IEICE TRANSACTIONS

on Information and Systems

VOL. E103-D NO. 2
FEBRUARY 2020

A PUBLICATION OF THE INFORMATION AND SYSTEMS SOCIETY
The Institute of Electronics, Information and Communication Engineers
Kikai-Shinko-Kaikan Bldg., 5-8, Shibakoen 3 chome, Minato-ku, TOKYO, 105-0011 JAPAN

IEICE TRANS. INF. & SYST., VOL.E103–D, NO.2 FEBRUARY 2020

PAPER

A Release-Aware Bug Triaging Method Considering Developers' Bug-Fixing Loads

Yutaro KASHIWA[†a)], ***Nonmember and*** **Masao OHIRA**[††b)], ***Member***

SUMMARY This paper proposes a release-aware bug triaging method that aims to increase the number of bugs that developers can fix by the next release date during open-source software development. A variety of methods have been proposed for recommending appropriate developers for particular bug-fixing tasks, but since these approaches only consider the developers' ability to fix the bug, they tend to assign many of the bugs to a small number of the project's developers. Since projects generally have a release schedule, even excellent developers cannot fix all the bugs that are assigned to them by the existing methods. The proposed method places an upper limit on the number of tasks which are assigned to each developer during a given period, in addition to considering the ability of developers. Our method regards the bug assignment problem as a multiple knapsack problem, finding the best combination of bugs and developers. The best combination is one that maximizes the efficiency of the project, while meeting the constraint where it can only assign as many bugs as the developers can fix during a given period. We conduct the case study, applying our method to bug reports from Mozilla Firefox, Eclipse Platform and GNU compiler collection (GCC). We find that our method has the following properties: (1) it can prevent the bug-fixing load from being concentrated on a small number of developers; (2) compared with the existing methods, the proposed method can assign a more appropriate amount of bugs that each developer can fix by the next release date; (3) it can reduce the time taken to fix bugs by 35%–41%, compared with manual bug triaging;

key words: *bug triage, optimization, project management, repository mining, machine learning*

1. Introduction

Modern software systems are used for a variety of purposes in a wide range of scenarios, and they are closely interconnected. Increasing numbers of new features improve convenience for users but also mean that development has become a more large-scale and more complicated process. In developing large-scale and complicated systems, a significant number of bugs are detected during testing because the developers cannot comprehend such large code bases in their entirety [1], [2]. Many system development processes use bug tracking systems such as Bugzilla [3] or Jira [4] to record how to reproduce and fix bugs in detail. The process of determining the importance and priority of each reported bug, and then assigning a developer to fix it is called bug triage [5].

When there are a large number of reported bugs and project developers in the test phase, understanding aspects such as what skills each developer has and how many bugs each developer is currently addressing is difficult. This would lead that not all bugs are properly triaged. In fact, The limitations of manual bug triage are well-known. In Eclipse Platform and Mozilla Firefox, about 40% of bugs are reassigned for fixing [6]. Reassignments should be prevented as much as possible because reassignments do not only waste human resources but also delay the fixing of the bug. Therefore, supporting bug triage have been actively studied [6]–[17] for a decade.

Most of the proposed methods have aimed to reduce reassignments by recommending developers who can reliably and quickly fix individual newly-reported bugs, based on previously-reported bugs and their bug-fixing history. However, they possibly concentrate their assignments on a small number of particular developers because the number of past bug fixes differ depending on the developer which makes the training data for each developer imbalanced. Since software development is generally tied to release dates, the number of bugs that can be fixed by even experienced developers before each release is limited. Therefore, the concentration on the specific developers may reduce the number of bugs that the developers can fix by the next release date.

In this research, we propose a Release-Aware Bug-Triaging method (RABT) for the test phase, which considers the bug-fixing loads placed on developers, to increase the number of bugfixes by the next release date. We regard the bug assignment problem as a combination problem between bugs and developers and we formulate it as a multiple knapsack problem to find the optimal combinations. We optimize the assignment process by finding bug assignments that satisfy certain constraints, aiming to (1) mitigate the task concentration problem caused by existing methods and (2) assign the appropriate amount of bugs to fix more bugs by the next release date.

Paper Organization: The rest of this paper is organized as follows. Section 2 describes problems with existing bug triage methods and our key idea for addressing them. Our method is introduced in Sect. 3, and its implementation is described in Sect. 4. Sections 5 and 6 present our experiments and results, respectively. We discuss the results in Sect. 7. Finally, Sect. 8 concludes this paper.

Manuscript received June 5, 2019.
Manuscript revised September 18, 2019.
Manuscript publicized October 25, 2019.

[†]The author is with Graduate School of Systems Engineering, Wakayama University, Wakayama-shi, 640–8510 Japan.
[††]The author is with the Faculty of Systems Engineering, Wakayama University, Wakayama-shi, 640–8510 Japan.
a) E-mail: kashiwa.yutaro@g.wakayama-u.jp
b) E-mail: masao@wakayama-u.ac.jp

DOI: 10.1587/transinf.2019EDP7152

2. Bug Triage

2.1 Role of Bug Triage in the Bug-Fixing Process

When a user (or developer) finds a bug, they report it to the project's bug tracking system (BTS), which manages bug information. Corporating with the co-developers, the project's main developers who are called dispatchers investigate the bug and identify its cause. The dispatchers then determine whether the bug should be fixed, and if so, they prioritize it. Then, the dispatchers find developers who are available and are suitable for the job. All of these activities that are performed by the dispatchers are necessary for bug triage. In particular, the choice of the developer assigned to each bug affects the bug-fixing time. Ways to support such decision-making are therefore studied in the field of mining software repositories. In this paper, we use "bug triage" in the narrow sense of deciding the developer to which a given bug should be assigned.

2.2 Manual Bug Triage and Its Problems

Dispatchers need to select an appropriate developer who should be assigned to a bug. In many cases, however, the selected developer cannot fix the bug, and the dispatchers must assign it to another developer instead (herein, known as reassignment). The reassignment process continues until the bug is finally fixed and leads to delay in the process of bug fixing [6], [15]. In fact, in the Eclipse Platform and Mozilla Firefox projects, reassignments are required for about 40% of bugs, increasing the average bug-fixing time by about 50 days per reassignment [6], [15]. Proper assignments are not easy because of the following reasons.

2.2.1 Identifying the Skills Needed to Fix a Bug is Difficult

Bugs reported to software developers range from ones that must be urgently fixed, such as security vulnerabilities, to less urgent ones such as usability improvement. Even for the same type of bug, fixing it can require advanced technical skills and expertise.

The dispatchers must check the bug's type and difficulty level and identify an appropriate developer for each of a large number of reported bugs every day. If the assigned developer does not have the skills needed to fix the bug, reassignment is required. To prevent this, time should be taken to check the developers' skills and experience. However, since dispatchers are often involved in, for example, bug-fixing and developing new features as well as assigning tasks, finding time to fully investigate bugs and comprehend the developers' skills and experience can be difficult. Previous studies have proposed methods to predict developers' skills needed to fix given bugs [18]–[21] and to recommend developers qualified to fix given bugs [5], [13], [16], [22], [23].

2.2.2 Understanding the Load on Each Developer is Difficult

Large projects involve many developers, who live all around the world. Since understanding each developer's bug-fixing workload can be difficult for dispatchers, particularly in distributed development, the dispatchers inevitably assign large numbers of tasks to particular developers, putting heavy loads on them. In fact, when we asked the most significant contributors in Eclipse Platform and Mozilla Firefox (the top 10 developers in each project who had previously fixed bugs) whether they felt overloaded with bug-fixing activities, four of the obtained five responses denoted that they "always" or "sometimes" felt overloaded.

Such assigning large numbers of tasks to certain developers may causes reassignments which delays the bugs from being fixed [24]. Attempting to triage bugs without considering each developer's bug-fixing workload does not only impose a burden on the developers but also mean that bugs are left unaddressed for long periods of time. Despite this, no bug triage methods have yet been proposed to consider the developer's workload for the bug-fixing tasks assigned to each developer.

2.3 Overview of Existing Methods

As discussed above, reassignment is a significant waste of resources and delays the process of bug fixing. The lengthy bug-fixing times caused by reassignment must be urgently addressed in large-scale open-source software (OSS) projects, and many methods for supporting bug triage have been proposed [5]–[17]. These methods are mainly classified into content-based recommendation [5]–[15] or cost-aware recommendation [15]–[17], we give an overview of representative approaches in each category in the following sections.

2.3.1 Content-Based Recommendation

The purpose of content-based recommendation (CBR) method [5] is to assign tasks to appropriate developers to avoid frequent reassignments. First, they parse textual data comprising titles and abstracts of fixed bug reports, noting the frequency of particular words appear and extracting the fixer of each bug. Next, they input this information into a machine learning algorithm (Naive Bayes [25], Support Vector Machine [26] or C4.5 [27]) and they obtain a model for recommending developers for each bug. Using this model, CBR can recommend developers who are capable of dealing with newly-reported bugs with relatively high accuracy (about 70%–75%)[†].

[†]Here, the correct answers are considered to be the assignments made by a developer with experience in bug assignment

2.3.2 Cost-Aware Recommendation (CosTriage)

The purpose of cost-aware recommendation is to reduce time for fixing bugs. Park et al. have proposed CosTriage which aims to assign tasks to developers who can quickly fix the bugs while keeping the accuracy as much as that of CBR [16]. First, as with CBR, it determines the probability that each developer is appropriate for the bug. Then, it calculates the time required to fix the bug. Next, it requires the balance of how much accuracy of assignments is important compared to the quickness of bug-fixing (accuracy = 1 - quickness). Finally, based on the assessment, it recommends the most appropriate developer for the bug. Although this reduced recommendation accuracy by about 5% compared with CBR, the average bug-fixing time is reduced by 7%-31%.

2.4 Problems with Existing Methods

The existing methods have advantages of being able to recommend appropriate developers or shortening the bug-fixing time. However, as we have already discussed, they do not consider how many bugs a developer can address in a given period of time, and they may assign even minor bugs to experienced developers. As a result, they may assign more tasks to some experienced developers than the developers can reasonably address in the available time. In the case that the methods are used in the test phase, they decrease the number of bugs that developers can fix by the next release.

To solve the problems in the existing methods, we previously proposed the method which considers developers' load in order to reduce bug-fixing time throughout the software development [28] (Note that this paper is written in Japanese). We confirmed the previous method reduces bug-fixing time, but we still have not evaluated how much bug-fixes (in the test phase) the methods would increase by the next release. This is because the previous method was designed for the bug-fixing throughout software development.

Moreover, the method ignores the contents of the symptoms described in the bug reports. To select an appropriate developer for the reported bug, the method measures the bug fixing experience based on the count of bug-fixes relating the component described in the reported bug. Generally speaking, dispatchers assign the bugs based on the contents of the bug reports. Hence, the count-based method is far from the bug-triaging activity (We have not compared the method with the existing methods in terms of accuracy of assignments in [28]).

In this study, we extend the previous method and replace the count-based recommendation into content-based recommendation used in the existing methods (CBR and CosTriage). For the evaluation, we evaluate the method with the number of fixed bugs by the next release date and the accuracy of assignments, in addition to the fixing-time.

3. Bug Triage as a Multiple Knapsack Problem

3.1 Multiple Knapsack Problem

The multiple knapsack problem [29], [30] is an optimization problem that involves finding the best combinations of items (with certain weights and values) to put in a series of knapsacks. Here, each knapsack has a maximum weight that it can carry. Figure 1 gives an overview of the multiple knapsack problem, which extends the well-known knapsack problem to multiple knapsacks. In addition to deciding whether or not to put an item in the knapsack, it requires us to decide what items to put into each knapsack, significantly increasing the computation required. The multiple knapsack problem can be formulated as follows.

$$Maximize: \sum_{i=1}^{m}\sum_{j=1}^{n} v_j x_{ij} \tag{1}$$

$$Subject\ to: \sum_{j=1}^{n} w_j x_{ij} \le c_i \quad (i = 1, 2, \ldots, m) \tag{2}$$

$$\sum_{i=1}^{m} x_{ij} \le 1 \qquad (j = 1, 2, \ldots, n) \tag{3}$$

$$x_{ij} \in \{0, 1\} \qquad (j = 1, 2, \ldots, n) \tag{4}$$

Here, v_j and w_j represent the value and weight of the j-th item, respectively, whereas x_{ij} is the objective variable, representing whether (1) or not (0) to put the j-th item into the i-th knapsack. Expression (1) is the objective function and is used to determine whether one combination of objective variable values is better than the other and in this case aims to maximize the total value of the selected items. In contrast, Expression (2) is a constraint that denotes that the total weight placed in the i-th knapsack must be less than the maximum weight it can carry(c_i), and Expression (3) prevents any item being placed in more than one knapsack. Expression (4) denotes the constraint that the x_{ij} should only take values of 0 (not selected) or 1 (selected), i.e., should represent whether the i-th knapsack contains item j.

The purpose of the multiple knapsack problem is to find combinations of x_{ij} values that maximize the value of Expression (1) under the constraints Expressions (2), (3), and (4), which can be reduced easily with a solver such as lp_solve [31]

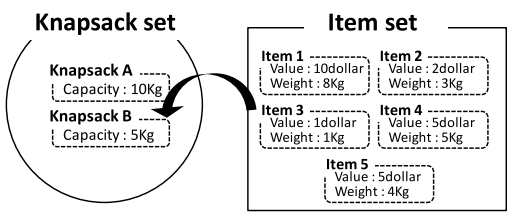

Fig. 1 Overview of the multiple knapsack problem

Table 1 List of terms used in this paper

Term	Variable	Meaning
Category	k	Bug category (classified by LDA).
Preference	P_{ij}	Used to prioritize developers when assigning bug-fixing tasks. P_{ij} is the probability that developer D_i is the most appropriate for fixing bug B_j.
Cost	C_{ij}	Time taken by developer D_i to fix bug B_j, equal to the median time required by developer D_i to fix a bug in category k in the past.
Limitation	L	Prevents task concentration.
Available assignment time	T_i	Time available for bug fixing within a given period. T_i is the amount of time developer D_i has available: T_i =Limitation$L-\sum_{j=1}^{n} C_{ij} * x_{ij}$

3.2 Application of the Multiple Knapsack Problem to Bug Triage

In this paper, we formulate bug triage as a multiple knapsack problem and use its solution to optimize task assignments. We obtain a combination of items (bugs) and knapsacks (developers) that maximizes the objective (bug-fixing efficiency for the whole project) under each knapsack's weight constraint (maximum time available to each developer or **limit**). The weights are the costs of fixing the bugs (**cost**), and the values are the developers' suitabilities for each bug (**preference**). The terms used in this paper are summarized in Table 1.

Notably, the developers' suitability (preferences) and costs will differ depending on which developers are assigned to which bugs. As a result, the variables in this problem are different from those in the general multiple knapsack problem (we have switched from v_j to P_{ij} and from w_j to C_{ij}).

3.2.1 Preferences (Developer Suitabilities)

Here, the coefficients for the objective variables in the multiple knapsack problem's objective function are the **preference** P, indicating which developers should be preferred for and can fix particular bug-fixing tasks.

The preference of developer D_i for fixing bug B_j is defined as the probability P_{ij} that developer D_i is the most appropriate for the task among all developers (i.e. the total of probability for each developers will be 1). The reasons we adopt the probabilities are that, the ones are commonly used in bug assigning methods [5], [14], [16], [22], they can take into account the contents in the descriptions of bug reports, and statistically measure the appropriateness of the task for developers. Although several studies weight the scores according to priority or severity included in bug reports [9], [32], the values of priority and severity are unreliable. Saha et al. reported that the levels of priority and severity are not actual [33]. Thus, RABT does not utilize the levels of priority and severity. To calculate the probabilities, we use an Support Vector Machine (SVM) [26] while there are numerous machine learning algorithms such as Naive Bayes [25], C4.5 [27] and so forth. SVM offers strong performance on unknown patterns (high generalization ability) [26] and it would help RABT assign bug reports including a wide variety of words. A prior study [5] has indicated that SVMs are the most accurate for bug assingments.

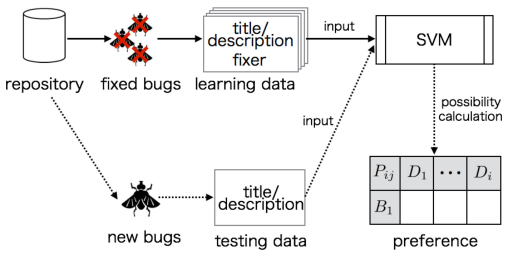

Fig. 2 Preference calculation procedure

Figure 2 shows the preference calculation procedure, and the steps are as follows.

Preparation phase

1. Collect data on fixed bugs from the BTS.
2. Retrieve the fixer and bug description (title and overview) from the data.
3. Train the SVM using fixer/description pairs.

Assignment phase

1. When a new bug B_j is reported, input its description to the previously-generated SVM to obtain the probability (preference P_{ij}) that each developer D_i suitable for fixing it.

3.2.2 Bug-Fixing Cost

The time required to fix bugs depends on which developers are assigned to fix them. Here, the time required for the developer D_i to fix the bug-fixing task B_j is defined as the **bug-fixing cost** C_{ij}. We use historical data to calculate how long it took for developer D_i to fix similar bugs B_j and use this as the cost C_{ij}. In our previous work [28], in order to calculate the approximate bug fixing-time for the costs, we used priority and component tags. Both tags are located in bug tracking systems, the priority tags show the importance of the bug-fixing, and component tags indicate the software parts (which constitute of the product) where the bug appears in. We calculated the median time of bug-fixing as costs by the levels of priority in each component. However, in addition to the calculation of the preference, the calculation of the costs do not use the contents of the bugs, in other words, it ignores what bug it is. Depended on the contents, the bug-fixing time will vary. For example, bugs

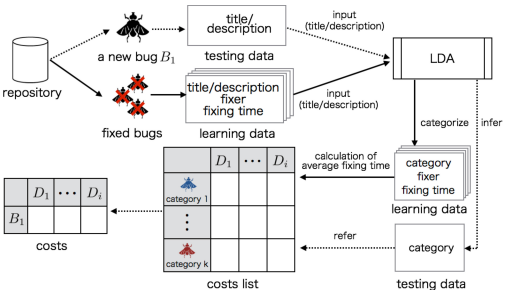

Fig. 3 Procedure for calculating bug-fixing costs

related to security are fixed faster than bugs about performance [34]. However, component tags do not include such information and only the description contains the information. Utilizing the descriptions, we use latent Dirichlet allocation (LDA) [35], which is used in [16], to assess whether a previous bug is of a type similar to the current bug B_j. Since LDA is useful for finding similar documents, it is widely used in the mining software repositories field [36]. The latent semantic indexing (LSI) [37] and pLSI [38] methods are similar to LDA, but LDA is different in that the words and topics are assumed to follow a Dirichlet distribution. The fact that it can handle words that were not in the training set is also very useful, and we chose it due to the high probability of new words appearing in the free description part of the input defect forms. Figure 3 shows the bug-fixing cost calculation procedure, and the steps are as follows.

Preparation phase

1. Collect the bug-fixing data from the BTS.
2. Retrieve the fixer and bug description (title and overview) from the data.
3. Input the free description part of the extracted data to the LDA and categorize the bug (as category k).
4. Calculate the average time taken for each developer to fix bugs in each category (called the cost list).

Assignment phase

1. When a new bug B_j is reported, input its description to the previously-generated LDA and infer the bug's category.
2. Find the average time taken by developer D_i to fix bugs in category k from the cost list and use that as the bug-fixing cost C_{ij}

3.2.3 Upper Limit

Naturally, the number of tasks developers can address in any given period of time is limited. Thus, when assigning bug-fixing tasks we consider the amount of time that developer D_i has available, i.e., the number of bugs they can fix. Figure 4 shows how the tasks are assigned. The number of tasks that can be assigned is obtained from the **available time slot** T_i, which is calculated from an **upper limit L** (per day) set

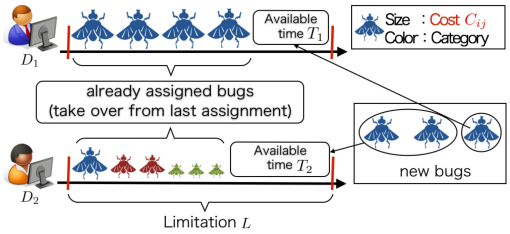

Fig. 4 Calculation of available time slot

in advance and the total cost C_{ij} already assigned to developer D_i.

Ensuring that the total cost of the newly-assigned bug-fixing tasks does not exceed T_i should have the effect of preventing these tasks from concentrating on specific developers. The upper limit L can be changed in size depending on the project. We set the same upper limit for all developers in our experiments, but in practice, this upper limit is likely to be different for each developer, in which case it can be set as L_i for developer D_i.

3.3 Formulation

Here, we define the objective variables, objective function, and constraints.

3.3.1 Objective Variable

The objective variables x_{ij} represent whether our method has assigned bug B_j to developer D_i: if $x_{ij} = 1$, then bug B_j has been assigned to developer D_i, and if $x_{ij} = 0$, then it has not.

$$x_{ij} \in \{0, 1\} \tag{5}$$

3.3.2 Objective Function

The objective is to maximize the total sum of the product of the preferences and objective variables for each bug and each developer. This means that our method finds the best combination of tasks and developers **for the project as a whole and not for individual developers**.

$$Maximize : \sum_{i=1}^{m} \sum_{j=1}^{n} P_{ij} x_{ij} \tag{6}$$

3.3.3 Constraints

Our method imposes two constraints: one is to prevent tasks from concentrating on a small number of experienced developers (Constraint 1), and the other is to avoid assigning a bug to multiple developers (Constraint 2).

Constraint 1: The total cost of the tasks assigned to each developer must not exceed their available time slots.

$$\sum_{j=1}^{n} C_{ij} x_{ij} \leq T_i \quad (i = 1, 2, \ldots, m) \tag{7}$$

Constraint 2: At most, one developer can be assigned to each bug.

$$\sum_{i=1}^{m} x_{ij} \leq 1 \qquad (j = 1, 2, \ldots, n) \tag{8}$$

4. Implementation

4.1 Overview of the Proposed Implementation

In this section, we give an overview of the proposed implementation, as shown in Fig. 5. First, we extract data about fixed bugs from the repository and use them to train an SVM and an LDA. Next, we obtain the cost list (the average time each developer has taken to fix each category of bugs) using the LDA.

When a new bug B_n is reported, we input its description to the SVM and LDA to obtain the preferences P_{in} for all developers and its category k. Then, we find the costs C_{in} for all developers from the cost list using the category k. Finally, we obtain each developer's available time slots T_i based on the (pre-determined) upper limit L and the total cost of the bugs already assigned to them. We can then determine the developer to be assigned based on the preferences P_{ij}, the costs C_{ij} and the available time slots T_i.

4.2 Procedure for the Release-Aware Bug Triaging Method (RABT)

Here, we describe how to use our method for daily bug assignment. The procedure is as follows.

Step 1: Set the parameters
Set the upper limit L in advance and initialize the available time slots T_i for each developer to L.
Step 2: Construct the SVM and LDA
Construct the SVM and LDA to compute the preferences P_{ij} and costs C_{ij} for each bug B_j and developer D_i.
Step 3: Compute the preferences and costs
Calculate the preferences and costs for each newly reported or unassigned bug.
Step 4: Increment T_i by n (days)
Add n (number of days from the last assignment date to this assignment date) to each developer's T_i (up to a maximum of L). If it is the first assignment, this step will be skipped.

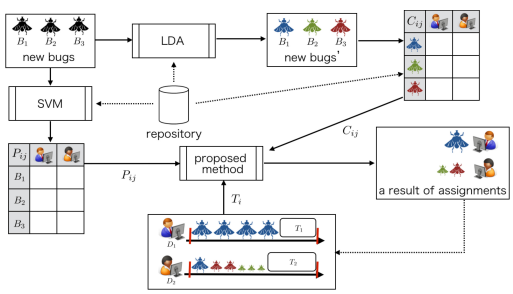

Fig. 5 Overview of the release-aware bug triaging method (RABT)

Step 5: Apply 0-1 application of integer programming
Assign these bugs to developers using the method described in the previous section.
Step 6: Update T_i
Reduce the number of available time slots T_i for each developer by the cost of the bugs assigned in Step 4.
Step 7: Go to the next assignment day (to Step 2)
Once the next assignment day comes, proceed to Step 2.

Here, the value of n (> 0) depends on the task assignment process and needs of each project and is difficult to uniquely determine. In this paper, we assume that n is a natural number, arbitrarily decided by the method's users, to keep the discussion general.

5. Experimental Design

5.1 Overview and Aims

We prepare four evaluations to investigate whether RABT could improve bug-fixing efficiency by assigning tasks to appropriate developers and considering the time they had available for bug-fixing. In Evaluation I, we make sure whether RABT can prevent tasks being concentrated on certain developers, with comparing the existing methods. In Evaluation II, comparing the existing methods, we confirm that RABT can reduce the numbers of overdue bugs (which are assigned but fixed after the release). In Evaluation III, we compare manual assignment (actual bug-fixing time) with the existing methods and RABT to see whether they could reduce bug-fixing delays. In Evaluation IV, we check whether the existing methods or RABT can assign bugs to suitable developers and prevent reassignment, which is the most significant cause of bug-fixing delays. Note that we do not compare our previous work [28] with the proposed work in this study and other existing works although this evaluation might show the difference between our current work and our previous work. This is because our previous method use components tag in the bug reports to assign bugs, which is far from typical bug-triage methods. Most of the bug-triaging studies use the description of the bugs to assign bugs and use the components to measure whether the assignment is appropriate. Basically, developers assign the bugs after reading the description of bug reports rather than component tags, therefore, using the description would be more realistic.

5.2 Datasets

We conducted a case study on three large OSS projects (Mozilla Firefox [39], the Eclipse Platform [40], and GNU compiler collection (GCC) [41]). Each of these is a long-established project, allowing us to acquire sufficient data for the experiment. In addition, many previous studies [5], [6], [14]–[16], [22], [42]–[45] have analyzed data from these projects, enabling us to validate the results obtained in this case study.

Table 2 Datasets

Project	Dataset	Date	Period	No. of bugs
Firefox	Training data	Jul. 5, 2010 – Jul. 4, 2011	1 year	1,043
	Testing data	Jul. 5, 2011 – Sep. 27, 2011	12 weeks	142
Eclipse Platform	Training data	Mar. 22, 2010 – Mar. 21, 2011	1 year	783
	Testing data	Mar. 22, 2011 – Jun. 22, 2011	3 months	168
GCC	Training data	Dec. 25, 2009 – Dec. 24, 2010	1 year	940
	Testing data	Dec. 25, 2010 – Mar. 25, 2011	3 months	250

Table 3 Dataset contents after each filtering step

	Filter	Firefox		Eclipse Platform		GCC	
		Training	Testing	Training	Testing	Training	Testing
A	Bugs collected from each project	14,915	2,254	3,503	847	4,191	1,173
B	Of (A), fixed bugs with known fixing-times	1,769	238	1,121	250	1,324	377
C	Of (B), bugs whose fixing-time is not an outlier value	1,568	211	946	202	1,116	320
D	Of (C), bugs fixed by active developers	1,043	142	783	168	940	250

Table 4 Statistics of fixing-time for each dataset

Project	Firefox	Platform	Gcc
# of bugs	1,185	951	1,190
Percentage of bugs in which the bug-fixing time is less than 2 days	19.2	49.2	51.2
average days to fix	12.5	6	4.4
median days to fix	7	2	1.9
minimum days to fix	1	1	1
maximal days to fix	59.9	38.5	27.5

Table 5 Active developers in each dataset

Projects	# of all developers	# of active developers
Firefox	215	19
Platform	61	20
Gcc	97	23

Table 2 outlines the datasets used, whereas Table 3 lists the filtering performed to create them and the number of bugs in each. Table 4 shows their statistics of bug-fixing days. Of all the bugs collected from each project, we only considered fixed bugs (i.e., bugs whose status was FIXED) where the fixer and fixing time could be identified. Some of the bugs were only fixed after several years, so we removed these outliers by confirming the fixing time distributions using boxplots.

In this study, we assigned bug-fixing tasks to developers using existing methods and RABT. However, assigning tasks to all of a project's developers is not realistic because OSS projects developers are often known to leave projects in a relatively short period of time [46]. Moreover, since not all developers actively fix bugs [43], tasks should necessarily only be assigned to developers who are likely to be in charge of bug-fixing tasks. Hence, we only assigned defined tasks to developers who had fixed six or more bugs within six months of their first assignment (i.e., fixed at least one bug per month), thus considering these developers to be "active" (Table 5). To guarantee the accuracy of task assignment, all bug reports fixed by non-target developers were excluded.

In this study, we prepared both learning and evaluation datasets, using one year of data (from the first assignment day) as training data for all projects, and 12 weeks of data (Firefox) and three months of data (Eclipse and GCC) before release as evaluation data. Only 12 weeks of Firefox evaluation data was used because the Firefox project has adopted a rapid release method [47] with a test period of 12 weeks for each release. In contrast, three months of evaluation data of Eclipse and GCC was used due to the large number of bug reports filed in the three months before release.

5.3 Comparison Methods

We compared the bug-fixing time of RABT with that of a manual assignment method and two existing methods (CBR and CosTriage). Among the machine learning algorithms used for CBR and CosTriage, we used the SVM-based method [5] that was found to give the most accurate recommendations.

5.4 Evaluations

We evaluate RABT in four different ways as described below.

Evaluation I: Prevention of task concentration
We confirm whether the number of tasks (bug-fixing time) assigned to each developer by the existing methods and RABT is higher for certain developers. Here, as an evaluation criterion, the bug-fixing time should not exceed the evaluation data period for each project.

Evaluation II: Reducing overdue bugs for the release
We confirm the numbers of bugs that assigned but fixed after the release ("# of overdue bugs"). Note that "# of overdue bugs" is different from the task concentration in Evaluation I, which shows the total time that developers devote fixing bugs in the period. Therefore, even if the task concentration did not happen in Evaluation I, if the bugs were assigned immediately before the release, "# of overdue bugs" might be more than zero.

Evaluation III: Reduction of overall bug-fixing time for

the project
We confirm whether the existing methods or RABT can improve bug-fixing efficiency by comparing their estimated bug-fixing times with the actual recorded times.
Evaluation IV: Accuracy of assignments
We evaluate to what extent the accuracy of assignments by RABT decreases compared to CBR. CBR assigns each bug to the most suitable developer (with the largest preference), whereas RABT assigns bugs to developers so that the total preferences for the project are the highest. Hence, we can assume that RABT will lower the accuracy of the assignment.

The accuracy of assignments measures a rate of the number of appropriate assignments and the number of all assignments. The appropriate assignment is defined as an assignment to the developer who has experienced fixing bugs with the same component as the target bug report. The components are software parts constituting of the product. The bug tracking systems in Eclipse, Firefox, and Gcc have the tag indicating which component includes the bug.

Here, several works evaluate their methods with top-X accuracy which is the performance measure how many developers are correctly selected when recommending multiple developers for a bug. We cannot use this because our bug assignments to developers are executed at the same time and determined dynamically, that is, each assignment is dependent on to whom each other's bug is assigned.

5.5 Experimental Procedure

In this experiment, tasks were assigned using both the existing methods and RABT, and the bug-fixing times were calculated based on the resulting assignments. An overview of the experiment is shown in Fig. 6.

We extracted the bug reports for each date in the evaluation data and used both RABT and the existing methods to assign the bugs day by day according to their reported date. Also, the assigned bugs to each developer are considered to be fixed in the order of the assignments. Developers' available time slots T_i will be incremented by one (T_i never surpass upper limit L) before assigning bugs.

Once assignments had been made for all days, the bug-fixing times were calculated (Fig. 6, right). Since the assignment methods considered here do not always assign the bugs to the developers who actually fixed them (i.e., the actual bug-fixing time cannot be calculated), we used the median times taken by the individual developers to fix the bugs in each category (that is, the costs C_{ij}) from the training data as the bug-fixing times for the experiment.

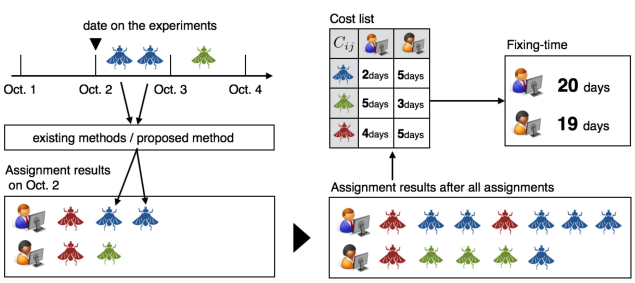

Fig. 6 Overview of experiment

5.5.1 Experimental Environment and Settings

Experimental environment: The open-source mathematical planning software package lp_solve 5.5.2.0 was used to solve the task assignment problem using 0-1 integer programming method, operating on a PC with an Intel Xeon 2.20 GHz CPU and 64 GB of RAM and running CentOS 7.
Parameter settings: RABT requires the upper limit L and the assignment interval (Sect. 4.2, Step 6, n) to be set in advance. Here, the third quartile value of the times required to fix the bugs in the dataset was calculated and rounded, to obtain L values of 15 for Firefox, 6 for the Eclipse Platform, and 6 for GCC. In addition, the interval n was set to 1 (day). While applying LDA, deciding how many bug categories to use for classification was important, so we determined the optimal number of categories for each project using Arun's method [48]. This yielded 7 for Firefox, 12 for the Eclipse Platform, and 11 for GCC.
Experimental settings: The procedure of RABT includes the recalculation process of preferences and costs (Step2). However, if we use this recalculation in the evaluation, we cannot compare the three methods under the same condition because the preference and cost gradually vary as the simulation progresses. In order to prevent from changing the cost and preference during the experiment, we return to Step 3 rather than Step 2 after Step 7 in this experiment.

As in a previous study [16], the bug-fixing times for prior bugs were obtained as follows.

$$\text{fixing time} = \text{fix day} - \text{assignment day} + 1 \text{ day} \quad (9)$$
*round down to the nearest decimal

The assignment date is the date when the bug was assigned to the developer who fixed it. In other words, we do not include the time spent by previous developers in attempting to fix the bug (the reassignment time) here.

6. Result

6.1 Preliminary Experiment: Evaluation of a Method to Calculate Bug-Fixing Time

To the best our knowledge, no software provides us with a simulation of bug-fixing activity. In this experiment, we can not time the bug-fixing time if tasks are assigned to developers other than the developer who actually fixed the bug. This is the reason why costs are substituted as bug-fixing time in this experiment. To use the cost as the bug-fixing time, we confirm whether the cost can substitute as the bug-fixing time in a preliminary experiment.

First, from the bug-fixing history, we prepare two kinds of information ("who fixed which bugs" and "the fixing time"). Using the former information (who fixed which

Table 6 Evaluation of calculations for bug-fixing time

	Actual fixing day	Simulated fixing day	delta	delta per bug
Firefox	1,799	1,643	156	1.1
Platform	822	778	44	0.2
Gcc	1,148	992	156	0.6

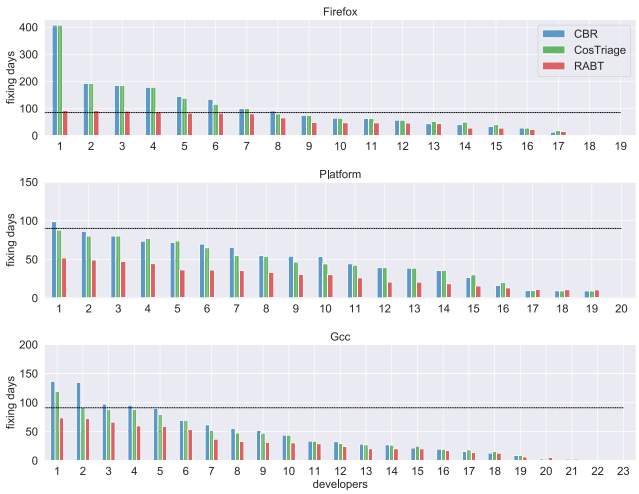

Fig. 7 Fixing time by each developer

Table 7 The statistics of fixing-time assigned by each method

Projects	Firefox		
Methods	CBR	CosTriage	RABT
mean	97.0	96.7	52.5
median	63.6	63.6	46.9
max	407.9	407.9	92.1
min	0.0	0.0	0.0
variance	9393.3	9117.8	960.6
stdev	96.9	95.5	31.0
entropy	3.6	3.7	4.0
Projects	Platform		
Methods	CBR	CosTriage	RABT
mean	46.8	44.8	27.0
median	48.9	43.3	28.0
max	98.6	87.9	51.9
min	0.0	0.0	0.0
variance	828.3	716.7	217.8
stdev	28.8	26.8	14.8
entropy	4.0	4.0	4.1
Projects	Gcc		
Methods	CBR	CosTriage	RABT
mean	45.1	40.4	29.7
median	32.4	29.1	24.3
max	136.5	119.6	73.6
min	0.0	0.0	0.0
variance	1716.4	1167.7	552.4
stdev	41.4	34.2	23.5
entropy	3.9	4.0	4.1

bugs) and cost, we calculate the simulation bug-fixing time. Then, we compare the simulation bug-fixing time and the actual bug-fixing time. As the difference between these two fixing times is smaller, the calculation of the bug-fixing time in this simulation is reasonable.

The results of the experiment are shown in the Table 6. In Firefox 156 days (1.1 days per bug), Platform 44 days (0.2 days), and Gcc 156 days (0.6 days) errors were seen. The error per bug is about one day in Firefox or is less than one day in Platform and Gcc, and can be said to be acceptable.

6.2 Evaluation I: Mitigation of Task Concentration

Figure 7 shows the amount of task (the number of days to fix) that each active developer worked on[†]. The number of developers assigned tasks that require more than the evaluation data period (Firefox: 12 weeks, Platform and Gcc: 3 months) by CBR, is eight developers in Firefox, one developer in Platform, four developers in Gcc. We can see a lot of loads on some developers. Even when using CosTriage, since the number of developers is seven developers in Firefox, no developers in Platform, two developers in Gcc. It shows Costriage mitigate the task concentration compared with CBR, however, tasks are still concentrated on a few developers. In the case of applying RABT, the number of developer is four developers in Firefox, no developers in Platform and Gcc. For all projects, the number of developers is reduced compared with existing methods.

As for the bug-fixing times of the developers who concentrated tasks by RABT and the existing methods, it can be seen that the bug-fixing time of the developer assigned by RABT is significantly reduced (especially, the fixing-times of developers assigned a lot of tasks by existing methods are reduced).

Table 7 summarizes the statistics of the fixing-time that each developer devotes to fixing bugs. The variance of the fixing-time assigned by RABT is smaller than the others in all projects and also entropy is larger, which show RABT can mitigate the tasks more than the traditional methods do.

The existing methods tend to concentrate the task assignment on some developers. Compared with the existing methods, RABT can mitigate the task concentration.

6.3 Evaluation II: Reducing Overdue Bugs for the Release

Table 8 shows the numbers of bugs that assigned but fixed after the release ("# of overdue bugs") and the numbers of un-fixed bugs by the release which is the sum of "# of overdue bugs" and "# of un-assigned bugs".

In Firefox, 37 by the manual assignment method, 77 bugs by CBR, 75 bugs by CosTriage, and 31 bugs by RABT were overdue. Next, in Platform, 53 by the manual assignment method, 21 bugs by CBR, 20 bugs by CosTriage, 14 bugs by RABT. Finally, in Gcc, 64 by the manual assignment method, 52 bugs by CBRs, 40 bugs by CosTriages

[†]Note that the numbers on the horizontal axis represent the order when the developer's task amount (bug-fixing days) is arranged in descending order for each method, therefore different developers even if same axis numbers for each method.

Table 8 Comparing the results of each method

Projects	Firefox			
Methods	Manual	CBR	CosTriage	RABT
# of assigned bugs	142	142	142	123
# of un-assigned bugs	0	0	0	19
# of assigned developers	15	17	17	18
# of overdue bugs	37	77	75	31
# of un-fixed bugs	37	77	75	50
Fixing-days for project	1,702	1,843	1,838	997
Avg. Fixing-days per bug	12.0	13.0	12.9	8.1
Accuracy of assignments	—	81.7	81.0	60.6
Projects	Platform			
Methods	Manual	CBR	CosTriage	RABT
# of assigned bugs	194	194	194	185
# of un-assigned bugs	0	0	0	9
# of assigned developers	19	19	19	19
# of overdue bugs	53	21	20	14
# of un-fixed bugs	53	21	20	23
Fixing-days for project	828	936	897	540
Avg. Fixing-days per bug	4.3	4.8	4.6	2.9
Accuracy of assignments	—	68.0	68.0	60.3
Projects	Gcc			
Methods	Manual	CBR	CosTriage	RABT
# of assigned bugs	250	250	250	246
# of un-assigned bugs	0	0	0	4
# of assigned developers	19	21	20	21
# of overdue bugs	64	52	40	22
# of un-fixed bugs	64	52	40	26
Fixing-days for project	1,085	1,037	929	684
Avg. Fixing-days per bug	4.3	4.1	3.7	2.8
Accuracy of assignments	—	74.4	71.6	66.4

and 22 bugs by RABT. Overall, RABT can assign a more appropriate amount of bugs to each developer compared to the existing methods. Considering the un-assigned bugs, the number of un-fixed bugs by RABT is more than CBR and Costriage in Platform, but is fewer than CBR and Costriage in Firefox and Gcc. We looked into the assigned date of the overdue bugs and found that the overdue bugs by RABT were reported and assigned just before release. For the existing methods, in addition to the reason, the task concentration made overdue bugs.

Compared with CBR and CosTriage, RABT can assign a more appropriate amount of bugs that each developer can fix by the immediate release.

6.4 Evaluation III: Reduction of Bug-Fixing Time

Table 8 shows the bug-fixing time of the project when using the manual assignment method, CBR, CosTriage and RABT, respectively.

In Firefox, the total bug-fixing days for the project is 1,702 days in the manual assignment method, 1,843 days in CBR, 1,838 days in Costriage, 997 days in RABT. CBR increased about 8% (8% = (1,843 - 1,702) / 1,702) of the days compared with the manual assignment method, Costriage also raised about 7% (8% = (1,838 - 1702) / 1,702), RABT could reduce about 41% (-41% = (997 - 1,702) / 1,702) compared to the manual assignment method. Moreover, RABT could reduce about 46% (-46% = (997 - 1,843) / 1,843) of the days compared to CBR, and about 46% (-46% = (997 - 1,838 /1,838) compared to CosTriage.

In Platform, the total number of bug-fixing days for the project is 828 days in the manual assignment method, 936 days in CBR, 897 days in Costriage, 540 days in RABT. CBR increased about 13% (13% = (936 - 828) / 828) of the days compared with the manual assignment method, Costriage raised about 8% (8% = 897 - 828) / 828), RABT could reduce the bug-fixing time of about 35% (-35% = 540 - 828) / 828) compared with the manual assignment method. In addition, RABT could reduce the bug-fixing time of about 42% (-42% = (540 - 936) / 936) compared to CBR, while RABT increased about 40% (-40% = (540 - 897) / 897) of the days compared to CosTriage.

In Gcc, the total number of bug-fixing days for the project is 1,085 days in the manual assignment method, 1,037 days in CBR, 929 days in Costriage, and 684 days in RABT. CBR could reduce about 4% (-4% = (1,037 - 1,085) / 1,037), about 14% (-14% = (929 - 1,085) / 929), RABT can reduce the bug-fixing time of about 37% (-37% = (684 - 1,085) / 684) compared to the manual assignment method. In addition, RABT could reduce the bug-fixing time of about 34% (-34% = (684 - 1,037) / 684) compared with CBR, while increased about 26% (-26% = (684 - 929) / 929) compared to CosTriage.

From the above, RABT can reduce fixing-time less than or equal to CosTriage, but considering the release, RABT can improve bug-fixing activities for the project compared to CosTriage.

Compared to the manual task assignment method, RABT can reduce the bug-fixing time from 35% to 41%.

6.5 Evaluation IV: Accuracy of Assignments

Table 8 also shows the accuracy of assignments by CBR, CosTriage and RABT, respectively. In Firefox, the accuracy of assignments was 81.7% by CBR, 81.0% by CosTriage, 60.6% by RABT. Next, in Platform, CBR was 68.0%, CosTriage was 68.0%, and RABT was 60.3%. Finally, in Gcc, CBR was 74.4%, CosTriage was 71.6%, and RABT was 66.4%. Taking the average accuracy for each of the three methods, CBR is 74.7%, the CosTriage is 73.5%, RABT is 62.4%. In addition, the accuracy of CosTriage was 2% lower than that of CBR, and the accuracy of RABT decreased by 20% compared to CBR.

Although RABT can reduce the bug-fixing time and mitigate the concentration of tasks, it has been found that the assignment accuracy decreases by 20% on average, compared to CBR.

7. Discussion

7.1 The Effect of Lowering Accuracy

Throughout the evaluations, we showed RABT outperforms the existing methods in terms of mitigation of the task concentration, the number of bugs that developers can fix by the next release, the total fixing-time for the project. However, in evaluation IV, we found RABT decreases 20% of the accuracy of assignments, comparing with the existing methods. We concern the effect of lowering the accuracy which induces the reassignments of the bugs. In the following sections, we discuss the cause and effect of the lowering 20% of the accuracy.

7.1.1 The Cause of Lowering Accuracy

There are two conceivable reasons why the accuracy of RABT is lower. The first case is when we still have the other developers that have fixed a bug in the same component (which would be a correct recommendation if methods assign a bug to the developers). RABT would assign bugs to the developers whose preference is not the largest. Thus, in the case that the assignments are inaccurate even though there are alternative developers, this suggests that the second or the third (or so on) recommendations should be improved. Since the number of fixes is considerably different depending on developers, the sizes of the training dataset for each developer also differ. In this experiment, we used the dataset which contains one year of data. This is because the dataset size becomes bigger, the more the existing methods would concentrate bugs on specific developers, therefore we avoid using plenty of the data to equally evaluate. If we use more data, the second or third recommendations would be improved.

Another case is when there is only one developer (there are no alternative developers) in the component. In this case, the preference of the developers that should be assigned by RABT should be 100 (which is the maximum value of preference). To realize the value, we could train the model with the component tags in addition to text data. Although the component tags were used to evaluate the appropriateness of the recommendations in this paper, we can exploit the tags when applying RABT to the actual projects. For both cases, improving the classifier would be an effective option.

7.1.2 Estimation of the Effect of Lowering Accuracy

In this section, in order to estimate the effect of inappropriate assignment, granted that the developer needs double fixing-time when the assignment is not appropriate, we confirm how the results of evaluations vary.

As for Evaluation I, Fig. 8 and Table 9 show the bug-fixing days and the statistics of the fixing-time that each developer devotes to fixing bugs, respectively. In the table, Δ shows the difference between the original results and the penalized results.

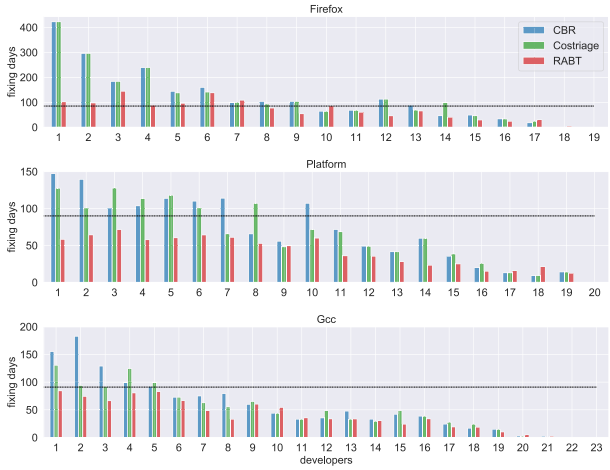

Fig. 8 Penalized fixing time by each developer

Looking into the penalized result of Firefox in Fig. 8, the numbers of developers who devoted to bug-fixing over the evaluation data period (Firefox: 12 weeks, Platform and Gcc: 3 months) are eleven, eleven, and eight assigned by CBR, CosTriage, and RABT, respectively. Compared with the original results (the number of developers were eight, seven, and four respectively), the increases were three, four, and four developers respectively. In Platform, the numbers were eight, seven, and no developers assigned by CBR, CosTriage, and RABT, respectively. Compared with the original results (the number were one developer by CBR and no developers by CBR and RABT), the increases were seven, seven, and no developers. In Gcc, the numbers were five, five, and no developers assigned by CBR, CosTriage, and RABT, respectively. Compared with the original results (the number were four, two, and no developers, respectively), the increases were one, three, and no developers.

Regarding the variance and entropy of the fixing-time for each developer, in all project, the variance of RABT is still smaller than CBR and CosTriage, and the entropy of RABT is larger than CBR and CosTriage, which shows RABT distribute the task more than CBR and CosTriage. Overall, even if the project prepares doubled bug-fixing time for each developer when the assignment is not appropriate, the workloads by assigned by RABT are less affected.

As for Evaluation II and III, Table 10 shows the number of overdue bugs and the fixing days when the fixing-time whose assignments are inappropriate are doubled. Note that the table does not include "# of assigned bugs", "# of unassigned bugs", "# of assigned developers", and "Accuracy of assignments" since they did not change from Table 8.

In terms of the number of overdue bugs, the numbers are increased in all projects and methods. Compared with the original results, the numbers in all project by RABT were smaller than others whereas the increases of RABT were larger than others in Firefox and Gcc.

In terms of the fixing-time, in all projects, the fixing-

Table 9 The statistics of penalized fixing-time assigned by each method

Projects	Firefox		
Methods	CBR	CosTriage	RABT
mean	117.5	117.6	67.7
Δ	20.5	20.9	15.2
median	99.4	99.2	65.3
Δ	35.8	35.6	18.4
max	422.7	422.7	144.8
Δ	14.8	14.8	52.7
min	0.0	0.0	0.0
Δ	0.0	0.0	0.0
variance	11575.2	11281.4	1745.5
Δ	2181.9	2163.6	784.9
stdev	107.6	106.2	41.8
Δ	10.7	10.7	10.8
entropy	3.7	3.7	3.9
Δ	0.1	0.0	-0.1

Projects	Platform		
Methods	CBR	CosTriage	RABT
mean	68.6	65.0	40.6
Δ	21.8	20.2	13.6
median	62.8	62.8	42.8
Δ	13.9	19.5	14.8
max	147.2	127.8	71.5
Δ	48.6	39.9	19.6
min	0.0	0.0	0.0
Δ	0.0	0.0	0.0
variance	2104.5	1759.9	472.8
Δ	1276.2	1043.2	255
stdev	45.9	42.0	21.7
Δ	17.1	15.2	6.9
entropy	4.0	4.0	4.1
Δ	0.0	0.0	0.0

Projects	Gcc		
Methods	CBR	CosTriage	RABT
mean	55.6	49.5	39.0
Δ	10.5	9.1	9.3
median	41.7	43.5	33.6
Δ	9.3	14.4	9.3
max	183.0	130.5	84.4
Δ	46.5	10.9	10.8
min	0.0	0.0	0.0
Δ	0.0	0.0	0.0
variance	2469.2	1477.2	770.2
Δ	752.8	309.5	217.8
stdev	49.7	38.4	27.8
Δ	8.3	4.2	4.3
entropy	4.0	4.0	4.1
Δ	0.1	0.0	0.0

Table 10 Comparing the penalized results of each method

Projects	Firefox		
Methods	CBR	CosTriage	RABT
# of overdue bugs	87	83	43
Δ	10	8	12
# of un-fixed bugs	87	83	62
Δ	10	8	12
Fixing-days for project	2232	2235	1286
Δ	389	397	289
Avg. Fixing-days per bug	15.7	15.7	10.5
Δ	2.7	2.8	2.4

Projects	Platform		
Methods	CBR	CosTriage	RABT
# of overdue bugs	49	42	20
Δ	28	22	6
# of un-fixed bugs	49	42	29
Δ	28	22	6
Fixing-days for project	1372	1301	812
Δ	436	404	272
Avg. Fixing-days per bug	7.1	6.7	4.4
Δ	2.8	1.9	1.5

Projects	Gcc		
Methods	CBR	CosTriage	RABT
# of overdue bugs	67	57	43
Δ	15	17	21
# of un-fixed bugs	67	57	47
Δ	15	17	21
Fixing-days for project	1280	1139	897
Δ	243	210	213
Avg. Fixing-days per bug	5.1	4.6	3.6
Δ	1.0	0.9	0.8

time for project and the average fixing-days per bug, which is assigned by RABT, are still smaller than others.

As long as the fixing-time is doubled when the assignment is inappropriate, the estimated effect of lower accuracy than CBR and CosTriage is not tremendous.

7.2 How to Handle Unassigned Bugs?

Evaluation II shows that RABT decreases the number of overdue bugs in all projects, compared with existing methods. However, in all project, there were unassigned bugs and we need to discuss how we should handle the unassigned bugs. We believe intently not assigning bugs will be a compromise option because fixing all reported bugs in modern software development is far from easy. Therefore, RABT has the possibility to provide projects with practical assignments. To realize the practical assignments for the future work, RABT needs to utilize the priority and severity tags in the bugs reports (specifically, bug priority predictions [49], [50] and severity prediction [51], [52]) in order to prevent higher priority bugs from remaining unassigned rather than lower priority bugs[†]. For instance, according to the levels of the priority (i.e., P1 [the highest priority], P2, P3, P4, P5 [the lowest priority]), constant values (e.g., 500, 400, 300, 200, 100, respectively) should be added to the preferences in order to prioritize bugs. By utilizing priority or severity tags, it provides dispatchers with a new approach to decide which bugs should be assigned and which bugs should not be assigned for the next release, which is far from general developer recommendations.

7.3 How to Set Appropriate Limit L?

In this experiment, we set the upper limit $L = 6$ for Platform and Gcc, 15 for Firefox. However, it is unclear what impact the size of the upper limit L has on the project. Hence, we confirm how the accuracy of assignments and the number of overdue bugs vary with the size of the upper limit L. Figure 9 is the accuracy of assignments and the number of overdue bugs for each upper limit L. The accuracies dynamically increase from 1 to 14 for Firefox, from 1 to 5 for Platform, from 1 to 5. After that, the accuracies gradually increase until L is 31. Compared with the accuracies, in all projects, the number of overdue bugs constantly (not dynamically) raise while L increases. Therefore, we can say that parameter L should be decided based on the accuracy. Based on the accuracy, the points of which the increase of

[†]As described in Sect. 3.2.1, most of the bugs have incorrect priority and severity. Only projects that correctly label bugs with priorities and severities should utilize them.

Fig. 9 Relationship between the size of L and the accuracy and the number of overdue bugs

the accuracy calm down are 15 for Firefox and 6 for Platform. Thus, our way to set L, which is referring to the 3rd quartile, might be appropriate.

7.4 How to Deal with Irregular Situations

As RABT is designed for automating usual bug assignments, developers would have to help RABT to assign bugs if unexpected problems happened. In the following, we discuss about likely irregular situations.

Nobody can fix the bug in the project: This situation is likely to happen to any other bug-assignment methods including manual bug-triage, but practically RABT should be more careful about this situation because it aims to automate bug assignments. In the RABT, the preference is relative scores, produced by Support Vector Machine. That is the total of the possibility for each developer will be 1. Thus, even if there are no appropriate developers, RABT (but also CBR and CosTriage) can choose developers who are relatively appropriate among all developers. In advance, the project has to make the rule in case the problem happens. For example, when the assigned developer thinks that no developers in the project can fix the bug, the assigned developer must lead the discussion with the other members about how to handle the bug. By doing this, the project could take measures such as calling for other professionals from outside of the project.

Developers are faced with technical or private problems: In the procedure of RABT, on every assignment, available time slots (T_i) is incremented by the days from the last assignment day to the assignment day. However, if unexpected problems happened, there is a probability that such simply incrementing might not correctly reflect their workloads. For example, given that a developer is taking more time than the estimated time because of technical or private problems, RABT will continue to assign new bugs. In case of unexpected problems, RABT needs a function to stop assigning them (and/or a function to reassign the bugs to the others) when RABT is implemented for applying to practical projects.

As an alternative option instead of the simply increments function (Step 4), RABT could replace a new update function that removes the occupied cost after the bug is fixed in the available slot. Note that, even using the update function, the reassignment function should be required at least because there are cases that developers cannot fix the assigned bugs because of technical problems.

7.5 Limitation

7.5.1 Fix Order of Bugs

In Experiment II, by comparing the number of bugs fixed by the release with RABT and the two existing methods, we confirmed that the existing methods remain many bugs not fixed by the release. However, the number of bugs not fixed by the release depends on the order in which the bugs are fixed. In other words, if the bugs which are long bug-fixing time was assigned in the early time of the experiment, the number of the bugs not fixed by the release will increase.

7.5.2 Impact of Mitigating Task Concentration

In the experiment, we could confirm the effect of mitigating the task concentration in RABT. However, mitigating task concentration of some developers is also that other developers are assigned the tasks. Even though developers who are relatively not in charge of tasks seem to have a scope at first glance, they may have other development projects or volunteers, so the time of activities may be limited. Hence, developers who do not have many tasks are not necessarily in a condition that can handle tasks. Since RABT has not ascertained how long it can participate in the bug-fixing activity in the month, the developer might be forced an excessive loads.

8. Conclusion

In this paper, we proposed a release-aware bug triaging method (RABT) that aims to increase the number of bugs that developers can fix by the next release date. Existing methods tend to concentrate assignments of bug-fixing task to a small number of developers because it does not consider the difficulty and costs of individual bug-fixing. Since general software development has the releases, even an experienced developer can finish the bug-fixing work that can be used until the next release. Hence, the existing methods are not practical.

RABT is characterized by considering the upper limit of the amount of task that developers can work on during a certain period, in addition to the ability of developers. In this method, we considered the bug assignment problem as a multi-knapsack problem, finding a combination of bugs and developers that maximize developers' ability under constraints which the method can assign in the only time that developers can use for bug-fixing work. As a result of a

case study on Mozilla Firefox, Eclipse Platform, GNU Gcc project, the following three effects on the proposed method were confirmed.

(1) RABT mitigates the situation where bug-fixing tasks are concentrated to a small number of developers
(2) RABT can assign a more appropriate amount of bugs that each developer can fix by the next release date
(3) RABT can reduce the time to fix bugs, compared with the manual bug triaging method and the existing methods

Acknowledgments

This research is conducted as part of Grant-in-Aid for Japan Society for the Promotion of Science (JSPS) Research Fellow and Scientific Research JP17J03330, JP17H00731 and JP18K11243.

References

[1] A. Endres and D. Rombach, A handbook of software and systems engineering: empirical observations, laws and theories, Pearson/Addison Wesley, 2003.
[2] E.S. Raymond, The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary, Oreilly & Associates Inc., 2001.
[3] "Bugzilla." https://www.bugzilla.org.
[4] "Jira." https://www.atlassian.com/software/jira.
[5] J. Anvik and G.C. Murphy, "Reducing the effort of bug report triage: Recommenders for development-oriented decisions," ACM Transactions on Software Engineering and Methodology, vol.20, no.3, pp.1–35, 2011.
[6] P. Bhattacharya and I. Neamtiu, "Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging," Proc. 26th IEEE International Conference on Software Maintenance (ICSM'10), pp.1–10, 2010.
[7] G. Bortis and A. van der Hoek, "Porchlight: A tag-based approach to bug triaging," Proc. 2013 International Conference on Software Engineering (ICSE'13), pp.342–351, 2013.
[8] H. Kagdi, M. Gethers, D. Poshyvanyk, and M. Hammad, "Assigning change requests to software developers," Software: Evolution and Process, vol.24, no.1, pp.3–33, 2012.
[9] Z. Lin, F. Shu, Y. Yang, C. Hu, and Q. Wang, "An empirical study on bug assignment automation using chinese bug data," Proc. 3rd International Symposium on Empirical Software Engineering and Measurement (ESEM'09), pp.451–455, 2009.
[10] D. Matter, A. Kuhn, and O. Nierstrasz, "Assigning bug reports using a vocabulary-based expertise model of developers," Proc. 6th IEEE International Working Conference on Mining Software Repositories (MSR'09), pp.131–140, 2009.
[11] H. Naguib, N. Narayan, B. Brügge, and D. Helal, "Bug report assignee recommendation using activity profiles," Proc. 10th Working Conference on Mining Software Repositories (MSR'13), pp.22–30, 2013.
[12] M.M. Rahman, G. Ruhe, and T. Zimmermann, "Optimized assignment of developers for fixing bugs an initial evaluation for eclipse projects," Proc. 3rd International Symposium on Empirical Software Engineering and Measurement (ESEM'09), pp.439–442, 2009.
[13] R. Shokripour, Z.M. Kasirun, S. Zamani, and J. Anvik, "Automatic bug assignment using information extraction methods," Proc. 2012 International Conference on Advanced Computer Science Applications and Technologies (ACSAT'12), pp.144–149, 2012.
[14] A. Tamrawi, T.T. Nguyen, J.M. Al-Kofahi, and T.N. Nguyen, "Fuzzy set and cache-based approach for bug triaging," Proc. joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'11), pp.365–375, 2011.
[15] G. Jeong, S. Kim, and T. Zimmermann, "Improving bug triage with bug tossing graphs," Proc. 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'09), pp.111–120, 2009.
[16] J. Park, M. Lee, H.S. Kim, Jinhan, and S. Kim, "Costriage: A cost-aware triage algorithm for bug reporting systems," Proc. Twenty-Fifth Conference on Artificial Intelligence (AAAI'11), pp.139–144, 2011.
[17] T. Zhang and B. Lee, "A hybrid bug triage algorithm for developer recommendation," Proc. 28th Annual ACM Symposium on Applied Computing, pp.1088–1094, 2013.
[18] P. Bhattacharya, I. Neamtiu, and M. Faloutsos, "Determining developers' expertise and role: A graph hierarchy-based approach," Proc. 30th International Conference on Software Maintenance and Evolution (ICSME'14), pp.11–20, 2014.
[19] D.W. McDonald, "Evaluating expertise recommendations," Proc. 2001 International ACM SIGGROUP Conference on Supporting Group Work (GROUP'01), pp.214–223, 2001.
[20] T.T. Nguyen, T.N. Nguyen, E. Duesterwald, T. Klinger, and P. Santhanam, "Inferring developer expertise through defect analysis," Proc. 34th International Conference on Software Engineering (ICSE'12), pp.1297–1300, 2012.
[21] R. Robbes and D. Röthlisberger, "Using developer interaction data to compare expertise metrics," Proc. 10th Working Conference on Mining Software (MSR'13), pp.297–300, 2013.
[22] J. Anvik, L. Hiew, and G.C. Murphy, "Who should fix this bug?," Proc. 28th International Conference on Software Engineering (ICSE'06), pp.361–370, 2006.
[23] J. Xuan, H. Jiang, Z. Ren, and W. Zou, "Developer prioritization in bug repositories," Proc. 34th International Conference on Software Engineering (ICSE'12), pp.25–35, 2012.
[24] P.J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy, "'not my bug!' and other reasons for software bug report reassignments," Proc. 2011 ACM Conference on Computer Supported Cooperative Work (CSCW'11), pp.395–404, 2011.
[25] G. John and P. Langley, "Estimating continuous distributions in bayesian classifiers," Proc. Eleventh Conference on Uncertainty in Artificial Intelligence (UAI'95), pp.338–345, 1995.
[26] S.R. Gunn, "Support vector machines for classification and regression," tech. rep., University of Southampton, Faculty of Engineering, Science and Mathematics; School of Electronics and Computer Science, University of Southampton, 1998.
[27] J.R. Quinlan, C4.5: programs for machine learning, Morgan Kaufmann Publishers Inc., San Francisco, 1993.
[28] Y. Kashiwa, M. Ohira, H. Aman, and Y. Kamei, "A bugtriaging method for reducing the time to fix bugs in large-scale open source software development," Journal of Information Processing Society of Japan, vol.56, no.2, pp.669–781, 2015, (in Japanese).
[29] S. Martello and P. Toth, Knapsack Problems: Algorithms and Computer Implementations, John Wiley & Sons, Inc., New York, 1990.
[30] D. Pisinger, Algorithms for Knapsack Problems, Department of Computer Science, University of Copenhagen, 1995.
[31] "Lpsolve." http://lpsolve.sourceforge.net/5.5/.
[32] N. Kumar Nagwani and S. Verma, "Rank-Me: A Java Tool for Ranking Team Members in Software Bug Repositories," Journal of Software Engineering and Applications, vol.5, no.4, pp.255–261, 2012.
[33] R.K. Saha, J. Lawall, S. Khurshid, and D.E. Perry, "Are these bugs really 'normal'?," IEEE International Working Conference on Mining Software Repositories, pp.258–268, 2015.
[34] S. Zaman, B. Adams, and A.E. Hassan, "Security versus performance bugs: A case study on firefox," Proc. 8th Working Conference on Mining Software Repositories (MSR'11), pp.93–102, 2011.
[35] D.M. Blei, A.Y. Ng, and M.I. Jordan, "Latent dirichlet allocation," Machine Learning Research, vol.3, pp.993–1022, 2003.

[36] T.T. Nguyen, A.T. Nguyen, and T.N. Nguyen, "Topic-based, time-aware bug assignment," ACM SIGSOFT Software Engineering Notes, vol.39, no.1, pp.1–4, 2014.
[37] S. Deerwester, S.T. Dumais, G.W. Furnas, T.K. Landauer, and R. Harshman, "Indexing by latent semantic analysis," Journal of the American Society of Information Science, vol.41, no.6, pp.391–407, 1990.
[38] T. Hofmann, "Probabilistic latent semantic indexing," Proc. 22nd annual international ACM SIGIR conference on Research and development in information retrieval, pp.50–57, 1999.
[39] "Mozilla firefox." https://www.mozilla.org/en-US/firefox/new/.
[40] "Eclipse platform." https://projects.eclipse.org/projects/eclipse.platform.
[41] "Gnu gcc." https://gcc.gnu.org/.
[42] D. Cubranic and G.C. Murphy, "Automatic bug triage using text categorization," Proc. Sixteenth International Conference on Software Engineering & Knowledge Engineering (SEKE'04), pp.92–97, 2004.
[43] A. Mockus, R.T. Fielding, and J.D. Herbsleb, "Two case studies of open source software development: Apache and mozilla," ACM Transactions on Software Engineering and Methodology, vol.11, no.3, pp.309–346, 2002.
[44] C. Sun, D. Lo, X. Wang, J. Jiang, and S. Khoo, "A discriminative model approach for accurate duplicate bug report retrieval," Proc. 32nd ACM/IEEE International Conference on Software Engineering (ICSE'10), pp.45–54, 2010.
[45] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, "An approach to detecting duplicate bug reports using natural language and execution information," Proc. 30th International Conference on Software Engineering (ICSE'08), pp.461–470, 2008.
[46] C. Bird, A. Gourley, P. Devanbu, A. Swaminathan, and G. Hsu, "Open borders? immigration in open source projects," Proc. 4th International Workshop on Mining Software Repositories (MSR'07), p.6, 2007.
[47] M. Mäntylä, F. Khomh, B. Adams, E. Engstrom, and K. Petersen, "On rapid releases and software testing," Proc. 29th IEEE International Conference on Software Maintenance (ICSM'13), pp.20–29, 2013.
[48] R. Arun, S. Vommina, C.V. Madhavan, and M.N. Murthy, "On finding the natural number of topics with latent dirichlet allocation: Some observations," Proc. 14th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining (PAKDD'10), pp.391–402, 2010.
[49] Y. Tian, D. Lo, and C. Sun, "DRONE: Predicting priority of reported bugs by multi-factor analysis," IEEE International Conference on Software Maintenance, ICSM, pp.200–209, 2013.
[50] J. Kanwal and O. Maqbool, "Bug prioritization to facilitate bug report triage," Journal of Computer Science and Technology, vol.27, no.2, pp.397–412, 2012.
[51] T. Menzies and A. Marcus, "Automated severity assessment of software defect reports," Proc. 24th IEEE International Conference on Software Maintenance, pp.346–355, 2008.
[52] R. Malhotra, N. Kapoor, R. Jain, and S. Biyani, "Severity Assessment of Software Defect Reports using Text Classification," International Journal of Computer Applications, vol.83, no.11, pp.13–16, 2013.

Yutaro Kashiwa received his B.E. and M.E. degrees in engineering from Wakayama University in 2013 and 2015 respectively. He worked for Hitachi, Ltd. as a full-time software engineer for two years. He has been a Ph.D. student at Wakayama University and a JSPS research fellow since 2017. His research interests include bug triaging and software release engineering. He is a member of IEEE.

Masao Ohira received his Ph.D. degree from Nara Institute of Science and Technology, Japan in 2003. Dr. Ohira is currently Associate Professor at Wakayama University, Japan. He is interested in software maintenance and software repository mining. He is a director of Open Source Software Engineering (OSSE) Laboratory at Wakayama University. He is a member of ACM and IEEE.