Effective Bug Triage based on Historical Bug-Fix Information

Hao Hu, Hongyu Zhang, Jifeng Xuan, Weigang Sun

▶ To cite this version:

Hao Hu, Hongyu Zhang, Jifeng Xuan, Weigang Sun. Effective Bug Triage based on Historical Bug-Fix Information. ISSRE - The 25th IEEE International Symposium on Software Reliability Engineering, 2014, IEEE, Nov 2014, Naples, Italy. hal-01087444

HAL Id: hal-01087444

https://inria.hal.science/hal-01087444

Submitted on 26 Nov 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Effective Bug Triage based on Historical Bug-Fix Information

Hao Hu
Tsinghua University
Beijing, China
haohu.th@gmail.com

Hongyu Zhang
Microsoft Research
Beijing, China
honzhang@microsoft.com

Jifeng Xuan
INRIA Lille – Nord Europe
Lille, France
jifeng.xuan@inria.fr

Weigang Sun
Inspur Worldwide Services Ltd.
Beijing, China
hauhotsun@gmail.com

***Abstract*—For complex and popular software, project teams could receive a large number of bug reports. It is often tedious and costly to manually assign these bug reports to developers who have the expertise to fix the bugs. Many bug triage techniques have been proposed to automate this process. In this paper, we describe our study on applying conventional bug triage techniques to projects of different sizes. We find that the effectiveness of a bug triage technique largely depends on the size of a project team (measured in terms of the number of developers). The conventional bug triage methods become less effective when the number of developers increases. To further improve the effectiveness of bug triage for large projects, we propose a novel recommendation method called BugFixer, which recommends developers for a new bug report based on historical bug-fix information. BugFixer constructs a Developer-Component-Bug (DCB) network, which models the relationship between developers and source code components, as well as the relationship between the components and their associated bugs. A DCB network captures the knowledge of "who fixed what, where". For a new bug report, BugFixer uses a DCB network to recommend to triager a list of suitable developers who could fix this bug. We evaluate BugFixer on three large-scale open source projects and two smaller industrial projects. The experimental results show that the proposed method outperforms the existing methods for large projects and achieves comparable performance for small projects.**

***Keywords*—bug report assignment; bug triage; developer recommendation; bug repository**

I. INTRODUCTION

For a complex and popular software system the project team could receive a large number of bug reports. Once a bug is reported, it is typically recorded in a bug tracking system, and is assigned to a developer to resolve. Bug report assignment is an important phase of bug triage, since inappropriate developers may delay the bug resolution time. Empirical studies on Eclipse and Mozilla [11] show that 37%-44% of bugs have been re-assigned (tossed) at least once to another developer. One of the common reasons for bug tossing is that bugs are assigned to developers by mistake. As a result, the bug-fixing time is prolonged.

Current practice of bug triage is largely a manual process. The triager (the developer who triages bugs) needs to examine a bug report and decide who has the expertise in resolving the bug. Such a process could be a tedious and costly process when the number of bug reports is large. For example, more than 333,000 bugs were reported for Eclipse project from Oct. 2001 to Dec. 2010, in average 99 bugs every day [28]. Assuming that it took 5 minutes to triage a bug, then over 8 hours would be spent on bug assignment alone. According to Tyler Downer[1], a former Mozilla Community Lead, there were 5934 unconfirmed bugs in the shipping version of Firefox 4, among them 2598 had not been touched over the past 150 days since the launch of Firefox 4. In a post that explains the reasons for his departure, he voices frustration with Mozilla that no attention is paid to triage bugs.

Many researchers have proposed methods for automated bug report assignment. These methods are largely based on text categorization and machine learning techniques [1], [3], [7], which treat bug reports as documents and the associated developers of bugs as categories, and apply machine learning techniques to assign new bug reports to developers. For example, Čubranić and Murphy [7] proposed a Naïve Bayes based method to predict developers who should fix the bug based on the bug's description. They evaluated 15,670 Eclipse bugs and can correctly predict 30% of the bug report assignments. However, to our knowledge, these methods have not been widely applied in practice. Furthermore, their accuracy could be further improved.

For some software projects, especially the small to medium projects that adopt an agile development process (e.g., Scrum [18]), the team sizes are typically small. An empirical study of 109 different Scrum teams found that the teams have 4 to 18 members [6]. For other projects, especially large-scale open source projects such as Eclipse, the team size is often large, even up to hundreds or thousands. In this paper, we evaluate the effectiveness of conventional bug triage methods on two medium-sized industrial projects of a company ABC (the actual company name is not disclosed for the sake of confidentiality). Our results show that the conventional bug triage methods are effective for these industrial projects. We also analyze the impact of team size on the effectiveness of bug triage methods, and find that the conventional bug triage methods become less effective when the number of developers increases.

[1] http://www.somethingawful.com/news/bugs-of-firefox/

To improve the effectiveness of bug triage methods for large projects, in this paper we propose BugFixer, an automated bug report assignment method based on historical bug-fix information. BugFixer constructs a Developer-Component-Bug (DCB) network, which models the relationship between the developers and the source code components they worked on, as well as the relationship between the components and the associated bugs. For a new bug report, BugFixer computes the similarity between this bug and the existing bugs, calculates the relevance between the new bug and developers, and recommends a ranked list of developers who could fix this bug. BugFixer also applies a new preprocessing technique to tokenize the bug reports, and computes the similarity between two bug reports using the Vector Space Model (VSM), taking into consideration both bug report and source file information. With BugFixer, a human bug triager can examine the recommendation list and assign the new bug report to an appropriate developer.

We have evaluated BugFixer on three large-scale open source projects (namely Eclipse, Mozilla, and Netbeans), as well as two smaller industrial projects. In total, we have experimented on 9,779 bug reports. The results show that for smaller projects, BugFixer achieves comparable performance as the conventional bug triage methods do (such as those based on SVM [1], [3], [7]). BugFixer is more effective in recommending bug fixers for larger projects. For example, for 42.36% bugs in Eclipse, their fixers are correctly ranked by BugFixer in the top 1 of the recommendation list; for 73.85% bugs in Eclipse, their fixers are correctly ranked in top 5. The experimental results show that BugFixer is suitable for projects of different sizes and is more effective than the conventional bug triage methods. We believe that our method can help accelerate the process of bug triage in practice.

The remainder of this paper is organized as follows: Section II introduces the bug triage background. Section III describes our case study on applying bug triage methods to industrial projects. Section IV describes the proposed BugFixer method for large projects. Section V presents our experimental design for evaluating BugFixer and shows the experimental results. We discuss the proposed approach and threats to validity in Section VI. Section VII surveys related work followed by Section VIII that concludes this paper.

II. BACKGROUND

A. Bug Triage

Bugs are inevitable in software development. In daily testing and maintenance, bug reports are accumulated and stored in projects' bug tracking system. A bug tracking system provides a platform for tracking the status of bug reports. Once a bug report is created, the bug begins its lifecycle (e.g., being assigned, resolved, or closed). Through the bug tracking system, developers can also collaborate on reproducing and fixing bugs.

A bug report should be assigned to an appropriate developer before the process of bug fixing [3]. The step of assigning bug reports to developers is called bug triage. The developer who triages bugs is called a triager. For a new bug report, the triager first determines whether this bug report contains sufficient information. For a meaningful bug report, the triager sets the bug severity and assigns it to a developer who is responsible for fixing the bug. If the assigned developer is not able to fix the bug, the bug can be assigned again to another developer. In real-world software projects, the process of manual bug triage could be tedious and time-consuming since the number of bug reports is overwhelming [1], [9].

B. Conventional Bug Triage Methods

Since manual bug triage is time-consuming, automated techniques are proposed to reduce the time cost [1], [3], [7], [13]. Čubranić and Murphy [7] proposed one of the first methods of automated bug triage in 2004. In their work, they employed a text-categorization approach based on the Naïve Bayes classifier and achieved the precision around 30% on Eclipse. The problem of assigning bug reports to developers is mapped to a text categorization problem. That is, a new bug report (a textual document) is classified to a set of candidate developers (categories) based on a classifier, which is trained using historical data. Anvik et al. [3] extended the above text-categorization approach with various classifiers and a recommendation list. Based on the recommendation list, a human triager can make a decision by leveraging the results of automated bug report assignment.

Figure 1 shows a general process of a text-categorization based bug triage method. It mainly has two phases. First, given the historical bug reports in a bug repository, the textual information of bug reports and their associated developers are extracted to train a classifier (e.g., Naïve Bayes or SVM). Second, for a new bug report, the classifier ranks candidate developers and recommends a list of developers to a human triager. The triager can then assign the new bug report to an appropriate developer who can fix the bug.

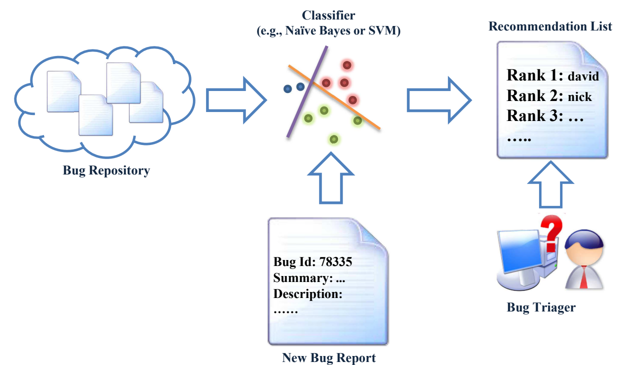


Figure 1. The general process of a bug triage method.

III. BUG TRIAGE - AN INDUSTRIAL CASE STUDY

A. Bug Triage Process of the Company

We study the bug triage process in an IT company ABC, which provides worldwide IT solutions and services. The bug tracking system used in the company is built with Team Foundation Server (TFS) on Microsoft Visual Studio Platform. Figure 2 gives a snapshot of the TFS bug tracking system.

The studied project teams in the company adopt the Scrum software development process. The team members play sever-

Figure 2. A snapshot of the TFS bug tracking system.

al roles such as product owner, scrum master, as well as development team members and test team members. Members in development team are responsible for the product code while test team members are responsible for testing.

In practice, most bugs are filed by the test team but it is acceptable to record bugs by the development team as well. The process of bug triage is easy if a bug is recorded by the development team. Most of such bugs will be directly assigned to the developer who filed it. If a bug report is created by test team members, the process of bug triage is more complicated. After the bug is created, it should first be assigned to the test team leader for review. Then the test team leader decides if the bug is valid and reproducible according to the reproduce steps. Once the bug report is validated, it is assigned to development team leader. The development team leader then reviews the bug and assigns the bug to a developer who has the appropriate knowledge of the bug. This is the common triage process adopted in the company. One problem is that, there might be quite a number of bugs filed, especially when new releases are coming out. In this situation the triage burden is rather heavy.

B. *Studied Projects*

We evaluate the effectiveness of the automated bug triage methods on industrial projects. The TFS bug tracking system used in the company records a list of updated source files if the bug is fixed. In general, when a developer commits bug-fixing code, she/he needs to specify the bug being fixed. Thus the modified source files can be automatically linked to the bug report. We take the developer who committed the bug-fixing changes as the bug fixer, and developed tools to collect the data.

We have collected data from the following two industrial projects from the company:

- EBCP, which is a workflow process management tool. For this project, we studied its 1008 bug reports filed from 3/5/2012 to 9/26/2012, which include 1008 bugs. There have been 19 developers involved in this project.
- DMGR, which is a solution accelerator for Microsoft Server and Cloud platform, enabling visualized upgrading process for IT administrators. For this project, we studied its 686 bug reports filed from 9/3/2010 to 12/20/2012. There have been 11 developers involved in this project.

C. *Evaluating Automated Bug Triage Methods*

We apply the conventional SVM and Naïve Bayes based bug triage methods (as described in Section II-B) to the industrial projects.

We perform a 5-round incremental analysis [28] on the two project datasets. First, we chronologically sort all the bug reports in a project; then, these bug reports are divided into 6 equally-sized folds. We form 5 rounds evaluation with these 6 folds. For the ith round analysis, the first i folds are used as the training set and the $(i+1)$th fold is used as the test set.

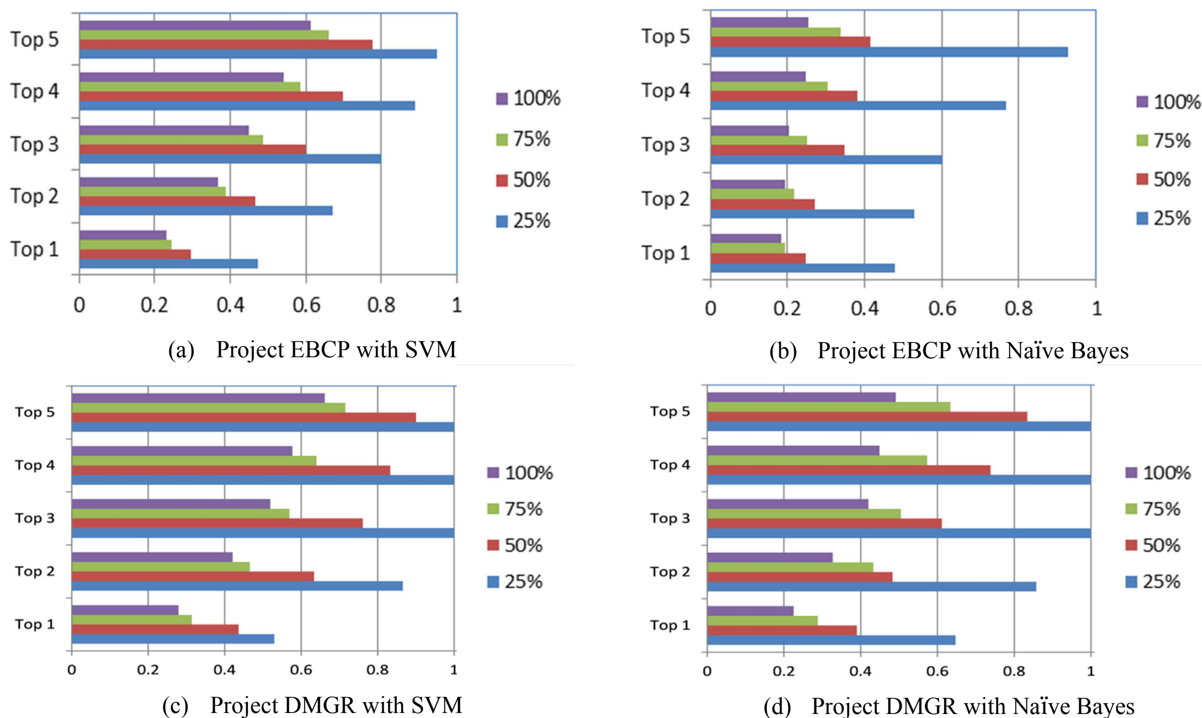
We use the Recall@k metric to evaluate the effectiveness of bug triage. Recall@k is the number of bugs whose associated developer is ranked in the top k (k=1, 3, 5) of the returned results. Given a bug report, if the top k results contain the developer who fixed the bug, we consider the developer is located. The higher the metric value is, the better the performance is.

We show the evaluation results in Table I. For EBCP, the SVM-based bug triage method achieves an average Recall@1 value of 23.19%, which means that for 23.19% bug reports, SVM-based method can successfully recommend their associated developers as top 1. The Recall@5 value is 61.19%, which means that for 61.19% bugs, their developers can be found in the top 5 return results. Similarly, for DMGR, the Recall@1 and Recall@5 values achieved by the SVM-based method are 27.92% and 66.15%, respectively. Overall, the SVM-based results are considered satisfactory, and are much better than the results of the Naïve Bayes based method. These results are consistent with what obtained by others [1], [3].

We also analyze the impact of project team size on the evaluation results. To do so, we evaluate the performance of SVM and Naïve Bayes based bug triage methods with different numbers of developers. For each project, we randomly select 25%, 50%, and 75% of the original developers. Then bug reports associated with these developers are extracted to form new datasets. In this way, we can form another three datasets from the sampling of the original dataset. For each dataset, we evaluate the results with the 5-round incremental learning framework. The experiments are repeated for 10 individual runs and the averages of the 10 runs are taken as the final results. Figure 3 presents the final results of this analysis. We can see that when the number of developers increases, the

TABLE I. RESULTS OF AUTOMATED BUG TRIAGE ON INDUSTRIAL PROJECTS

Project	Rank	SVM	Naïve Bayes
EBCP	Top1	23.19%	18.26%
	Top3	44.86%	20.21%
	Top5	61.19%	25.31%
DMGR	Top1	27.92%	22.54%
	Top3	51.90%	42.03%
	Top5	66.15%	49.15%



(a) Project EBCP with SVM

(b) Project EBCP with Naïve Bayes

(c) Project DMGR with SVM

(d) Project DMGR with Naïve Bayes

Figure 3. The impact of project team sizes on bug triage results.

performance of SVM and Naïve Bayes based bug triage methods decreases. For example, for the SVM results of the EBCP project, the Recall@5 values drops from 94.62% to 61.19% when the number of sampled developers increases from 25% to 100%.

In summary, the findings we obtained from the case study are as follows:

- Conventional bug triage methods, especially the SVM-based method [1], [3], work well for the studied industrial projects.
- However, the performance of conventional bug triage methods drops when the size of project team increases.

To improve the performance of bug triage methods for large projects, we propose a novel method called BugFixer, which will be described in the next section.

IV. BugFixer - The Proposed Approach

We propose BugFixer, an automated bug triage method. The overall structure of BugFixer is shown in Figure 4. BugFixer leverages the bug report information mined from bug tracking systems as well as the bug-fix information mined from the change logs of source code repository. BugFixer then constructs a network called Developer-Component-Bug (DCB) network. In a DCB network, the bugs are linked to the components in which they are fixed; and the developers are linked to the components to which they committed bug-fixing changes. Once a new bug report arrives, BugFixer applies the VSM model to compute the similarity between the new bug report and the previously fixed bug reports. It then performs recommendation over DCB and outputs a ranked list of developers based on the relevance scores between the developers and the new bug report.

A. Bug Report Similarity

1) Preprocessing

Existing bug triage methods extract features (words) from bug summary and description, and use these features as input to a machine learning algorithm to classify the bug reports to potential developers. The conventional preprocessing technique splits text into separate words using delimiters (such as "," and "."), removes common stop words (such as "a" and "the"), and performs stemming. However, some words used in

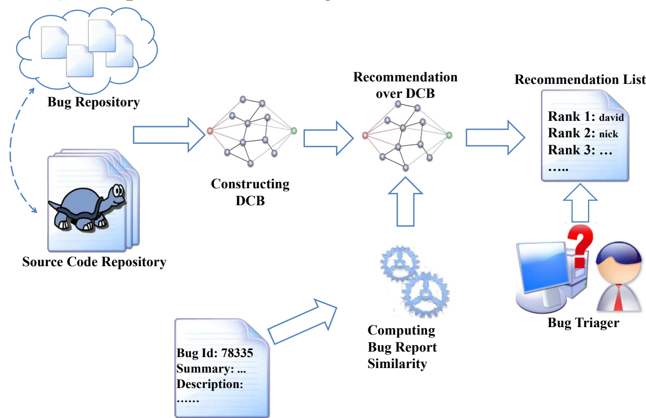


Figure 4. The overall structure of BugFixer.

```
Input:   an original Text
Output: a preprocessed Text
 1 removeStopWord(Text)
 2 for each word in Text
 3     translateToSemanticWord(word)
 4     splitAccordingToConvention(word)
 5     addWordIntoWordSet(word, wordSet)
 6 end
 7 for each word in Text
 8     for each position in word
 9        [word1,word2]=splitWordAccordToPosition(word)
10        if wordSet contains word1 and word2
11             addNewSplitRule(word, word1, word2)
12             removeWord(wordSet, word)
13        end
14     end
15 end
```

Figure 5. The proposed tokenization algorithm.

bug reports are abbreviations or composed words. These words may imply different aspects of bug characteristics. The conventional preprocessing technique has difficulties in handling these words. Therefore, the accuracy of recommending developers could be adversely affected.

We develop a new tokenization algorithm to split these words into single words. For example, in the word "ICVSUIPlugin", the prefix "I" and the suffix "Plugin" will be extracted according to the naming convention of Java. The remaining "CVSUI" will be further split into "CVS" and "UI" as these two words exist in the set of words extracted from all bug reports. Therefore "ICVSUIPlugin" will finally be split into three words "CVS", "UI" and "Plugin" ("I" is considered as a stop word).

Figure 5 illustrates the tokenization algorithm for preprocessing the text in bug reports. We first remove all stop words from the text (Line 1). Then we use WordNet[2] to transform all words into unified forms (Line 3) and split the words according to the commonly-adopted naming conventions such as Java naming convention (Line 4). In this way, we obtain a set of words (*wordSet*). For each word, we split it into two words if possible, by iteratively searching for two potential subwords that exists in the *wordSet* (Lines 7-14).

2) Computing Bug Report Similarity using VSM

BugFixer computes the similarity between two bug reports using Vector Space Model (VSM). Besides the bug report information, we leverage the source file information of previously fixed bugs. A project's source code repository contains logs of bug-fixing changes, from which we can establish links between bug reports and associated source files. The source file information provides extra information about the bugs. For example, the name of a source file (such as *org.eclipse.debug.internal.ui.views.memory.MemoryViewTab.java*) may contain keywords that suggest the characteristics of bugs, and these keywords are supplementary to those in the bug reports. Therefore, analyzing source file information related to a bug report can help identify the similarity among bug reports. Note that we only consider the source file information for previously fixed bugs. For newly arrived bug reports, such information remains unknown and thus is not used.

[2] WordNet, http://wordnet.princeton.edu/.

To compute the similarity between two bug reports, BugFixer first combines the source file name (for fixed bug only), bug summary, and bug description into one text. It then preprocesses the text according to the algorithm described in Figure 5. Finally, VSM is applied to calculate the similarity between the two bug reports.

The similarity function used by VSM is defined as follows:

$$Similarity(b,t) = \cos(b,t) = \frac{\vec{V_b} \bullet \vec{V_t}}{\left|\vec{V_b}\right|\left|\vec{V_t}\right|}$$

where $\vec{V_b}$ and $\vec{V_t}$ are the vectors of term weights for an existing bug report (with source file name) and a new bug report respectively. $\vec{V_b} \bullet \vec{V_t}$ represents the inner product of the two vectors. The weight of vector $\vec{V_b}$ and $\vec{V_t}$ is computed based on tf/idf (term frequency/inverse document frequency).

B. The DCB Network

We build a Developer-Component-Bug (DCB) network to determine the relevance between bug reports and developers and perform recommendation over the network.

1) Constructing DCB

A DCB network is a directed graph, $G_{dcb}=(V_{dcb}, E_{dcb})$, where V_{dcb} is a set of nodes representing the three kinds of nodes (*Developer*, *Component*, *Bug*). A *Developer* is the bug fixer who fixes the bug by committing source code changes to version control system. A *Component* is a group of source code files that achieves certain functionalities. A component could correspond to a source file package or a sub-package. A *Bug* is denoted by a bug report that is comprised of a unique bug id, bug summary, and bug description. E_{dcb} is a set of edges between bugs and components, and between components and developers. The DCB network is constructed using historical bug-fix data obtained by mining software repositories (including bug repository and SVN/CVS logs). It models the relationship between the developers and the source code components they worked on, as well as the relationship between the components and the associated bugs.

In a DCB, each edge (e_{b2c}) between a bug b_i and a component c_j is assigned a weight, as well as each edge (e_{c2d}) between a component c_j and a developer d_k. For e_{b2c}, its weight is set as the number of files that were modified in order to fix a bug. Formally:

$$e_{b2c}(b_i, c_j).weight = count(F_{ij})$$

where F_{ij} is a set of files in the component c_j that were changed to fix the bug b_i. For e_{c2d}, its weight is set as the number of files

in a component to which a developer has committed bug-fixing changes. Formally,

$$e_{c2d}(c_i, d_j).weight = \sum_{k=j_0}^{j_n} e_{b2c}(b_k, c_i)$$

where $\{b_k \mid j_0 \le k \le j_n\}$ is a set of bugs that were fixed by the developer d_j.

Figure 6 illustrates a sample DCB network for real Eclipse developers, components and bugs. As shown in Figure 6, *nick*, *dpollock* and *johna* are three developers, 83658, 82854, 82802, 80059, and 63433 are previously fixed bugs. The four components are identified by mining the change logs to the bugs. From version control systems, we know that *nick* has fixed bug 83658 and bug 80059, *dpollock* has fixed bug 82854, and *johna* has fixed bug 82802 and bug 63433.

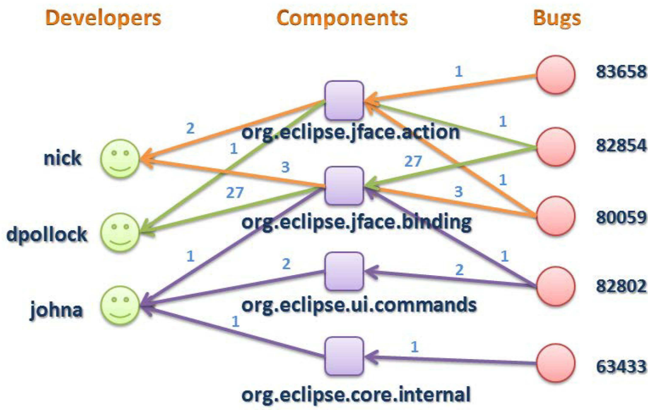


Figure 6. A sample DCB network for Eclipse developers, components, and bugs.

In Figure 6, the edges in the DCB network and their weights are set as follows:

- Every component is connected to the associated bugs. The weight between a bug and a component is set as the number of files in the component that should be changed in order to fix the bug. For example, to fix bug 82854, 27 source files in component *org.eclipse.jface.binding* and 1 file in component *org.eclipse.jface.action* are changed, therefore the weight from bug 82854 to components *org.eclipse.jface.action* and *org.eclipse.jface.binding* are 1 and 27, respectively.
- Every developer is connected to the components that she/he worked on before (i.e., the components to which the developer has committed bug-fixing changes). The weight of the edge between a component and a developer is set as the number of files in the component the developer changed. For example, *nick* has fixed two bugs, bug 83658 and bug 80059, both of which changed 1 file in component *org.eclipse.jface.action*, therefore the weight of the edge between component *org.eclipse.jface.action* and developer *nick* is 2.

2) Recommendation over DCB

When a new bug report arrives, the DCB network will be updated to determine the relevance between the new bug report and developers. Developer recommendation can be performed by computing the relevance values and ranking the developers based on the relevance values.

We calculate the relevance between the new bug report b_N and the developers level by level. Firstly, for each previously fixed bug b_i ($0 \le i \le n$) in a DCB network, we initialize its weight based on the similarity between the new bug b_N and b_i as follows:

$$b_i.weight = \frac{Similarity(b_i, b_N)}{\sum_{j=0}^{n} Similarity(b_j, b_N)}$$

We then calculate the weight of each component node c_j as follows:

$$c_j.weight = \sum_{i=j_0}^{j_n} \frac{e_{b2c}(b_i, c_j)}{\sum_{k=i_0}^{i_n} e_{b2c}(b_i, c_k)} \times b_i.weight$$

where $\{b_i \mid j_0 \le i \le j_n\}$ is a set of fixed bugs that are connected to the component node c_j and $\{c_k \mid i_0 \le k \le i_n\}$ is a set of components that are connected to the bug b_i. Finally, we calculate the relevance between a developer d_k and the new bug b_N as follows:

$$d_k.weight = \sum_{j=k_0}^{k_n} \frac{e_{c2d}(c_j, d_k)}{\sum_{i=j_0}^{j_n} e_{c2d}(c_j, d_i)} \times c_j.weight$$

where $\{c_j \mid k_0 \le j \le k_n\}$ is a set of components that are connected to the developer d_k and $\{d_i \mid j_0 \le i \le j_n\}$ is a set of developers that are connected to the component c_j.

Based on the relevance, BugFixer recommends the developers for resolving the new bug report. The developers of previously fixed bugs are ranked in descending order according to their relevance to the new bug. Developers with the higher ranks are the more relevant ones, i.e., more likely to be able to fix the new bug. The bug triager can examine the recommendation list and assign the new bug report to a suitable developer.

V. EVALUATION OF BUGFIXER

A. Data Collection

To evaluate the effectiveness of BugFixer for large projects, we choose three large-scale open source projects: Eclipse, Mozilla, and Netbeans. All of these three projects are well-known large-scale open source systems and are widely used in empirical software engineering research. They all use the Bugzilla bug tracking system and the CVS/SVN version control system. We apply the standard mining software repository approach [4] to identify the fixers of the bugs (i.e.,

the developer who committed the bug-fixing changes) for the three projects.

For Eclipse, we studied its bugs reported for Eclipse 3.1 from July 2004 to Sep 2006, which includes 2858 bug reports that have valid links to the bug-fixing changes. There have been 77 developers involved in fixing bugs. For Mozilla, we studied its bugs reported from Nov 2007 to Jun 2009, which contains 3179 bug reports that are linked to the bug-fixing commits. These bugs are committed by 119 developers. For Netbeans, we collected the bug data from Mar 2001 to Sep 2012 and identified 2048 valid bug reports, which are committed by 127 developers.

To evaluate the performance of BugFixer on different sizes of projects, we also run BugFixer on the two industrial projects as described in Section III.

In our experiments, the components in DCB are defined at the package level. For Eclipse and Netbeans, we define a component as a fourth-level Java package (such as *org.eclipse.core.debug*). For Mozilla, as it does not have explicit package structure, we identify the components by grouping files according to their full file name (with path) and the common words in the names. For example, the two files "/projects/sumo/branches/production/webroot/categorize.php" and "projects/sumo/trunk/webroot/categorize.php" are grouped into the same component named "sumo.webroot.categorize".

B. Research Questions

In order to evaluate BugFixer, we propose three Research Questions (RQs):

RQ1: How effective is the BugFixer method?

RQ1 evaluates the performance of BugFixer. To answer RQ1, we perform a 5-round incremental analysis [28] for each industrial project, as described in Section III-C. For each open source project, we perform a 10-round incremental analysis [28] because of the abundance of the data.

We also compare BugFixer with the existing machine learning based developer recommendation methods, such as those based on Naïve Bayes and SVM [1], [3], [7]. The default Naïve Bayes (with multi-nominal) and LibSVM classifiers provided by the Weka tool [24] are used as the Naïve Bayes and SVM implementation, respectively.

RQ2: Is the proposed method for computing bug similarity effective?

In Section IV-A, we propose a method for computing bug report similarity. The method includes a novel tokenization algorithm, which splits words in bug reports using the algorithm described in Figure 5. The method also utilizes the source file names to supplement the bug report information of existing bugs. Without using the proposed bug similarity method, BugFixer could use the conventional preprocessing method, which separates words using non-alphabetical letters, removes common stop words, and performs stemming. Furthermore, the conventional method does not consider source file information. To answer this RQ, we compare the performance of BugFixer using the proposed and the conventional bug similarity methods, while keeping the rest elements (VSM similarity measure, DCB network) the same.

RQ3: Is the proposed recommendation over DCB technique effective?

In Section IV-B, we propose to use a DCB network to perform recommendations by computing relevance between developers and new bugs. This RQ evaluates the effectiveness of the DCB technique by comparing the performance of BugFixer with and without using DCB. Without using DCB, BugFixer could compute the similarity between the new bug and the previously fixed bugs as described in Section IV-A, and recommend the developers of similar bugs to resolve the new bug. The developers are ranked in descending order according to their similarities to the new bug. When using DCB, BugFixer performs recommendations over DCB as described in Section IV-B.

We use the same Recall@k metric as described in Section III to evaluate the performance of BugFixer, which shows the number of bugs whose associated developer is ranked in the top k (k=1, 3, 5) of the returned results.

C. Evaluation Results

RQ1: How effective is the BugFixer method?

We evaluate BugFixer by performing the 10-round incremental analysis on Eclipse, Mozilla and Netbeans datasets. For projects EBCP and DMGR, we perform the 5-round incremental analysis. The results are shown in Table II. For Eclipse, BugFixer achieves average Recall@1 value 42.36%, which means that for 42.36% bugs, BugFixer successfully recommends their associated developers as top 1. The Recall@5 value is 73.85%, which means that for 73.85% bugs, their developers can be found in the top 5 return results.

For Mozilla, BugFixer achieves Recall@1 and Recall@5 values 27.38% and 60.02%, respectively. For Netbeans,

TABLE II. EVALUATION RESULTS OF BUGFIXER ON FIVE PROJECTS

Project	Rank	SVM	Naïve Bayes	BugFixer
Eclipse	Top1	26.30%	26.18%	42.36%
	Top3	45.03%	27.21%	67.31%
	Top5	54.23%	29.43%	73.85%
Mozilla	Top1	25.16%	25.81%	27.38%
	Top3	45.88%	28.89%	48.78%
	Top5	55.57%	30.45%	60.02%
Netbeans	Top1	10.86%	14.03%	20.93%
	Top3	19.18%	14.67%	36.70%
	Top5	26.60%	15.80%	44.54%
EBCP	Top1	23.19%	18.26%	23.50%
	Top3	44.86%	20.21%	47.08%
	Top5	61.19%	25.31%	51.95%
DMGR	Top1	27.92%	22.54%	28.60%
	Top3	51.90%	42.03%	51.69%
	Top5	66.15%	49.15%	64.54%

TABLE III. RESULTS OF INCREMENTAL ANALYSIS ON FIVE PROJECTS

Project	Rank	Round1	Round2	Round3	Round4	Round5	Round6	Round7	Round8	Round9	Round10	Avg
Eclipse	Top1	38.08%	42.69%	41.92%	46.15%	45.00%	45.38%	49.62%	41.54%	46.54%	26.64%	**42.36%**
	Top3	67.31%	62.31%	65.38%	71.92%	69.23%	71.15%	60.38%	69.23%	46.72%	64.36%	**67.31%**
	Top5	73.85%	72.31%	73.46%	80.77%	80.77%	80.00%	69.23%	76.15%	58.69%	73.14%	**73.85%**
Mozilla	Top1	35.80%	35.02%	27.24%	24.51%	26.07%	22.96%	25.29%	25.68%	17.12%	34.13%	**27.38%**
	Top3	58.75%	56.03%	44.75%	43.19%	44.75%	41.63%	50.58%	56.81%	38.13%	53.17%	**48.78%**
	Top5	66.93%	68.48%	54.47%	54.09%	53.31%	53.70%	62.65%	66.54%	52.14%	67.86%	**60.02%**
Netbeans	Top1	13.37%	10.70%	11.23%	25.13%	33.16%	37.97%	24.06%	27.81%	9.09%	16.76%	**20.93%**
	Top3	25.67%	21.93%	21.93%	39.04%	51.34%	57.75%	45.99%	54.01%	20.86%	28.49%	**36.70%**
	Top5	36.90%	26.74%	25.67%	47.06%	58.82%	63.10%	54.01%	60.96%	36.90%	35.20%	**44.54%**
EBCP	Top1	27.71%	21.69%	20.48%	30.12%	17.50%	-	-	-	-	-	**23.50%**
	Top3	44.58%	36.14%	43.37%	65.06%	46.25%	-	-	-	-	-	**47.08%**
	Top5	53.01%	37.35%	49.40%	67.47%	52.50%	-	-	-	-	-	**51.95%**
DMGR	Top1	20.17%	41.18%	32.77%	20.17%	28.70%	-	-	-	-	-	**28.60%**
	Top3	42.02%	80.67%	45.38%	51.26%	39.13%	-	-	-	-	-	**51.69%**
	Top5	68.91%	93.28%	51.26%	58.82%	50.43%	-	-	-	-	-	**64.54%**

BugFixer achieves Recall@1 and Recall@5 values 20.93% and 44.54%, respectively. The results show that BugFixer is effective in recommending developers to fix bugs for large projects.

Table II also shows that, for large projects (Eclipse, Mozilla, and Netbeans), BugFixer obtains better results than the conventional machine learning based recommendation methods. Comparing with Naïve Bayes, the results of BugFixer are 6 ~ 182% better. For example, BugFixer obtains 182% ((44.54-15.80)/15.80) better result on Recall@5 for Netbeans. Comparing with SVM, the results of BugFixer are 6 ~ 93% better. For example, Bugfixer achieves 93% ((20.93-10.86)/10.86) better result on Recall@1 for Netbeans. For smaller projects, i.e., EBCP and DMGR, BugFixer generally achieves comparable accuracy as SVM (except the Recall@5 result for EBCP), and better accuracy than Naïve Bayes.

We also present the results of incremental analysis for each project in Table III. For each project, Recall@k values for each round are listed. For Eclipse, the Recall@5 values range from 58.69% to 80.77%. For Mozilla, the Recall@5 values range from 52.14% to 68.48%. For Netbeans, the Recall@5 values range from 25.67% to 63.10%. The difference between each round is caused by the data characteristics of bug reports and their developers. Note that the second round of Recall@5 result for EBCP is low, therefore causing the overall low average Recall@5 value for EBCP.

In summary, our experiments show that BugFixer is effective in bug triage, especially for large-scale projects. For small projects, BugFixer generally achieves comparable performance as the SVM-based bug triage method.

RQ2: Is the proposed method for computing bug similarity effective?

For the large projects, we compare the performance of BugFixer with and without using the proposed method for computing bug similarity (Section IV-A). The comparison results are shown in Table IV. By adopting the proposed method, the performance of BugFixer increases for all projects. The relative improvement ranges from 6.46% to 23.18%. The results confirm the effectiveness of the proposed method for computing bug similarity.

RQ3: Is the proposed recommendation over DCB Technique effective?

We compare the performance of BugFixer on large projects with and without the proposed DCB network. The comparison results are shown in Table V. By adopting the recommendation over DCB technique as described in Section IV-B, the performance of BugFixer increases for all three projects. The relative improvement ranges from 7.85% to 48.58%. The results confirm the effectiveness of the proposed DCB-based recommendation technique.

TABLE IV. EVALUATION OF THE PROPOSED METHOD FOR COMPUTING BUG SIMILARITY

Project	Rank	Conventional	Proposed	Improve
Eclipse	Top1	34.39%	42.36%	23.18%
	Top3	55.48%	67.31%	21.32%
	Top5	65.33%	73.85%	13.04%
Mozilla	Top1	24.88%	27.38%	10.05%
	Top3	45.65%	48.78%	6.86%
	Top5	56.38%	60.02%	6.46%
Netbeans	Top1	18.15%	20.93%	15.32%
	Top3	31.77%	36.70%	15.52%
	Top5	38.64%	44.54%	15.27%

TABLE V. EVALUATION OF THE PROPOSED DCB TECHNIQUE

Project	Rank	Without DCB	With DCB	Improve
Eclipse	Top1	28.51%	42.36%	48.58%
	Top3	50.55%	67.31%	33.16%
	Top5	60.94%	73.85%	21.18%
Mozilla	Top1	24.49%	27.38%	11.80%
	Top3	45.23%	48.78%	7.85%
	Top5	55.25%	60.02%	8.63%
Netbeans	Top1	16.32%	20.93%	28.25%
	Top3	29.84%	36.70%	22.99%
	Top5	37.02%	44.54%	20.31%

VI. DISCUSSIONS

A. Why does BugFixer Work Better than the Conventional Methods?

Conventional machine learning based methods are essentially a text-categorization approach, utilizing a classifier such as Naïve Bayes or SVM. They treat each developer as a category and each bug report as a document. For a small number of developers, such categorization could be effective. However, when the number of developers (categories) increases, it is hard for a classifier to perform multi-class classification. Furthermore, designing an accurate and efficient classifier is very challenging for multi-class classification.

BugFixer does not adopt a text-categorization approach. It utilizes historical bug-fix information. The Developer-Component-Bug network captures the knowledge of "who fixed what, where". In a DCB network, the bugs are linked to the components in which they are fixed; and the developers are linked to the components in which they fix the bugs. The components could reflect the expertise of the developers and the characteristic of the bugs. Once a new bug report arrives, BugFixer performs recommendation over DCB based on the relevance scores between the developers and the new bug report. In this sense, BugFixer is more like a random walk approach [21].

B. The Missing Cases

Although our experiments show that BugFixer can effectively recommend suitable developers for fixing bugs, it may still fail to provide good recommendation for certain bug reports.

Some bugs could be fixed by a new developer (the developer who did not fix any bugs before), therefore his/her expertise cannot be learned from historical data. For example, in our experiments, on average, 4.36% of the Mozilla bugs and 2.89% of the DMGR bugs are fixed by a new developer in each round, therefore the performance of BugFixer is affected. Furthermore, some developers may have the expertise for fixing a bug, but they were busy with other duties at the time of bug triage therefore the task was eventually assigned to another developer.

Once a new bug report arrives, BugFixer applies the VSM model to compute the similarity between the new bug report and the previously fixed bug reports, before performing recommendation over DCB. However, some bug reports could be very brief, as the users/testers did not spend much time on writing detailed problem scenarios. This causes difficulty in similarity comparison. Furthermore, different users/testers may use different words to describe the same problem. For example, when reporting the problem of a software crash, different reporters may use different words such as "exception", "crash", "failure", "error", "down", etc. Currently BugFixer does not consider the semantic similarity among words. Therefore, it may treat these reports as dissimilar texts, causing the inaccuracy in recommendation.

We will address the above issues in our future work and to further improve the performance of the proposed approach.

C. Threats to Validity

In this section, we list the potential threats to validity of our work as follows:

In this paper, we evaluate automated bug triage methods through a case study of two industrial projects. Then we propose a new method to further improve the effectiveness of bug triage on five projects. There is still room for further improvement regarding generality. In our work, we find out that the project size can affect the performance of automated bug triage. This finding may be limited by the types of projects. Thus, it is helpful to evaluate more bug data from different types of projects. Moreover, in our industrial case study, to form projects with different sizes, we randomly sample developers in the original projects. We individually run such sampling 10 times to avoid the sampling bias.

In bug tracking systems, all the bug reports are described in natural languages. Hence, the performance of our approach relies on the quality of bug reports. A poorly-written bug report cannot provide sufficient information for bug triage, or even mislead classifiers. To avoid the disturbing of low-quality bug reports, identifying the quality of bug reports [10] would be useful.

In our proposed method BugFixer, we design a Developer-Component-Bug network to model the relationship between bug reports and their developers. We leverage component information to connect bug reports and developers. The links between components and bugs and the links between components and developer are obtained by mining change logs and bug reports. However, sometimes such links could be noisy. For example, some links between a bug and the component where the bug is fixed could be missing because the developers did not explicitly mark the bug ID in change logs or source code [25]. In the future, we will study the impact of data quality on bug triage.

VII. RELATED WORK

In recent years, many studies have been carried out on bug reports, for example, bug-fixing time prediction [23], [30], bug-proneness prediction [31], [34], bug localization [33], and quality analysis of bug reports [10]. A prior step to bug triage is duplicate bug detection. The aim of duplicate bug detection

is to identify whether a new bug report has the same root cause of an existing one. Runeson et al. [17] first proposed this problem and used an information retrieval technique to model the similarity between duplicate bug reports. Wang et al. [22] combined the execution trace with the bug description to further improve duplicate bug detection. Recent work by Zhou and Zhang [32] employed a learning-to-rank technique to train the rankings of potential duplicate bugs.

For bug triage, Anvik et al. [1] experimented with six machine learning algorithms for automated bug assignment. Such algorithms include Naïve Bayes, SVM, C4.5, expectation maximization, nearest neighbor, and conjunctive rules. Their experimental results show that SVM and Naïve Bayes algorithms produced the highest precision when making one recommendation. However, when making two or more recommendations, SVM generally provides a higher precision. They therefore chose SVM as the algorithm for designing a developer recommender. In recent years, methods are investigated to improve the text-categorization based bug triage, such as those using the semi-supervised text classification [29], the fuzzy set [20] and the data reduction [27]. In our work, we utilize historical bug-fix information to improve the bug triage accuracy.

Bug triage is related to the work on determining developers' expertise. Developers could accumulate expertise over years in certain areas of software development. Therefore, identifying the developer expertise could be helpful to the work allocation. Expertise Browser [15] considers the person who commits the code to source code repository an expert of the source file. Anvik and Murphy [2] proposed to mine the implementation expertise from bug reports and evaluate various methods to model the expertise. Fritz et al. [8] proposed a degree-of-knowledge approach to capture the implementation expertise. Their model utilizes both code authorship and developer interaction information. Moreover, Matter et al. [14] proposed a vocabulary-based method to measure the developer expertise and suggest developers with appropriate expertise to handle bugs. Using eight years of Eclipse development as a case study (13,077 bugs in total), their method can achieve 33.6% top-1 precision and 71.0% top-10 recall. Wu et al. [26] utilized the expertise to identify the developers who can make contributions to projects. They examined various models of the developer expertise and found that the frequency and the out-degree in developer collaboration outperform the other models. Developer expertise can be also helpful to recommending developers to assistant performing software changes. For example, Kagdi and Poshyvanyk [12] proposed an approach that combines the text similarity in Latent Semantic Indexing with the code expertise obtained by mining software repositories. This approach can be used to recommend a ranked list of candidate developers for making source code changes.

In bug triage, after the initial assignment, a bug report could be reassigned to other developers. Many studies have investigated the bug report reassignment. Guo et al. [9] presented a large-scale analysis of the process of bug reassignment in Microsoft Windows Vista. In contrast to the popular opinion that reassignments are always harmful, their work found that certain reassignments are useful to determine the best person for bug fixing. Jeong et al. [11] explored the historical reassignments in bug triage and proposed a bug tossing graph to reduce the reassignment. Bhattacharya and Neamtiu [5] extended the tossing graph with multiple features to further improve the effectiveness.

In recent years, developer social networks have been employed for recommendation and prediction. Xuan et al. [28] prioritized developers with a developer communication network based on bug comments. Pinzger et al. [16] built a developer-module network and predicted software module failures with social network metrics. Surian et al. [19] proposed a Developer-Project-Property (DPP) structure as a representation of the developers' collaboration network for developer recommendation. Zhou et al. [32] utilized a bug-file structure to recommend source code files for fixing a bug. In contrast to above approaches, the proposed method in this paper, BugFixer, utilizes the relationship between developers, components, and bug reports. The Developer-Component-Bug network in our work can effectively model the developer activities in bug triage.

VIII. CONCLUSIONS

Bug triage could be a time-consuming and costly process. Our empirical studies on two industrial projects show that the conventional bug triage methods are effective for these projects. We also find that the conventional bug triage methods become less effective when the number of developers increases.

To improve the performance of bug triage for large projects, we have proposed BugFixer, an automated bug report assignment method that utilizes historical bug fix data. BugFixer adopts a new method for computing bug report similarity. BugFixer also constructs a novel network structure, called Developer-Component-Bug (DCB), to model the relationship between the developers and the source code components they worked on, as well as the relationship between the components and the associated bugs. For a new bug, BugFixer calculates its similarity to existing bugs, and recommends developers based on the structure of DCB network. We have evaluated BugFixer on three large-scale open source projects, namely Eclipse, Mozilla, and Netbeans. The evaluation results are promising. For example, for the Eclipse project, the actual developers of 42.36% bugs are ranked as top 1 in our recommendation list while the actual developers of 73.85% bugs are ranked as top 5. These results are significantly better than those of the conventional machine learning based methods (SVM and Naïve Bayes based ones). Our experimental results also show that, for small projects, BugFixer generally achieves comparable performance as the conventional, SVM-based method.

In the future, we plan to carry out large-scale evaluations of the proposed method on a variety of projects. We will also identify more features (such as those described in [9]) that could affect the bug triage process.

ACKNOWLEDGMENT

We thank the students Shuai Chen, Ke Ma, and Pinjia He for their initial efforts in helping with the project. This work is supported by the National Natural Science Foundation of China under a grant 61272089 and the INRIA Postdoctoral Research Fellowship.

REFERENCES

[1] J. Anvik, and G. Murphy, Reducing the effort of bug report triage: Recommenders for development oriented decisions. ACM Trans. Softw. Eng. Methodol. 20(3), Aug 2011.

[2] J. Anvik, G.C. Murphy, Determining implementation expertise from bug reports. In Proc. MSR '07, May 2007.

[3] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In Proc. ICSE'06, Shanghai, China, May 2006, pp. 361–370.

[4] A. Bachmann and A. Bernstein. Software process data quality and characteristics: a historical view on open and closed source projects. In Proc. IWPSE-Evol '09, Aug 2009, pp. 119-128.

[5] P. Bhattacharya and I. Neamtiu, Fine-Grained Incremental Learning and Multi-Feature Tossing Graphs to Improve Bug Triaging. In Proc. ICSM'10, Sep 2010, pp. 1-10.

[6] M. Cohn, Succeeding with Agile: Software Development Using Scrum", 2009, available at www.mountaingoatsoftware.com.

[7] D. Čubranić and G. C. Murphy, Automatic Bug Triage Using Text Categorization, In Proc. SEKE '04, Jun 2004, pp. 92-97.

[8] T. Fritz, J. Ou, G. Murphy, and E. Murphy-Hill, A Degree-of-Knowledge Model to Capture Source Code Familiarity. In Proc. ICSE 2010, May 2010, pp. 385-394.

[9] P. J. Guo, T. Zimmermann, N. Nagappan, B. Murphy: "Not my bug!" and other reasons for software bug report reassignments. In Proc. CSCW 2011, March 2011, pp. 395-404.

[10] P. Hooimeijer and W. Weimer. Modeling bug report quality. In Proc. ASE'07, Nov 2007, pp. 34‑43.

[11] G. Jeong, S. Kim, and T. Zimmermann, Improving Bug Triage with Tossing Graphs, in Proc. FSE'09, Aug 2009, pp. 111-120.

[12] H. Kagdi and D. Poshyvanyk, "Who can help me with this change request?" in Proc. ICPC 2009, pp. 273–277.

[13] D. Ma, D. Schuler, T. Zimmermann, and J. Sillito, Expert recommendation with usage expertise, in Proc. ICSM'09, Sep 2009, pp. 535-538.

[14] D. Matter, A. Kuhn, and O. Nierstrasz, Assigning bug reports using a vocabulary-based expertise model of developers, Proc. MSR'09, May 2009, pp. 131-140.

[15] A. Mockus and J. D. Herbsleb. Expertise browser: a quantitative approach to identifying expertise. In Proc. ICSE 2002, pp. 503–512.

[16] M. Pinzger, N. Nagappan, and B. Murphy, Can Developer-Module Networks Predict Failures?, In Proc. FSE'08, Atlanta, GA, Nov 2008, pp. 2-12.

[17] P. Runeson, M. Alexanderson, O. Nyholm, Detection of Duplicate Defect Reports Using Natural Language Processing. In Proc. ICSE'07, May 2007, pp. 499-510.

[18] K. Schwaber, M. Beedle, Agile software development with Scrum, Prentice Hall, 2002.

[19] D. Surian, N. Liu, D. Lo, H. Tong, E. Lim, and C. Faloutsos, Recommending People in Developers' Collaboration Network. In Proc. WCRE'11, Oct 2011, pp. 379-388.

[20] A. Tamrawi, T.T. Nguyen, J.M. Al-Kofahi, and T.N. Nguyen, Fuzzy-set and cache-based approach for bug triaging, In Proc. FSE 2011, pp. 365-375.

[21] H. Tong, C. Faloutsos, and J. Pan, Fast Random Walk with Restart and Its Applications. In Proc. ICDM 2006, Hong Kong, Dec 2006, pp. 613-622.

[22] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, An approach to detecting duplicate bug reports using natural language and execution information. In Proc. ICSE'08, Leipzig, Germany, May 2008, pp. 461-470.

[23] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller. How long will it take to fix this bug? In Proc. MSR'07, May 2007.

[24] I. H. Witten, E. Frank, M. A. Hall, Data Mining: Practical Machine Learning Tools and Techniques (Third Edition), Morgan Kaufmann, 2011.

[25] R. Wu, H. Zhang, S. Kim, S.C.Cheung, ReLink: Recovering Links between Bugs and Changes. In Proc. ESEC/FSE 2011, Szeged, Hungary, Sep 2011, pp.15-25.

[26] W. Wu, W. Zhang, Y. Yang, and Q. Wang, DREX: Developer Recommendation with K-Nearest-Neighbor Search and Expertise Ranking. In Proc. APSEC 2011, Dec 2011, pp.389-396.

[27] J. Xuan, H. Jiang, Y. Hu, Z. Ren, W. Zou, Z. Luo, and X. Wu, Towards Effective Bug Triage with Software Data Reduction Techniques. IEEE Trans. Knowledge and Data Engineering, preprint, 2014.

[28] J. Xuan, H. Jiang, Z. Ren, and W. Zou, Developer Prioritization in Bug Repositories. In Proc. ICSE'12, June 2012, pp. 25-35.

[29] J. Xuan, H. Jiang, Z. Ren, J. Yan, and Z. Luo, Automatic Bug Triage using Semi-Supervised Text Classification. In Proc. SEKE'10, Redwood City, California, Jul. 2010, pp. 209-214.

[30] H. Zhang, L. Gong, and S. Versteeg. Predicting bug-fixing time: an empirical study of commercial software projects. In Proc. ICSE'13, San Francisco, USA, May 2013, pp. 1042–1051.

[31] H. Zhang, An Investigation of the Relationships between Lines of Code and Defects. In Proc. ICSM 2009, Edmonton, Canada, Sep 2009, pp.274-283.

[32] J. Zhou and H. Zhang, Learning to rank duplicate bug reports. In Proc. CIKM 2012, Maui, Hawaii, Oct 2012, pp.852–86.

[33] J. Zhou, H. Zhang, D. Lo, Where Should the Bugs Be Fixed? More Accurate Information Retrieval-Based Bug Localization Based on Bug Reports. In Proc. ICSE'12, June 2012, pp. 14-24.

[34] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for eclipse. In Proc. PROMISE'07, Minneapolis, Minnesota, USA, May 2007, pp. 1-9.