$N_{\mathrm{UH}}(l)=\frac{\sum_{k=1}^{K}(\sum_{i=1,j=i+1\&d_i,d_j\in Q_k}^{i\le l,j\le l+1}S_{\mathrm{UH}}(d_i,d_j)/l)}{K}$ (r10)
developer workspace	component	$W_{\mathrm{C}}(l)=\frac{\sum_{k=1}^{K}(\sum_{i=1,j=i+1\&d_i,d_j\in Q_k}^{i\le l,j\le l+1}\delta(d_i[\hat{c}],d_j[\hat{c}])/l)}{K}$ (r11)
	product	$W_{\mathrm{P}}(l)=\frac{\sum_{k=1}^{K}(\sum_{i=1,j=i+1\&d_i,d_j\in Q_k}^{i\le l,j\le l+1}\delta(d_i[\hat{p}],d_j[\hat{p}])/l)}{K}$ (r12)
developer expertise	fixing probability	$E_{\mathrm{P}}(l)=\frac{\sum_{k=1}^{K}(\sum_{i=1,d_i\in Q_k}^{l+1}P(d_i)/l+1)}{K}$ (r13)
	working theme	$E_{\mathrm{T}}(l)=\frac{\sum_{k=1}^{K}(\sum_{i=1\&d_i\in Q_k}^{l+1}\delta(d_i[\hat{t}],b[t])/l+1)}{K}$ (r14)
developer collaboration	tossing probability	$O_{\mathrm{P}}(l)=\frac{\sum_{k=1}^{K}(\sum_{i=1,j=i+1\&d_i,d_j\in Q_k}^{i\le l,j\le l+1}P(d_i,d_j)/l)}{K}$ (r15)
	fraction of tossers	$O_{\mathrm{T}}(l)=\frac{\sum_{k=1}^{K}(\sum_{i=1,d_i\in Q_k}^{l+1}R(d_i)/l+1)}{K}$ (r16)

used to reduce the number of tosses [4, 14, 15, 34]. This measure is defined by the following equation:

$$P(d_i,d_j)=\frac{k_{d_i\to d_j}}{m_{d_i}}, \tag{17}$$

where $k_{d_i\to d_j}$ is the number of bug reports tossed from developer d_i to developer d_j.

Developer identity: Many developers have never fixed a bug, and they always toss out bug reports to other developers when they receive any bug reports. In this study, this category of developers is called tosser. A developer d_i is a tosser if $P(d_i)=0$, i.e. the number of bug reports fixed by the developer equals 0, which is a special case of (16). The tosser identity of d_i is defined in (18), where $R(d_i)=1$ indicates that d_i is a tosser

$$\begin{cases}R(d_i)=1 & \text{if } P(d_i)=0,\\ R(d_i)=0 & \text{if } P(d_i)>0.\end{cases} \tag{18}$$

4 Experimental setup

4.1 Data collection

We retrieved 200,000 and 220,000 bug reports from Eclipse and Mozilla, respectively, and the 'Status' and 'Resolution' of the collected bug reports were labelled with 'Verified' and 'Fixed', respectively. All the bug reports were fixed during the period from 11 October 2001, to 2 November 2011, and 3893 and 7936 developers were involved in Eclipse and Mozilla, respectively. Meanwhile, Eclipse and Mozilla have 570 and 2834 tossers, respectively. Besides, Eclipse includes 167 products and 783 components, and six products and 85 components for Mozilla.

4.2 Data processing

The data processing in our work includes two steps: the first one is to clean the raw data and filter out unnecessary information, and the second one is to prepare the experimental data for answering our research questions.

Tossing records in the 'History' field were extracted to construct tossing step and developer network. The 'Summary,' 'Description', and 'Comment' attributes of bug reports were put together as the free-form textual content, and the filter we used for the free-form textual content was similar to that of the previous study [13]. Note that we lessen the dimensionality of training data (more than 100,000 dimensions) using LDA.

Due to the differences in the scales of numerical values for the four types of features, all the features should be normalised. Each bug report that has been resolved has a tossing path including several developers, the normalisation of a given feature is to average the values of the feature for all the developers at each tossing path. So the average value of the 10 network centralities for K tossing paths with length l is then normalised in (r1)–(r10), respectively, in Table 3. Here, Q_k denotes the set of developers at the kth tossing path and $l+1$ is the number of developers.

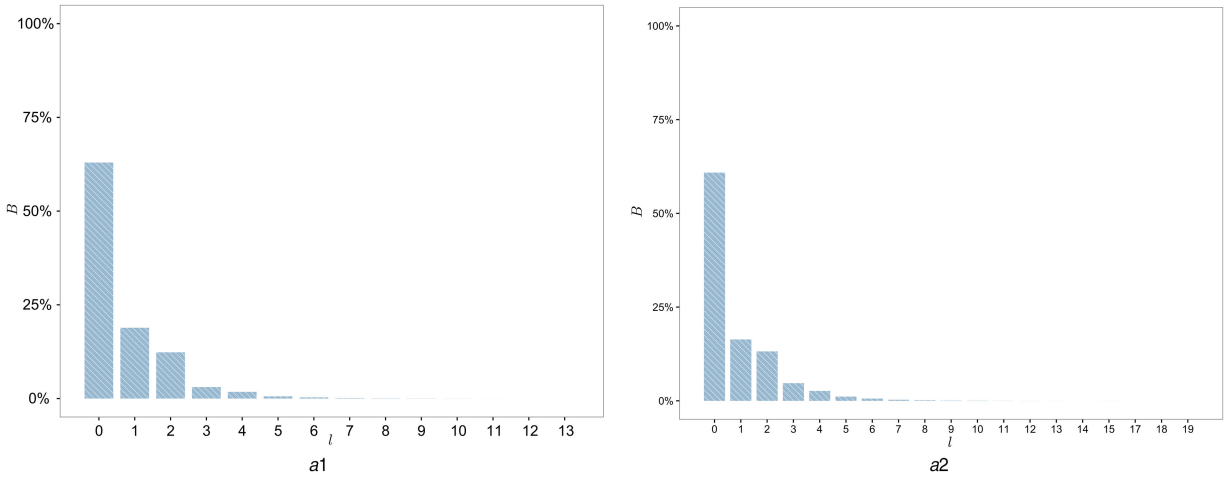

Fig. 2 *Bar plot of the percentage of bug reports grouped by TPL. The left panel is for Eclipse and right one for Mozilla*

Any two developers at a tossing path may not work in the same workspace. The average probability ($W_C(l)$) that two developers working on the same component for K tossing paths with length l is defined in (r11) in Table 3. If the components of two adjacent developers d_i and d_j are the same, $\delta(d_i[c], d_j[c])$ equals 1; otherwise, $\delta(d_i[c], d_j[c])$ is 0. Similarly, the average probability ($W_P(l)$) of two adjacent developers working on the same product is shown in (r12).

For the developer expertise, the average fixing probability of developers is defined in (r13), and the average probability that a developer's working theme is the same with the topic of bug report b defined in (r14). If d_i shares the same topic with b, $\delta(d_i[t], b[t]) = 1$; otherwise, $\delta(d_i[t], b[t]) = 0$.

Finally, for the fourth type of feature, the average tossing probability ($O_P(l)$) is normalised in (r15). Equation (r16) represents the average proportion of tossers ($O_T(l)$) in a tossing path with length l.

4.3 Data analysis

Fig. 2 shows the percentage of bug reports (B) grouped by TPL (l). The largest TPL is 13 for Eclipse, and 19 for Mozilla. It is noticeable that the bar at $l = 16$ does not exist in Fig. 2*a*2. For Eclipse and Mozilla, about 60% of bug reports were fixed at $l = 0$, i.e. these bug reports have not been reassigned. Also, about 20% of the remaining bug reports are tossed more than twice ($l \geq 2$).

5 Results and discussion

In this section, we will show the analysis and experimental results to answer the research questions. It is noticeable that, for each pair of the first three successive research questions, the answer to the latter depends on the result of the former. For the last research question, *RQ4*, it is an application of leveraging developer features, discussed in *RQ2* and *RQ3*, respectively, to recommend potential developers using ML algorithms.

5.1 Answers to research questions

Previous studies have proposed many approaches to recommend potential developers, predict bug fixing time, and understand repeatedly-tossed bug reports. However, the question why many bug reports are repeatedly tossed among developers remains answered quantitatively. In this study, we formulate this issue as a regression problem (i.e. exploring the prime factors that result in long tossing paths) and break down it into three research questions *RQ1–RQ3*.

RQ1. Which developer features are relevant to the change of TPL?

Motivation. As shown in Tables 1 and 2, those previous studies utilised different types of features. However, whether these commonly-used features contribute to the change of TPL remains unknown. That is to say, we need to filter out irrelevant and weakly-correlated features from the 16 features in question first. As shown in Tables 1 and 2, 16 features are commonly used in many previous studies. However, whether these commonly-used features contribute to the change of TPL remains unknown. Therefore, we need to explore the correlation between TPL and the features.

Method. The Pearson correlations were used to measure the correlation of TPL with the four types of features quantitatively. A Pearson correlation coefficient (r) between two variables X and Y is defined as

$$r = \frac{E(XY) - E(X)E(Y)}{\sqrt{E(X^2) - E^2(X)}\sqrt{E(Y^2) - E^2(Y)}}. \tag{19}$$

Generally speaking, two variables are highly correlated when $|r| \geq 0.6$, and they are moderately correlated when $0.6 \geq |r| \geq 0.4$. Otherwise, they are weakly correlated.

Result. The Pearson correlation coefficients of TPL with the 16 features are listed in Table 4, where the coefficients are marked with a symbol '*' if $|r| \geq 0.6$.

Table 4 shows that, in both Eclipse and Mozilla, there are significantly negative Pearson correlations between TPL (l) and four features (i.e. W_P, W_C, E_P, and E_T). However, the Pearson correlation coefficients between l and the other features vary from project to project. For example, l is highly correlated with N_D in Eclipse, but they are weakly correlated in Mozilla. To address this problem, if a Pearson correlation coefficient between l and a feature in either Eclipse or Mozilla is marked with the symbol '*', the feature is highly correlated with l in this study. According to this definition, five features (i.e. N_D, N_{ID}, N_{OD}, N_{DH}, and O_T) are also considered to be highly correlated with l.

Based on the results of Table 4, a few interesting findings on the nine features that increase TPL can be drawn. First, bugs are likely to be tossed to developers with significant degree centrality. Second, usually, developers who work at different tossing steps belong to a different workspace. Second, usually, developers who work at an arbitrary tossing step belong to a different workspace. Third, sometimes the topics of bug reports do not match with the working themes of assigned developers. Fourth, the proportion of tossers in the developer collaboration network increases when TPL increases.

> *Answer to RQ1: Nine features, N_D, N_{ID}, N_{OD}, O_T, N_{DH}, W_P, W_C, E_P, and E_T, are highly correlated with the change of TPL.*

RQ2. Which developer features are crucial for the change of TPL?

Motivation. Although the Pearson correlation coefficient measures the linear association between each feature under discussion and TPL, we are still unable to determine which main features result in the change in the length of tossing paths based only on the answer to *RQ1*.

Table 4 Pearson correlation coefficients between TPL and the four types of features. NA denotes that there is no correlation between the given feature and TPL

Corr	Eclipse	Mozilla	Significant
$l \sim N_D$	0.878^*	0.371	Y
$l \sim N_{ID}$	0.8790^*	0.352	Y
$l \sim N_{OD}$	0.862^*	0.458	Y
$l \sim N_C$	NA	0.218	N
$l \sim N_{IC}$	NA	NA	N
$l \sim N_{OC}$	NA	NA	N
$l \sim N_{DB}$	−0.045	0.227	N
$l \sim N_{UB}$	−0.002	0.304	N
$l \sim N_{DH}$	0.806^*	0.545	Y
$l \sim N_{UH}$	0.378	−0.031	N
$l \sim W_C$	-0.903^*	-0.918^*	Y
$l \sim W_P$	-0.743^*	-0.846^*	Y
$l \sim E_P$	-0.669^*	-0.739^*	Y
$l \sim E_T$	-0.938^*	-0.657^*	Y
$l \sim O_P$	0.034	0.250	N
$l \sim O_T$	0.820^*	0.349	Y

Table 5 Coefficients and significance of independent variables in Eclipse (above the bold line) and Mozilla (under the bold line)

Variable	*SC*	*p*-value	*In*(%)
N_D	0.790	0.033940^*	2.181
E_T	−8.640	0.000433^{***}	13.90
W_P	−3.628	0.007451^{**}	4.74
N_{ID}	0.934	0.018989^*	3.031
W_C	−1.007	0.017484^*	3.168
4 E_T	−3.313	$1.80 \times 10^{-5***}$	13.87
W_P	−3.010	0.00518^{**}	2.98
W_C	−2.917	0.00126^{**}	4.58
N_D	0.829	$9.64 \times 10^{-6***}$	2.05

P-value is marked with '***' when it is <0.001, '**' for *p*-value <0.01, and '*' for *p*-value <0.05.

Method. For this research question, we model the relationship between TPL and the nine features using a multivariable regression model and identify the key features that result in long tossing paths by analysing their relative importance. In the regression analysis, the nine features are called independent variables (or predictors), and the dependent variable (or responser) refers to TPL *l*.

Result. The best regression model fitted for Eclipse data is presented in the following equation:

$$l = 5.788N_D - 16.419E_T - 15.080W_P + 12.727N_{ID} - 13.265W_C + 21.048. \quad (20)$$

Note that the model was obtained by the subsets regression method and selected by meeting three criteria (i.e. the adjusted R^2 value is >0.95, low Mallow's Cp, and a minimum number of independent variables). The significance level of each independent variable, denoted by *p*-value, is presented above the bold line in Table 5. On the one hand, the adjusted R^2 is 0.9705 in this model, which indicates that the five independent variables, N_D, N_{ID}, E_T, W_P, and W_C, explain 97.05% of the variance of the dependent variable. On the other hand, the *p*-values are <0.05 in Table 5, which suggests that the five features are statistically significant variables for the model. According to the signs of the five features in (20), *l* will increase with the increase of N_D and N_{ID} and decrease with the increase of E_T, W_P, and W_C.

To further determine the relative importance of independent variables, two statistical indicators, standardised regression coefficient and incremental impact on *R*-squared (R^2), were used in our experiment. The former estimates the mean change of the dependent variable based on a standard deviation change in the independent variable in question while holding other independent variables constant in a regression model. The latter calculates the increase of R^2 each independent variable produces when it is added to the regression model that already contains all of the other independent variables. In statistics, independent variables are usually more relevant when they have larger absolute standardised regression coefficients and larger increases in R^2.

As shown in Table 5, E_T explains the greatest amount of variance of TPL regarding the two indicators. In this model (see (20)), the criteria of *SC* and *In* generate the same conclusion on the relative importance of the five independent variables, and they are sorted in descending order by relative importance: E_T, W_P, W_C, N_{ID}, and N_D.

Similarly, the best model fitted for Mozilla data is presented in (21), and the adjusted R^2 is 0.959. As shown in Table 5, according to the criterion of *SC*, E_T has a standardised coefficient with the largest absolute value, followed by W_P, W_C, and N_D. The result obtained by using the criterion of *In* presents a similar order of these features, i.e. E_T contributes the most to the model, followed by W_C, W_P, and N_D

$$l = -22.860E_T - 13.230W_P - 18.70W_C + 8.000N_D + 79.138. \quad (21)$$

Answer to RQ2: Four features, E_T, W_P, W_C, and N_D, are the key factors affecting the change of TPL, among which E_T is the most important one.

Table 6 Experimental feature groups

Group	Features of Developers	Features of bug reports
a1	$\{E_T, W_P, W_C\}$	—
a2	$\{N_D, E_T, W_P, W_C\}$	—
a3	$\{N_D, N_C, N_{UB}, O_P\}$	—
b1	—	{*topic*, *product*, *component*}
b2	—	{*summary*, *description*, *comment product*, *component*}

RQ3. Is there a minimum feature subset that determines the change of TPL?

Motivation. On the one hand, it is an accepted practice that we can obtain a satisfactory (or comparable) result with fewer features, thus leading to lower training cost. On the other hand, multicollinearity is a common phenomenon in multiple regression models. Therefore, those highly correlated features that provide redundant information can be further removed, so as to form a minimum feature subset.

Method. The VIF tells whether the multicollinearity problem exists in a regression model. A VIF of 1 means that there is no correlation between a given predictor and the remaining predictors, and, if a VIF is >10, this indicates that the multicollinearity problem exists. In this study, the minimum subset of developer features can be obtained by removing those features with high VIF values. The removal process ends until the VIF values of all the remaining features are <10.

Result. By using the method mentioned above, we removed N_{ID} and N_D one by one from the model (see (20)), so as to make sure that the VIF values of the rest of the independent variables are <10. Hence, the minimum subset of developer features in Eclipse includes only E_T, W_P, and W_C. Also, we recalculated p-value and the adjusted R^2 after the removal process ends. Compared with the value of R^2 in Table 5), all the p-values are <0.001, and the adjusted R^2 value reaches 0.933, i.e. the loss of R^2 is 4.2%. Similarity, for Mozilla, the minimum feature subset of the model (see (21)) also includes E_T, W_P, and W_C, and the p-values of the obtained three features are <0.002. The adjusted R^2 is 0.912 after the removal of N_D, i.e. the loss of R^2 is 4.7%.

> *Answer to RQ3: Three features, E_T, W_P, and W_C, are the core features that have a significant effect on the change of TPL.*

RQ4. Can the feature sets obtained in *RQ2* and *RQ3* contribute a better performance on developer recommendation?

Motivation. In a statistical sense, both the two sets of features obtained in *RQ2* and *RQ3* have significant effects on reducing the length of tossing paths. However, their practical applications in automatic bug triage are yet to be tested. As a result of feature engineering for ML-based approaches to developer recommendation, we need to carefully examine the contributions of our work to building quality classifiers (or called recommenders) which can achieve better prediction performance.

Method. To compare the impacts of different groups of features on developer recommendation, here we elaborately design a few controlled experiments based on these groups, including the following steps: *feature groups design*, *developer recommenders training*, *evaluation metrics selection*, and *statistical comparative analysis*. First, groups a1 and a2 (see Table 6) contain only developer features, and groups a3, b1, and b2 (see Table 6) contain the most commonly-used developer or textual features employed in many previous studies (see Table 1). Note that group a1 includes the three core features obtained in *RQ3*, group a2 is composed of the four key features achieved in *RQ2*, and group b1 contains the corresponding features of bug reports compared with group a1. Second, according to the five feature groups, we then train several developer recommenders using six common classification algorithms, i.e. decision tree (DT), SVM, NB, logistic regression (LR) with the stochastic gradient descent training, K-nearest neighbours (KNN), and random forest (RF). They were implemented by using an open-source Python library sklearn [http://scikit-learn.org/stable/index.html] with default settings. Third, three frequently-used metrics, accuracy, precision, and recall, are used to measure the performance of developer recommenders, and their definitions, please refer to [43]. Fourth, the Wilcoxon signed-rank test and an effect size (cliff's delta) are utilised to conduct an in-depth statistical comparative analysis.

Result. We divided the data sets of Eclipse and Mozilla sorted in chronological order into training and validation sets. Here, we present an example of an 80:20 split. As shown in Table 7, for both Eclipse and Mozilla, in most cases SVM outperforms the other five classification algorithms regarding the three metrics, regardless of feature groups and the number of recommended developers. Moreover, these six ML algorithms can achieve better results when recommending more suitable developers. For example, the highest accuracy, precision, and recall (at top 5) are 0.909, 0.598, and 0.661, respectively, for group a2 in Eclipse, and they are 0.774, 0.519, and 0.568, respectively, for the group in Mozilla. Note that because the results at top 1 are the same with and without a TG, they are omitted in this study like [4].

Since the six classification algorithms achieve the best accuracy at top 5, we then analyse the impact of the percentage of training data on prediction results when recommending the top five developers. In each of Figs. 3 and 4, five curves with different colour represent the corresponding five feature groups. The X-axis is the percentage of training samples, ranging from 50 to 90% with a step value of 2%, and the Y-axis represents a given evaluation metric. Accuracy, precision, and recall are placed in the first, second, and third column, respectively. Generally speaking, for each of the six classification algorithms, the five curves of prediction results increase slightly with an increase in the size of training samples. Obviously, the performance of group a1 is better than those of groups b1, a3 and b2. Also, the prediction results of groups a1 and a2 are very similar regarding the three evaluation metrics in most cases.

The Wilcoxon signed-rank test and Cliff's delta effect size are employed to compare the results of a1 and the other four feature groups quantitatively, and the comparisons of 21 predictions (with different percentages of training data) at top 5 are shown in Table 8. The Wilcoxon signed-rank test determines whether results of two groups are from the same distribution, and there is no significant difference between the results of the two groups when the p-value (p_v) of the Wilcoxon signed-rank test is >0.05 (i.e. a significance threshold in our experiment). The Cliff's delta (δ) measures how often the values in one distribution are larger than the values in a second distribution, and the difference is interpreted small and can usually be ignored when $\| \delta \| < 0.4$. Note that the value of δ is negative indicates that the result on the right-hand side is better than that on the left-hand side.

Table 8 shows that except for the four cases using DT and LR algorithms, the classifiers trained using group a1 perform better than those trained by groups b1, a3, and b2, according to the small p-values ($p_v < 0.05$) and large Cliff's delta values ($\delta > 0.6$) for comparisons a1Vb1, a1Va3 and a1Vb2. As to comparison a1Va2, the results of accuracy, precision, and recall for groups a1 and a2 are close because more than 60% (23/36) of the p-values are >0.05, which implies that there is no significant difference between groups a1 and a2. Considering the absolute values of δ in these 23 cases are <0.4, the difference between groups a1 and a2 could be ignored, especially for NB and LR in Eclipse and for NB, KNN, and RF in Mozilla.

> *Answer to RQ4: The feature sets we obtained can contribute to a better performance on developer recommendation across six common classification algorithms; compared with the set of the main features, the minimum feature subset has a similar impact on developer recommendation; and SVM is the best classification algorithm for the two sets.*

5.2 Discussion

Although the main goal of this study is to identify those developer features that have a significant impact on the change of TPL, a by-product of our work is that the feature sets we obtained can

Table 7 Prediction results of six ML algorithms for automatic bug triage in Eclipse and Mozilla with an 80:20 split of training and test sets. The best results on accuracy, precision, and recall are marked in boldface

			Eclipse						Mozilla					
			DT	NB	KNN	LR	SVM	RF	DT	NB	KNN	LR	SVM	RF
Top 2	a1	accuracy	0.388	0.599	0.594	0.608	**0.666**	0.536	0.215	0.370	0.352	0.411	**0.490**	0.371
		precision	0.170	0.191	**0.406**	0.191	0.381	0.282	0.101	0.126	0.204	0.132	**0.335**	0.249
		recall	0.211	0.259	0.384	0.238	**0.421**	0.238	0.089	0.102	0.182	0.103	**0.311**	0.217
	b1	accuracy	0.256	0.446	0.430	0.453	**0.509**	0.324	0.136	0.226	0.306	0.278	**0.334**	0.225
		precision	0.149	0.162	0.263	0.156	**0.270**	0.156	0.087	0.088	**0.128**	0.090	0.110	0.098
		recall	0.119	0.139	0.249	0.145	**0.275**	0.157	0.091	0.094	0.116	0.097	**0.121**	0.103
	a2	accuracy	0.472	0.598	0.646	0.610	**0.761**	0.604	0.216	0.410	0.460	0.425	**0.565**	0.533
		precision	0.175	0.192	**0.432**	0.193	0.412	0.28	0.101	0.128	0.207	0.135	**0.365**	0.250
		recall	0.211	0.259	0.393	0.251	**0.429**	0.238	0.091	0.102	0.185	0.106	**0.346**	0.219
	a3	accuracy	0.176	0.084	0.177	0.094	**0.298**	0.263	0.104	0.082	0.181	0.122	**0.217**	0.131
		precision	0.062	0.059	0.051	0.045	**0.211**	0.120	0.069	0.049	0.097	0.049	**0.108**	0.086
		recall	0.062	0.030	0.060	0.040	**0.225**	0.113	0.068	0.068	**0.093**	0.061	0.091	0.090
	b2	accuracy	0.350	0.423	0.520	0.495	**0.662**	0.460	0.185	0.217	0.300	**0.409**	0.346	0.274
		precision	0.167	0.090	0.354	0.190	**0.378**	0.248	0.088	0.085	0.179	0.088	**0.183**	0.158
		recall	0.171	0.142	0.290	0.194	**0.360**	0.231	0.082	0.089	0.161	0.089	**0.174**	0.170
Top 3	a1	accuracy	0.458	0.657	0.673	0.678	**0.728**	0.608	0.312	0.424	0.469	0.469	**0.585**	0.430
		precision	0.362	0.415	0.598	0.378	**0.662**	0.485	0.311	0.360	0.438	0.370	**0.561**	0.446
		recall	0.272	0.296	0.469	0.281	**0.541**	0.315	0.132	0.159	0.242	0.248	**0.410**	0.267
	b1	accuracy	0.326	0.504	0.509	0.523	**0.571**	0.396	0.203	0.280	0.378	0.336	**0.389**	0.284
		precision	0.320	0.403	0.462	0.334	**0.534**	0.337	0.208	0.300	**0.354**	0.304	0.347	0.303
		recall	0.182	0.187	0.313	0.191	**0.389**	0.201	0.138	0.114	**0.178**	0.144	0.168	0.183
	a2	accuracy	0.542	0.656	0.725	0.68	**0.823**	0.626	0.340	0.428	0.532	0.462	**0.617**	0.593
		precision	0.367	0.416	0.604	0.392	**0.696**	0.517	0.313	0.361	0.439	0.370	**0.618**	0.449
		recall	0.272	0.296	0.467	0.286	**0.551**	0.315	0.133	0.161	0.246	0.225	**0.416**	0.268
	a3	accuracy	0.246	0.142	0.256	0.164	**0.360**	0.335	0.171	0.136	0.253	0.180	**0.272**	0.190
		precision	0.283	0.206	0.251	0.230	**0.495**	0.357	0.248	0.280	0.279	0.242	**0.331**	0.313
		recall	0.094	0.101	0.108	0.077	**0.350**	0.170	0.108	0.114	**0.171**	0.102	0.148	0.159
	b2	accuracy	0.420	0.481	0.643	0.565	**0.724**	0.532	0.252	0.271	0.372	**0.467**	0.401	0.333
		precision	0.325	0.276	0.492	0.371	**0.612**	0.433	0.288	0.302	0.373	0.299	**0.388**	0.359
		recall	0.227	0.201	0.351	0.216	**0.467**	0.271	0.131	0.129	0.226	0.169	**0.229**	0.243
Top 4	a1	accuracy	0.530	0.707	0.746	0.755	**0.813**	0.694	0.391	0.609	0.538	0.545	**0.620**	0.615
		precision	0.314	0.358	0.558	0.332	**0.590**	0.422	0.255	0.292	0.380	0.300	**0.515**	0.399
		recall	0.332	0.366	0.525	0.333	**0.605**	0.375	0.188	0.233	0.284	0.211	**0.477**	0.344
	b1	accuracy	0.398	0.554	0.582	0.600	**0.656**	0.482	0.263	0.343	0.443	0.412	**0.464**	0.349
		precision	0.315	0.334	0.402	0.292	**0.487**	0.289	0.260	0.239	**0.313**	0.237	0.285	0.263
		recall	0.226	0.251	0.382	0.259	**0.457**	0.244	0.164	0.167	**0.244**	0.188	0.241	0.261
	a2	accuracy	0.614	0.706	0.798	0.757	**0.887**	0.762	0.460	0.649	0.597	0.538	**0.692**	0.658
		precision	0.319	0.359	0.564	0.346	**0.644**	0.421	0.257	0.293	0.382	0.300	**0.645**	0.400
		recall	0.332	0.366	0.523	0.338	**0.615**	0.375	0.189	0.233	0.296	0.207	**0.513**	0.345
	a3	accuracy	0.318	0.192	0.329	0.241	**0.445**	0.421	0.231	0.199	0.318	0.256	**0.347**	0.255
		precision	0.223	0.196	0.215	0.188	**0.431**	0.297	0.212	0.206	0.248	0.194	**0.274**	0.236
		recall	0.148	0.168	0.173	0.142	**0.407**	0.222	0.173	0.166	**0.250**	0.175	0.218	0.205
	b2	accuracy	0.492	0.531	0.716	0.642	**0.809**	0.618	0.312	0.334	0.437	**0.543**	0.476	0.398
		precision	0.302	0.227	0.515	0.321	**0.578**	0.406	0.245	0.250	0.308	0.255	**0.329**	0.309
		recall	0.230	0.249	0.400	0.285	**0.524**	0.327	0.198	0.193	0.280	0.210	0.270	**0.309**
Top 5	a1	accuracy	0.587	0.766	0.814	0.799	**0.892**	0.798	0.489	0.567	0.661	0.628	**0.702**	0.580
		precision	0.282	0.332	0.536	0.284	**0.564**	0.410	0.230	0.290	0.353	0.289	**0.503**	0.357
		recall	0.386	0.409	0.589	0.402	**0.651**	0.413	0.255	0.291	0.331	0.272	**0.532**	0.416
	b1	accuracy	0.455	0.613	0.650	0.644	**0.735**	0.544	0.340	0.423	0.507	0.495	**0.546**	0.434
		precision	0.244	0.336	0.378	0.269	**0.467**	0.258	0.169	0.238	0.281	0.217	**0.290**	0.216
		recall	0.269	0.295	0.414	0.329	**0.520**	0.291	0.210	0.235	0.299	0.229	**0.304**	0.303
	a2	accuracy	0.671	0.765	0.866	0.801	**0.909**	0.824	0.537	0.571	0.669	0.621	**0.774**	0.743
		precision	0.287	0.333	0.542	0.298	**0.598**	0.442	0.232	0.291	0.354	0.289	**0.519**	0.358
		recall	0.386	0.409	0.587	0.415	**0.661**	0.413	0.256	0.291	0.331	0.273	**0.568**	0.419
	a3	accuracy	0.375	0.251	0.397	0.285	**0.524**	0.483	0.308	0.279	0.382	0.339	**0.429**	0.340
		precision	0.207	0.179	0.205	0.179	**0.411**	0.265	0.205	0.208	0.216	0.188	**0.266**	0.255
		recall	0.196	0.213	0.206	0.194	**0.473**	0.291	0.230	0.231	**0.324**	0.244	0.279	0.283
	b2	accuracy	0.549	0.590	0.784	0.686	**0.888**	0.680	0.389	0.414	0.501	**0.626**	0.558	0.483
		precision	0.253	0.180	0.497	0.306	**0.549**	0.406	0.200	0.248	0.284	0.224	**0.348**	0.290
		recall	0.293	0.289	0.514	0.338	**0.589**	0.375	0.231	0.242	0.318	0.264	0.335	**0.389**

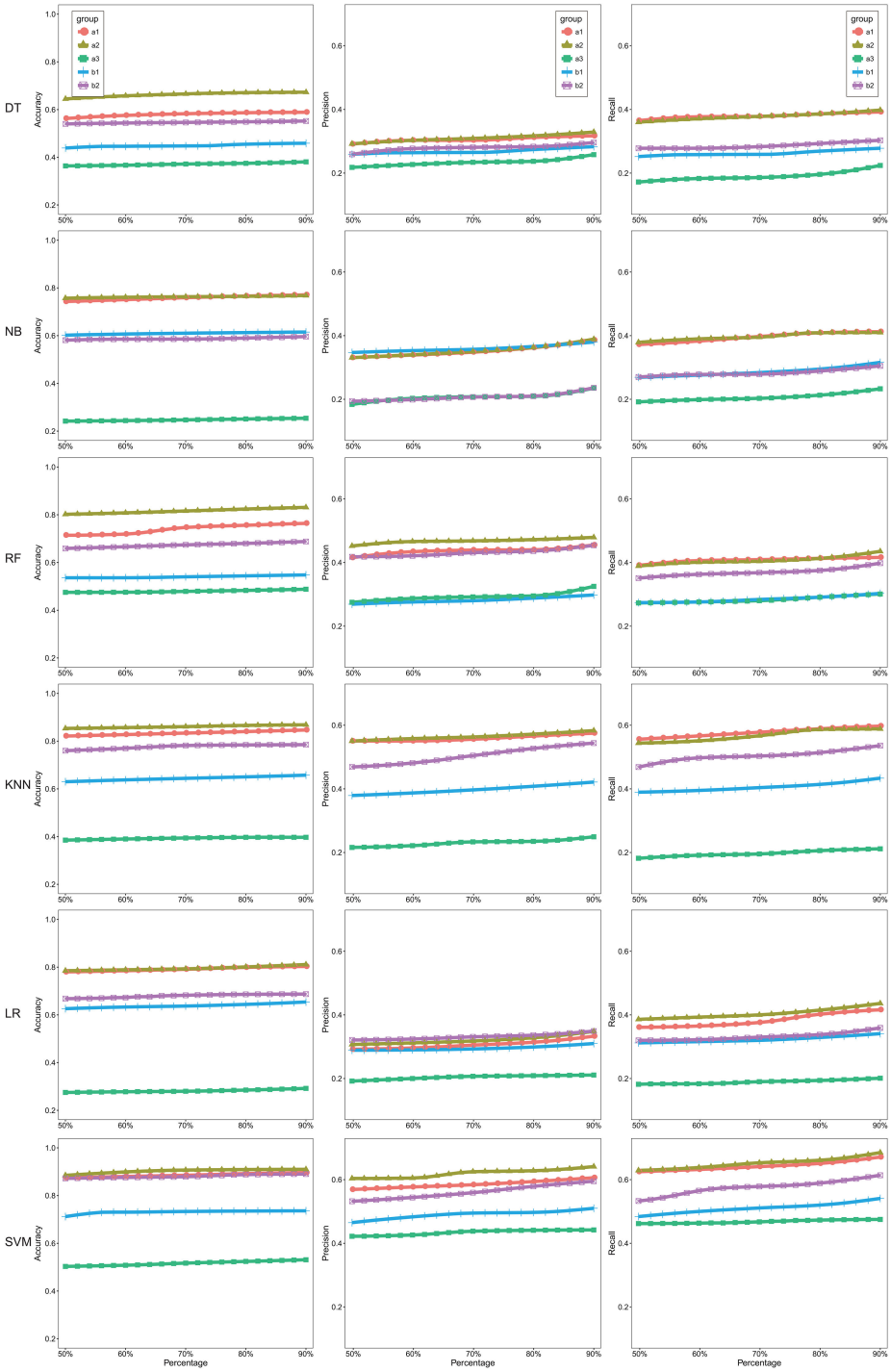

Fig. 3 *Prediction results (at top 5) by the six classification algorithms with different percentages of training samples in Eclipse*

contribute to a better performance on developer recommendation, which is very useful for automatic bug triage. Some interesting findings and implications for software engineering research and practice are listed below.

According to the results shown in Tables 7 and 8, the classifiers trained by developer features outperform those trained using the textual information of bug reports. This finding embodies people-centred ideas in the process of software development and maintenance. Although the textual information of bug reports has been widely used in previous studies in the field of automatic bug triage, such textual information has two disadvantages that deserve attention. First, the cost of processing time for large amounts of text is much higher than that of developer features. For example, for simple classification algorithms like DT and NB, it takes about 170 times longer to train a classifier using group b2. Second, the diversity of natural language processing (NLP) techniques may lead to the low repeatability of experimental results. For example, if we use different dimensionality reduction techniques such as the singular value decomposition and word embedding, the prediction results for groups b1 and b2 are likely to change a bit.

The hints of the minimum feature subset we obtained for empirical software engineering are twofold. On the one hand, to reduce TPL and improve the accuracy of developer recommendations, the most important factor is developer's working theme. That is, a recommender should give priority to the similarity between the topic(s) of a given bug report and the working theme of a possible developer. A significant similarity in semantics indicates a high probability that the developer fixes the bug report. On the other hand, developer's workspace (including product and component) is also an important factor to consider. Generally speaking, developers tend to fix bug reports from their familiar source location. This finding is consistent with a few previous studies (see Table 1) that have trained classifiers for developer recommendation using the two features.

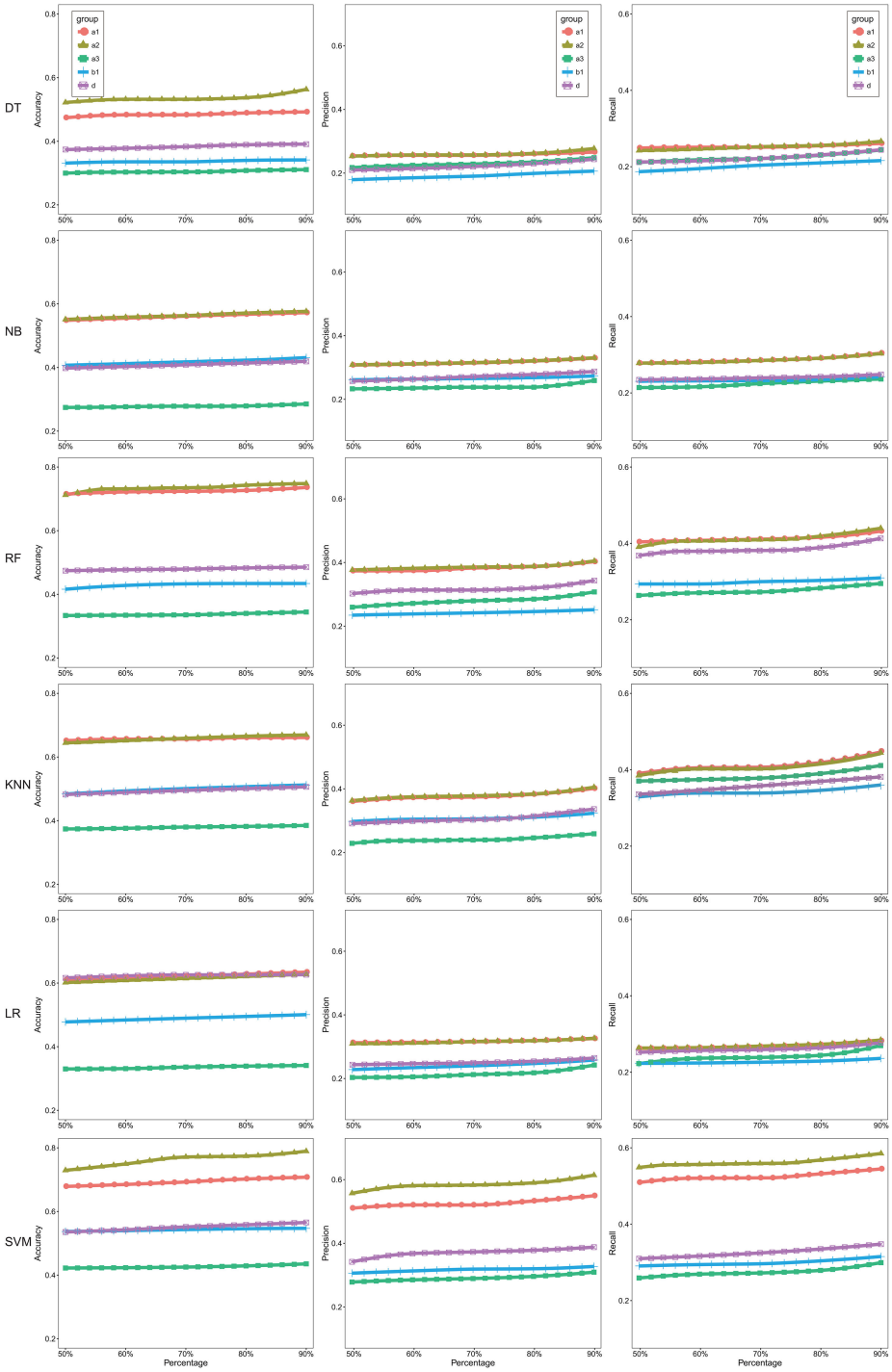

Fig. 4 *Prediction results (at top 5) by the six classification algorithms with different percentages of training samples in Mozilla*

Unlike those previous works based on tossing graphs [4, 14, 15], our results show that the tossing probability between developers is not a significant factor affecting the length of tossing paths, mainly because of a weak (positive) correlation between them. In fact, there are several reassignment rules which have been proposed. One of the easiest ways is to find a developer with high degree (or in-degree) centrality in the corresponding TG [27, 29, 34]. In this study, our results, however, indicate that there are always developers with high degree centrality on long tossing paths, especially in Eclipse. In a statistical sense, this developer factor is more likely to result in the increase of TPL. Influential developers usually have many acquaintances, and the effect of an inappropriate reassignment would be magnified by them, thus leading to cascading errors [44]. Therefore, the reassignment of a bug report to an influential developer should be careful unless the developer can fix it or find a true fixer within few hops on all possible tossing paths. Intelligent recommendation algorithms from influential developers will be our future research.

6 Threats to validity

Although our work obtains some interesting and useful results, there are several threats to the validity of our work that must be explained.

Construct validity: To identify developer factors that affect TPL, we consider a total number of 16 commonly-used features in this study, but in fact, there are also many other features which have been used in automatic bug triage. Our work excludes this potentially relevant feature information from outside this study, possibly leading to defining experimental outcome too narrowly.

Internal validity: There are three main threats to the internal validity of our work. First, various NLP techniques for feature words extraction and parameter tuning of classification algorithms

Table 8 Comparisons of the 21 predictions using the Wilcoxon signed-rank test and Cliff's delta in Eclipse and Mozilla. The *p*-values shown in bold text indicates no significant difference between the two groups in question. The values of Cliff's delta in parentheses denote a better result of group a1 than its rival, while the underlined ones are just the opposite

			DT		NB		KNN		LR		SVM		RF	
			p_v	δ	p_v	δ	p_v	δ	p_v	δ	p_v	δ	p_v	δ
Eclipse	a1Vb1	accuracy	0.000	1.000	0.000	1.000	0.000	1.000	0.000	1.000	0.000	1.000	0.000	1.000
		precision	0.000	0.948	0.000	0.918	0.000	1.000	0.000	1.000	0.000	1.000	0.000	1.000
		recall	0.000	1.000	0.000	1.000	0.000	1.000	0.000	1.000	0.000	1.000	0.000	1.000
	a1Va2	accuracy	0.000	−1.000	**0.696**	(0.073)	0.000	−1.000	**0.325**	−0.179	0.000	−1.000	0.000	−1.000
		precision	**0.107**	−0.293	**0.330**	(0.179)	0.011	−0.456	**0.470**	(0.134)	0.000	−0.927	0.001	−0.569
		recall	**0.372**	(0.163)	**0.070**	−0.329	**0.425**	−0.147	**0.076**	−0.313	**0.053**	−0.370	**0.372**	(0.163)
	a1Va3	accuracy	0.000	1.000	0.000	1.000	0.000	1.000	0.000	1.000	0.000	1.000	0.000	1.000
		precision	0.000	1.000	0.000	1.000	0.000	1.000	0.000	1.000	0.000	1.000	0.000	1.000
		recall	0.000	1.000	0.000	1.000	0.000	1.000	0.000	1.000	0.000	1.000	0.000	1.000
	a1Vb2	accuracy	0.000	1.000	0.000	1.000	0.000	1.000	0.000	1.000	0.000	1.000	0.000	1.000
		precision	**0.186**	−0.240	0.000	1.000	0.000	1.000	**0.538**	(0.113)	0.004	0.506	0.000	0.986
		recall	0.000	1.000	0.000	1.000	0.000	1.000	0.000	0.864	0.000	1.000	0.000	0.698
Mozilla	a1Vb1	accuracy	0.000	1.000	0.000	1.000	0.000	1.000	0.000	1.000	0.000	1.000	0.000	1.000
		precision	0.000	1.000	0.000	1.000	0.000	1.000	0.000	1.000	0.000	1.000	0.000	1.000
		recall	0.001	0.615	0.000	1.000	0.000	1.000	**0.821**	−0.043	0.000	1.000	0.000	1.000
	a1Va2	accuracy	0.000	−0.923	**0.688**	−0.061	**0.507**	−0.023	0.000	−1.000	0.000	−1.000	**0.087**	(0.268)
		precision	**0.173**	−0.247	**0.960**	−0.011	**0.073**	−0.358	**0.073**	−0.351	0.000	−0.914	**0.108**	−0.293
		recall	**0.236**	−0.215	**0.792**	(0.050)	**0.659**	−0.082	0.006	−0.716	0.000	−1.000	**0.069**	(0.356)
	a1Va3	accuracy	0.000	1.000	0.000	1.000	0.000	1.000	0.000	1.000	0.000	1.000	0.000	1.000
		precision	0.000	1.000	0.000	1.000	0.000	1.000	0.000	1.000	0.000	1.000	0.000	1.000
		recall	0.000	1.000	0.000	1.000	0.000	1.000	0.000	0.986	0.000	1.000	0.000	1.000
	a1Vb2	accuracy	0.000	1.000	0.000	1.000	0.000	1.000	**0.065**	−0.338	0.000	1.000	0.000	1.000
		precision	0.000	0.859	0.000	1.000	0.000	1.000	0.000	1.000	0.000	1.000	0.000	1.000
		recall	0.000	0.864	0.000	0.887	0.000	0.909	0.000	0.651	0.000	1.000	0.000	1.000

and LDA may change our results. For example, as stated earlier, the parameters of the six classification algorithms were set by default in sklearn. Second, unlike those previous studies that employed a ten-round incremental framework for experiments [15], we partitioned each of the two data sets of Eclipse and Mozilla into training data and test data with different ratios ranging from 1:1 to 9:1. Therefore, our results are indeed better than those of previous studies on Eclipse and Mozilla.

Statistical conclusion validity: The biggest concern with the statistical conclusion validity of our work is low statistical power. Since the sample size of our experiments with hypothesis tests for *RQ4* is 21, low power occurs when it is small given small effect sizes.

External validity: The results obtained from Eclipse and Mozilla could provide useful suggestions on feature selection for recommending appropriate developers to fix bug reports and promoting bug fixing efficiency by reducing TPL. However, the generality of our results remains unknown for other open-source and closed-source software projects. Besides that, we utilised only six common classification algorithms to train developer recommenders without additional optimisation.

7 Conclusion

To investigate why a few bug reports are repeatedly tossed, in this study we focus on developer factors which affect the length of tossing paths in two popular open-source software projects. Also, we consider four types of developer factors (including 16 features in total), i.e. network centrality, developer workspace, developer expertise, and transmissibility of developers. By the experimental results, the working theme, working product, working component, and degree centrality are four key impact factors to change TPL. Moreover, we also identify a minimum feature subset (including three core features, i.e. working theme, working product, and working component) that contribute largely to the change of TPL.

We then train developer recommenders using different feature groups and ML classification algorithms to predict matchable developers for a given bug report, and the empirical results show that the feature groups obtained in this study can contribute to a better performance regarding the three evaluation metrics. More specifically, the three core features have a similar effect on developer recommendation compared with the four key features.

Our future work includes two aspects: first, we will validate the generality of the feature groups obtained in this study on developer recommendation in other software bug repositories; second, we will train better developer recommenders based on a hybrid set of developer factors and textual information of bug reports using deep artificial neural networks.

8 Acknowledgments

This work was supported by the National Basic Research Program of China (no. 2014CB340404), the National Key Research and Development Program of China (no. 2016YFB0800400), the National Natural Science Foundation of China (nos. 61272111, 61672387, and 61702378), the Wuhan Yellow Crane Talents Program for Modern Services Industry, and the Strategic Team-Building of Scientific and Technological Innovation in Hubei Province.

9 References

[1] Anvik, J., Hiew, L., Murphy, G.C.: 'Who should fix this bug?'. ACM Int. Conf. on Software Engineering (ICSE)'06, New York, USA, 2006, pp. 361–370

[2] Anvik, J.: 'Automating bug report assignment'. ACM Int. Conf. on Software engineering (ICSE)'06, New York, USA, 2006, pp. 937–940

[3] Statistica Inc.: 'Projected revenue of open source software 2008–2020', Statista, 2017. Available at http://www.statista.com/statistics/270805/

[4] Jeong, G., Kim, S., Zimmermann, T.: 'Improving bug triage with bug tossing graphs'. European Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)'09, ACM, New York, USA, 2009, pp. 111–120

[5] Cubraknic, D., Murphy, G.C.: 'Automatic bug triage using text categorization'. International Conference on Software Engineering & Knowledge Engineering (SEKE), KSI Press, Pittsburgh, USA, 2004, pp. 92–97

[6] Ahsan, S.N., Ferzund, J., Wotawa, F.: 'Automatic software bug triage system (bts) based on latent semantic indexing and support vector machine'. Int. Conf. on Software Engineering Advances (ICSEA)'09, IEEE Computer Society, Washington, USA, 2009, pp. 216–221

[7] Lin, Z., Shu, F., Yang, Y., *et al.*: 'An empirical study on bug assignment automation using Chinese bug data'. Empirical Software Engineering and Measurement (ESEM)'09, IEEE Computer Society, Washington, USA, 2009, pp. 451–455

[8] Baysal, O., Godfrey, M.W., Cohen, R.: 'A bug you like: a framework for automated assignment of bugs'. International Conference on Program Comprehension (ICPC), IEEE Computer Society, Washington, USA, 2009, pp. 297–298

[9] Helming, J., Arndt, H., Hodaie, Z., *et al.*: 'Semi-automatic assignment of work items'. International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)'10, Setúbal, Portugal, 2010, pp. 149–158

[10] Anvik, J., Murphy, G.C.: 'Reducing the effort of bug report triage: recommenders for development-oriented decisions', *ACM Trans. Softw. Eng. Methodol.*, 2011, **20**, (3), pp. 10:1–10:35

[11] Alenezi, M., Magel, K., Banitaan, S.: 'Efficient bug triaging using text mining', *J. Softw.*, 2013, **8**, (8), pp. 2185–2190

[12] Shokripour, R., Anvik, J., Kasirun, Z.M., *et al.*: 'Why so complicated? Simple term filtering and weighting for location-based bug report assignment recommendation'. Mining Software Repositories (MSR)'13, Piscataway, NJ, USA, 2013, pp. 2–11

[13] Jonsson, L.J., Borg, M., Broman, D., *et al.*: 'Automated bug assignment: ensemble-based machine learning in large scale industrial contexts', *Empir. Softw. Eng.*, 2015, **21**, (4), pp. 1533–1578

[14] Bhattacharya, P., Neamtiu, I.: 'Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging'. Int. Conf. on Software Maintenance (ICSM)'10, IEEE Computer Society, Washington, USA, 2010, pp. 1–10

[15] Bhattacharya, P., Neamtiu, I., Shelton, C.R.: 'Automated, highly-accurate, bug assignment using machine learning and tossing graphs', *J. Syst. Softw.*, 2012, **85**, (10), pp. 2275–2292

[16] Borgatti, S.P.: 'Centrality and network flow', *Social Netw.*, 2005, **27**, (1), pp. 55–71

[17] Perry, D.E., Stieg, C.S.: 'Software faults in evolving a large, real-time system: a case study', 'In: Sommerville, I., Paul, M. (Eds): *Proceedings of the Fourth European Software Engineering Conference* (Springer, Berlin, Heidelberg, 1993), pp. 48–67

[18] Akila, V., Zayaraz, G., Govindasamy, V.: 'Bug triage in open source systems: a review', *Int. J. Collaborat. Enterp.*, 2014, **4**, (4), pp. 299–319

[19] Zhang, J., Wang, X., Hao, D., *et al.*: 'A survey on bug-report analysis', *Sci. China F, Inf. Sci.*, 2015, **58**, (2), pp. 1–24

[20] Canfora, G., Cerulo, L.: 'Supporting change request assignment in open source development'. ACM Symp. on Applied Computing (SAC)'06, New York, USA, 2006, pp. 1767–1772

[21] Nagwani, N.K., Verma, S.: 'Predicting expert developers for newly reported bugs using frequent terms similarities of bug attributes'. ICT and Knowledge Engineering. IEEE, Bangkok, Thailand, 2012, pp. 113–117

[22] Shokripour, R., Kasirun, Z.M., Zamani, S., *et al.*: 'Automatic bug assignment using information extraction methods'. Advanced Computer Science Applications and Technologies (ACSAT)'12, IEEE Computer Society, Washington, USA, 2012, pp. 144–149

[23] Kagdi, H., Gethers, M., Poshyvanyk, D., *et al.*: 'Assigning change requests to software developers', *J. Softw., Evol. Process*, 2012, **24**, (1), pp. 3–33

[24] Xia, X., Lo, D., Wang, X., *et al.*: 'Accurate developer recommendation for bug resolution'. IEEE Working Conf. on Reverse Engineering (WCRE)'13, Koblenz, Germany, 2013, pp. 72–81

[25] Xia, X., Lo, D., Wang, X., *et al.*: 'Dual analysis for recommending developers to resolve bugs', *J. Softw.: Evol. Process*, 2015, **27**, (3), pp. 195–220

[26] Zhang, T., Lee, B.: 'In: 'An automated bug triage approach: a concept profile and social network based developer recommendation' in Huang, D.-S., Jiang, C., Bevilacqua, V., *et al.* (Eds.): *Proceedings of the Eighth International Conference Intelligent on Computing Technology*, (Springer, Berlin, Heidelberg, 2012), pp. 505–512

[27] Wang, S., Zhang, W., Yang, Y., *et al.*: 'DevNet: exploring developer collaboration in heterogeneous networks of bug repositories'. IEEE Empirical Software Engineering and Measurement (ESEM)'13, Maryland, USA, 2013, pp. 193–202

[28] Park, J., Lee, M., Kim, J., *et al.*: 'Cost-aware triage ranking algorithms for bug reporting systems', *Knowl. Inf. Syst.*, 2015, **48**, (3), pp. 679–705

[29] Zhang, W., Wang, S., Wang, Q.: 'KSAP: an approach to bug report assignment using KNN search and heterogeneous proximity', *Inf. Softw. Technol.*, 2016, **70**, pp. 68–84

[30] Naguib, H., Narayan, N., Brügge, B., *et al.*: 'Bug report assignee recommendation using activity profiles'. Working Conference on Mining Software Repositories (MSR)'13, IEEE Computer Society, Washington, USA, 2013, pp. 22–30

[31] Matter, D., Kuhn, A., Nierstrasz, O.: 'Assigning bug reports using a vocabulary-based expertise model of developers'. Working Conference on Mining Software Repositories (MSR)'09, IEEE Computer Society, Washington, USA, 2009, pp. 131–140

[32] Xie, X., Zhang, W., Yang, Y., *et al.*: 'DRETOM: developer recommendation based on topic models for bug resolution'. International Conference on Predictive Models in Software Engineering (PROMISE)'12, New York, USA, 2012, pp. 19–28

[33] Linaresvasquez, M., Hossen, K., Dang, H., *et al.*: 'Triaging incoming change requests: bug or commit history, or code authorship?', IEEE International Conference on Software Maintenance (ICSM)'12, Trento, Italy, 2012, pp. 451–460

[34] Wu, W., Zhang, W., Yang, Y., *et al.*: 'DREX: developer recommendation with k-nearest-neighbor search and expertise ranking'. Asia-Pacific Software Engineering Conf. (APSEC)'11, IEEE Computer Society, Washington, USA, 2011, pp. 389–396

[35] Zanetti, M.S., Scholtes, I., Tessone, C.J., *et al.*: 'Categorizing bugs with social networks: a case study on four open source software communities'. Int. Conf. on Software Engineering (ICSE)'13, San Francisco, USA, 2013, pp. 1032–1041

[36] Xuan, J., Jiang, H., Ren, Z., *et al.*: 'Automatic bug triage using semi-supervised text classification'. Int. Conf. on Software Engineering & Knowledge Engineering (SEKE)'10, Pittsburgh, USA, 2010, pp. 209–214

[37] Chen, L., Wang, X., Liu, C.: 'Improving bug assignment with bug tossing graphs and bug similarities'. Int. Conf. on Biomedical Engineering and Computer Science (ICBECS)'10, Wuhan, China, 2010, pp. 56–60

[38] Pinzger, M., Nagappan, N., Murphy, B.: 'Can developer-module networks predict failures?'. ACM SIGSOFT Int. Symp. on Foundations of Software Engineering (SIGSOFT FSE)'08, Atlanta, USA, 2008, pp. 2–12

[39] Guo, P.J., Zimmermann, T., Nagappan, N., *et al.*: 'Characterizing and predicting which bugs get fixed: an empirical study of Microsoft Windows'. ACM/IEEE Int. Conf. on Software Engineering (ICSE)'10, Cape Town, South Africa, 2010, vol. 1, pp. 495–504

[40] Hooimeijer, P., Weimer, W.: 'Modeling bug report quality'. Automated Software Engineering (ASE)'07, ACM, New York, USA, 2007, pp. 34–43

[41] Opsahl, T., Agneessens, F., Skvoretz, J.: 'Node centrality in weighted networks: generalizing degree and shortest paths', *Social Netw.*, 2010, **32**, (3), pp. 245–251

[42] Blei, D.M., Ng, A.Y., Jordan, M.I.: 'Latent Dirichlet allocation', *J. Mach. Learn. Res.*, 2003, **3**, (2003), pp. 993–1022

[43] Makhoul, J., Kubala, F., Schwartz, R., *et al.*: 'Performance measures for information extraction'. Proc. Darpa Broadcast News Workshop, Herndon, VA, USA, 1999, pp. 249–252

[44] Motter, A.E., Lai, Y.C.: 'Cascade-based attacks on complex networks', *Phys. Rev. E, Stat. Nonlinear Soft Matter Phys.*, 2002, **66**, (2), p. 065102