An Empirical Study on Recommendations of Similar Bugs

Henrique Rocha*, Marco Tulio Valente*, Humberto Marques-Neto†, and Gail C. Murphy‡
*Federal University of Minas Gerais, Brazil
†Pontifical Catholic University of Minas Gerais, Brazil
‡University of British Columbia, Canada
{henrique.rocha, mtov}@dcc.ufmg.br, humberto@pucminas.br, murphy@cs.ubc.ca

***Abstract*—The work to be performed on open source systems, whether feature developments or defects, is typically described as an issue (or bug). Developers self-select bugs from the many open bugs in a repository when they wish to perform work on the system. This paper evaluates a recommender, called NextBug, that considers the textual similarity of bug descriptions to predict bugs that require handling of similar code fragments. First, we evaluate this recommender using 69 projects in the Mozilla ecosystem. We show that for detecting similar bugs, a technique that considers just the bug components and short descriptions perform just as well as a more complex technique that considers other features. Second, we report a field study where we monitored the bugs fixed for Mozilla during a week. We sent mails to the developers who fixed these bugs, asking whether they would consider working on the recommendations provided by NextBug; 39 developers (59%) stated that they would consider working on these recommendations; 44 developers (67%) also expressed interest in seeing the recommendations in their bug tracking system.**

I. Introduction

Many open source systems are the result of sustained development effort. For example, from January 2009 to October 2012, Mozilla developers fixed on average 2,837 bugs per month. This development involved 2,221 developers, most of them working as volunteers [1]. In this setting, each developer typically self-selects the bugs on which to work as they appear in a tracking system.[1] This process differs from most commercial software where managers or team leaders are often responsible to assign and monitor the work performed by the development team.

In this paper, we evaluate whether developers contributing to open source projects can work more efficiently if support is provided to suggest on which bug to work next, namely bugs that are more likely to require changes to the same parts of the system. By working on similar bugs, developers may avoid the effort of locating and understanding very different code fragments for bugs on which they volunteer to work on [2], [3]. Moreover, since many bugs appear daily in large open source projects, a developer may not notice bugs in the tracking system that are very similar to one he/she is currently working on. To clarify our motivation, consider the following Mozilla bug that was fixed by a developer on February 12, 2009:[2]

Bug 475327 - [Linux] New Tab button is still right side of the Tab bar

When the developer fixed Bug 475327, a bug with a similar description was also available in Mozilla's tracking system.

Bug 474908 - Dragging a tab in a window with only one tab detaches the tab

This second bug was fixed by the same developer of the first one, 24 days after the first bug. Fixing the second bug required changes in the exact XML file that was changed by the developer when fixing the first bug. In the interval between working on these bugs, the developer in question remained active, fixing another 12 non-related bugs. To find out how often such a situation occurs, we retrospectively analyzed bugs reported for Mozilla systems from January 2009 to October 2012. For 67% of the bugs we analyzed, just after the developers fixed a bug B, they chose to work on a bug B'' even though a more similar bug B' in terms of changed files was available in the tracking system.

In this paper, we evaluate whether similarity in textual bug descriptions can be used to predict bugs that require handling of similar code fragments, which we refer as similar bugs. For this purpose, we evaluate the recommendations produced by NextBug [4], which is a plug-in that extends Bugzilla—a widely used bug tracking system—with recommendations of similar bugs. The recommendations are presented each time a developer browses the page containing the main description of a bug. The intention is to remind the developer of other open bugs which are similar to the one he is browsing. In our previous motivating example, the page describing Bug 475327 is extended by NextBug with a list of bugs with a similar textual description, like Bug 474908.

To broadly assess whether the recommendations produced are good recommendations, we first report on a study where we compared the recommendations provided by NextBug and REP [5], a state-of-the-art duplicated bug detection technique. We rely on REP to check whether the detection of similar bugs can be initially seen as special case of the problem of

[1]In this paper, we preferably use the term bug to refer to work items such as tasks, defects, enhancements, tickets, etc. The main reason is that bug is the term used in the Mozilla systems we evaluate in the paper. However, to keep compatibility with related work sometimes we also use tasks and issues.

[2]bugzilla.mozilla.org/show_bug.cgi?id=475327, verified 2015-11-05.

detecting identical bugs. In this study, we use 65,590 bugs reported for 69 systems maintained by the Mozilla Foundation. Although REP shows a higher number of recommendations, both techniques showed similar values for precision and recall. NextBug has a precision of 70% for top-1 recommendations while REP's precision is 71%. Considering top-3 recommendations, NextBug has a likelihood of 82% and REP has a likelihood of 81%. Furthermore, to determine if the lists of similar bugs provided by NextBug are useful to open source developers, we conducted a week-long field study in the Mozilla ecosystem. In this study, we presented by email a personalized list of similar bugs to 176 developers. 37% (66 out of 176 developers) responded to our field study and 59% (39 out of 66 developers) stated that they would consider working on at least one of the bugs in the recommendation lists. Also 67% (44 out of 66 developers) expressed interest in seeing these recommendations in the bug system they use.

This paper makes two major contributions:

- First, we show that when detecting similar bugs (i.e., bugs requiring changes in the same parts of a system), a technique that considers just the bug component and short description performs as well as techniques optimized for identifying duplicated bug reports, which usually consider other fields and compute their recommendations using more complex ranking functions.
- Second, we evaluate NextBug in the field, with open source software developers. Based on the results of this study, we clarify the applicability, benefits, and limitations of recommendations of similar bugs.

The remainder of this paper is organized as follows. Section II presents NextBug and its underlying recommendation algorithm. Section III describes a comparative study and evaluates the similar bug recommendations provided by NextBug and REP. Section IV presents the results of a field study with Mozilla developers. We discuss the benefits and limitations of similar bug recommendations in Section V. Section VI discusses related work. Finally, we present the conclusions and outline future work ideas in Section VII.

II. NextBug: A Similar Bug Recommender

Our central goal is to evaluate a tool to recommend similar bugs to open source developers, called NextBug [4]. Figure 1 illustrates this tool. Specifically, NextBug relies on standard information retrieval techniques for preprocessing a *query* q (representing the short description of a bug a developer is interested on) and a collection of *documents* B (representing the short descriptions of the pending bugs in the tracking system referring to the same component of q). After this, NextBug computes the cosine similarity between q and each $b \in B$ and verifies whether the similarity measures passes a predefined threshold τ to recommend b as a similar bug to q. Therefore, NextBug does not require any training or tuning phase prior to use.

Figure 2 shows a recommended list of similar bugs to work on provided by NextBug (which is currently implemented as a Bugzilla plug-in). This figure shows the similar bugs

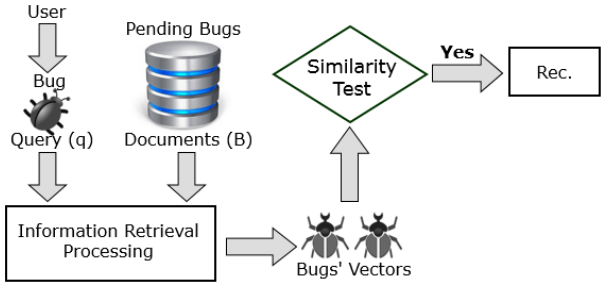


Fig. 1: NextBug algorithm for recommending similar bugs.

information for a bug related to the camera of a mobile device running FirefoxOS (Bug 937928 – "camera app freezes after switching between video and camera"). Three other bugs related to mobile device cameras are recommended by NextBug, including their similarity scores, the reporter, and the short description of the related bugs.

Similar Bugs Recommendation

956407, [Buri] Camera app temporarily disappeared from device. 2014-01-03 13:43 PST, Marcia Knous [:marcia]	*42% Similarity*
957910, [Camera] Quit camera hurriedly would record no video. 2014-01-08 19:20 PST, Greg Weng [:snowmantw][:gweng]	*40% Similarity*
959464, [Camera] Simplify build time configuration. 2014-01-13 20:20 PST, Diego Marcos [:dmarcos]	*26% Similarity*

Fig. 2: List of similar bugs provided by NextBug.

III. Comparative Study

In this section, we compare a technique for detecting similar bugs using cosine similarity (as implemented by NextBug) with REP [5], a state-of-the-art technique for detecting duplicated bug reports. Our overall aim is to check whether a duplicate bug detection technique is more appropriate for recommending similar bugs than NextBug. We address two research questions:

- **RQ #1:** How does NextBug compare with REP in terms of feedback, precision, likelihood, recall, and F-score?
- **RQ #2:** How does NextBug compare with REP in terms of runtime performance?

A. Data Collection

In this comparative study, we use a dataset of bugs reported for the Mozilla ecosystem. This ecosystem is composed of 69 products including systems such as Firefox[3] and Thunderbird.[4] For the study, we initially considered fixed issues (including both bugs and enhancements) reported from January 2009 to October 2012 (130,495 bugs). We evaluate the goodness of a recommendation based on whether two bugs required similar files to be changed. For Mozilla, we determine the

[3] www.mozilla.org/firefox, verified 2015-11-05.
[4] www.mozilla.org/thunderbird, verified 2015-11-05.

files changed to fix a bug using a heuristic introduced by Walker et al. [6]. Normally, in open source projects, including Mozilla, developers post a patch for a given bug, which must be approved by certified developers. Thus, patches can be considered as proxies for the files effectively changed when working on the bug. By using this heuristic, we linked 65,590 bugs (tasks/issues/enhancements) to patches and after that to the files in the patches.

B. Study Design

We compare NextBug with REP, a technique to detect duplicated bugs that employs a BM25F textual similarity function. REP uses non-textual fields (such as component, version, priority, etc.) along with a two-round stochastic gradient to train a duplicated bug retrieval function. For this comparison, we use the same algorithms and parameters reported in the original work about REP [5]. For example, in the parameter tuning phase, we use 30 training instances, 24 iterations, and an adjustment coefficient of 0.001. The difference is that we use similar bugs as training set, rather than duplicated ones. To compose the training set we randomly selected bugs from our dataset and each bug was paired with two others: a similar bug and a non-similar one. We select similar bugs from an oracle, as described next. We also normalize the results provided by the REP function. Basically, we always divide the results of $REP(d, q)$, where d is a document and q is a query, by the maximal possible result, i.e., $REP(q, q)$. In this way, REP normalized results range from 0 to 1. When evaluating and comparing the techniques, the bugs similar to a query q are the ones with a similarity measure (NextBug) and a normalized REP results greater than a threshold τ. Our implementation of REP and NextBug is publicly available on GitHub.[5]

We simulate NextBug and REP by retrospectively computing recommendations for the bugs in our dataset. Our goal is to reconstruct a scenario where similar bug recommendation lists would be in place when each bug in our dataset was marked as closed. To reconstruct this scenario, we call B_q the set of pending bugs at the exact moment that each bug q was fixed. For each bug $b \in B_q$, the similarity measure (or REP function result) is then used to decide whether b is similar to q or not. A lower bound threshold τ is used in this check. Finally, we call A_q the set of similar bugs that would be recommended for q, $A_q \subseteq B_q$.

After producing the recommendations, we evaluate whether each bug $r \in A_q$ is a useful recommendation by checking if both r and q changed similar files. Suppose that bugs q and r require changes in the sets of files F_q and F_r, respectively. The similarity of F_q and F_r is calculated using the Overlap coefficient [7]:

$$Overlap(F_q, F_r) = \frac{|F_q \cap F_r|}{min(|F_q|, |F_r|)}$$

The Overlap result is equal to one (maximal value) in the situations illustrated in Figure 3. First, when $F_q \subseteq F_r$, i.e., the "context" established by the developer when working on the bug q is reused when working on the recommended bug r. Second, when $Fr \subseteq Fq$, i.e., to work on the recommended bug r the developer does not need to set up new "context" items. In both cases, the developer concludes two bugs and one of the required contexts is completely reused.

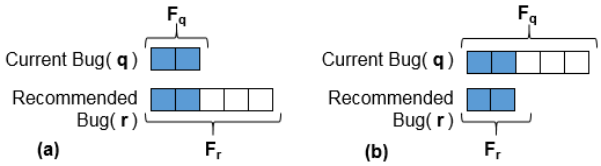


Fig. 3: Maximal overlap scenarios (boxes represent files changed to conclude a bug): (a) the context of the first bug is reused by the recommended bug; (b) no new files are included in the context of the recommended bug.

Overlap similarity is used to create an oracle O_q with the relevant bugs that must be recommended with q, as follows:

$$O_q = \{\ b \in B_q \mid Overlap(F_q, F_b) \geq 0.5\}$$

This oracle includes the pending bugs b at the moment that q was concluded (set B_q) and whose overlap coefficient calculated using the files changed by q and b is greater than 0.5. We are assuming that 50% of "context reuse" between bugs is enough to convince a developer to fix the bugs consecutively (or concurrently).

C. Evaluation Metrics

To answer *RQ #1*, we evaluate the recommendations using four well-known metrics for recommendation systems: feedback, precision, likelihood, and recall [8]. We also used a F-measure that combines both precision and recall into an averaged weighted result.

Feedback: Assuming that Z is the set of queries and that Z_k is the set of queries with at least k recommendations, feedback is the ratio of queries with at least k recommendations:

$$Fb(k) = \frac{|Z_k|}{|Z|}$$

Feedback is a useful metric for recommendation systems, because a recommender that rarely gives recommendations may not be practical. On the other hand, feedback is not the only important point to evaluate, as a recommender that gives too many imprecise recommendations is not trustworthy.

Precision: Precision is the ratio of relevant recommendations provided by the recommender. Assuming that $R_q(k)$ are the top-k recommendations more similar to the query q (measured in terms of the similarity measure or the normalized REP results), precision is defined as follows:

$$P_q(k) = \frac{|R_q(k) \cap O_q|}{|R_q(k)|}$$

[5]https://github.com/hscrocha/NextBug-REP-Comparative-Study-Impl.

The overall precision is the average of the precisions calculated for each query:

$$P(k) = \frac{1}{|Z_k|} \sum_{q \in Z_k} P_q(k)$$

Likelihood: Likelihood checks whether there are useful recommendations among the suggestions, i.e., recommendations included in the proposed oracle. Likelihood provides an indication of how often a query to produce recommendations provides at least one potentially useful result. Likelihood is defined as follows:

$$L_q(k) = \begin{cases} 1 & \text{if } R_q(k) \cap O_q \neq \emptyset \\ 0 & \text{otherwise} \end{cases}$$

Therefore, $L_q(k)$ is a binary measure. If there is at least one useful recommendation among the top-k recommendations, it returns one; otherwise, it returns zero. The overall likelihood is the average of the likelihood computed for each query. We decide to compute likelihood because a recommender might be useful even when it provides one or two incorrect recommendations plus one clearly useful result. In this case, humans with some experience in the domain can rapidly discard the incorrect recommendations and focus on the useful one [8], [9].

Recall: Recall is the ratio of recommendations in the oracle among the provided recommendations. The recall up to k recommendations is defined as follows:

$$Rc_q(k) = \frac{|R_q(k) \cap O_q|}{|O_q|}$$

In our case, recall can be seen as less important than feedback. Indeed, a recommender that gives results for most queries (feedback) is still useful, even if it does not cover all relevant recommendations (recall).

F_1-score: F_1-score is a measure of accuracy that considers both precision and recall. The classical F_1-score (also known as F-measure or F-score) is a balanced weighted average between precision and recall. Since we compute precision and recall for a list of k recommendations, the F_1-score also considers k recommendations as follows:

$$F_1(k) = 2 \times \frac{P(k) \times Rc(k)}{P(k) + Rc(k)}$$

D. Results

RQ #1: Figure 4 shows results of the evaluation metrics, for thresholds ranging from 0.0 to 0.8 (τ parameter). When the threshold is zero, both techniques behave as a ranking function, as the similarity test considers any pending bug.

Figure 4a shows the feedback results. REP provides more recommendations (maximum Feedback of 83%) than NextBug (maximum Feedback of 68%). NextBug feedback reveals how the technique is sensible to the threshold parameter. When we increase the similarity threshold, NextBug feedback decreases because the similarity test becomes more strict. When the threshold is zero, NextBug recommends less bugs than REP due to its component filter, i.e., NextBug only recommends bugs that share the same component as the query. On the other hand, there is a small impact on REP feedback as we increase the threshold. The main reason is that REP similarity function scores are usually high due to the use of many features (e.g., product, version, priority, etc.). Thus, it is very likely that it exists a pending bug sharing at least some features with the query. For example, the recommendations' average similarity for NextBug is 0.17, and for REP is 0.75 (considering a threshold of zero).

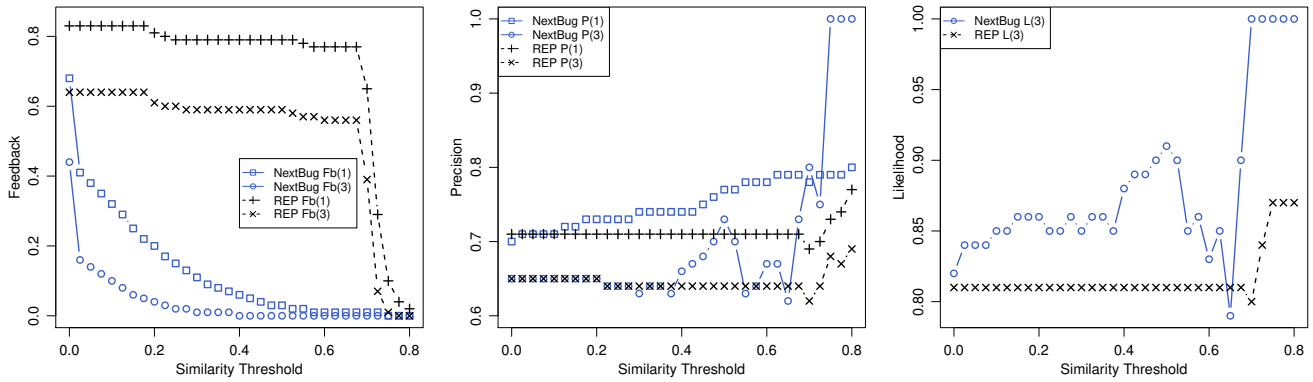
Regarding precision (Figure 4b), both techniques show precision(1) values around 70%, and precision(3) values greater than 62%. One technique may perform slightly better than the other depending on the similarity threshold. For example, if we consider a threshold of zero, then REP has a better precision(1) than NextBug (71% and 70%, respectively). However, for a threshold of 0.2, REP precision(1) is lower than NextBug (71% and 73%, respectively). Moreover, REP precision varies slightly for thresholds less than 0.6 because the feedback values also show a small variation in these cases.

Figure 4c shows that both techniques perform well for likelihood. First, it is worth to mention that likelihood(1) is always equal to precision(1), by definition. For this reason, we restrict the likelihood analysis on a list of three recommendations. NextBug has a minimum likelihood(3) of 79% for a similarity threshold of 0.65, and REP has a minimum likelihood(3) of 80% for a similarity threshold of 0.7.

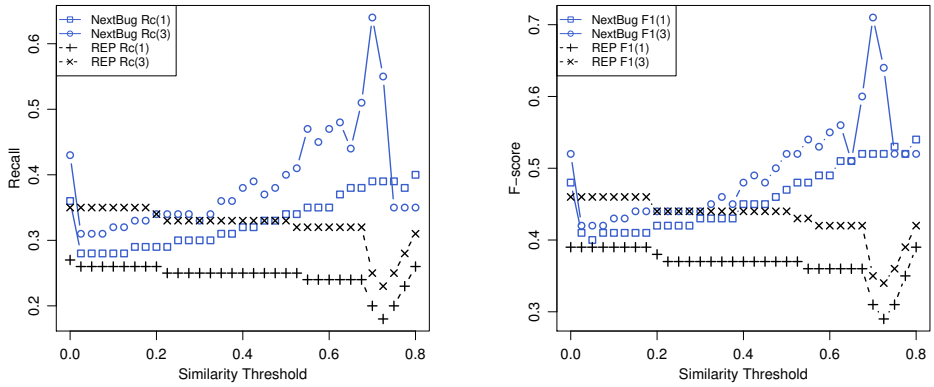
Considering recall, NextBug performs better than REP as shown in Figure 4d. For example, assuming a similarity threshold of zero, the recall(1) values for NextBug and REP are 36% and 27%, respectively. For the same threshold, recall(3) values are 43% for NextBug and 35% for REP. Although recall results may seem low, they are comparable with other recommendation systems. For example, ROSE (a system that recommends classes that usually change together) has an average recall of 33% (in the fine-grained navigation scenario) [8]. Another recommendation system that suggests the most suitable developer to fix a bug has a recall of 10% (highest average recall) [10].

Figure 4e shows the F_1-scores. NextBug shows higher F_1 values considering top-1 recommendations. For example, considering a threshold of zero and 0.4, NextBug $F_1(1)$ scores are 48% and 45%, while REP scores are 39% and 37%, respectively.

Finally, we decide to investigate the relevance of the component filter in NextBug results. As mentioned, NextBug only recommends bugs that share the same component with the query. For this reason, it shows a lower feedback than REP, which does not have this filter. Figure 5a shows the feedback results. As we can see, they are very similar to the ones presented by REP (Figure 4a). For example, for a similarity threshold of zero, feedback(1) is 83% for both NextBug and REP. Although it increases feedback, removing the component filter may have an impact on precision and recall. For example,



(a) Feedback (Fb) (b) Precision (P) (c) Likelihood (L)



(d) Recall (Rc) (e) F-score (F_1)

Fig. 4: Feedback (Fb), Precision (P), Likelihood (L), Recall (Rc), and F-score (F_1) results

Figure 5b shows the F_1 scores for a NextBug instance without component filtering. As we can see, for a similarity threshold of zero, NextBug $F_1(1)$ score decreases from 48% (with filter) to 38% (without filter). However, it is very close to the results provided by REP (39%).

The reason for NextBug (a simple technique) to perform just as well as REP (a more complex function) is because the textual description field of a bug report appears to be the most relevant field to predict similar bugs. As such, the extra fields processed by REP do not contribute significantly to better recommendations although they do contribute to increase feedback.

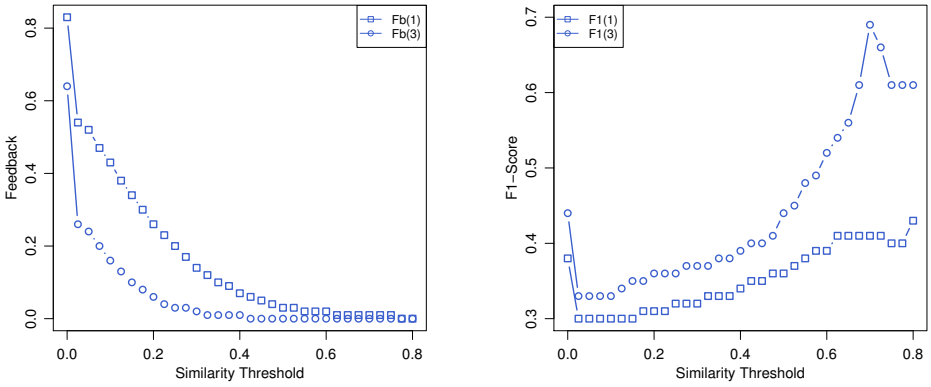
RQ #2: Table I shows the time to compute the recommendations evaluated in this study. The execution time refers to an HP Server, CPU Xeon Six-Core E5-2430 2.20 GHz, 64 GB RAM, operating system Ubuntu 12.04, 64 bits. Both algorithms (NextBug and REP) are implemented in Java and executed in a Java Virtual Machine (JVM) 1.7.0 80. The execution time only considers the information retrieval processing steps, the similarity test (according to each technique), and the time to return the recommendation list. Particularly, we do not consider the time to process the SQL queries that retrieve the pending bugs reported in a given date (since in our study all bugs are initially loaded in main memory, to optimize performance). For REP, the execution time shown in Table I does not include the training stage (or parameter tuning), since this stage can be performed off-line. We execute the experiments three times, using the threshold zero (the one with the highest feedback). We report the average values for the execution time results.

TABLE I: Execution Time (ms)

	Min	Max	Avg	Med	Std Dev
NextBug	0.0	163.6	3.8	3.7	2.1
REP	0.0	1,117.3	11.3	11.4	6.9

The maximum time to provide a recommendation is 163 milliseconds for NextBug and 1.1 seconds for REP. NextBug average, median, and standard deviation time are also lower than REP. In any case, the reported execution times show that is feasible to (re-)compute the recommendations for both techniques each time a developer requests a web page with a bug report.

Summary of findings: For detecting similar bugs (i.e., bugs requiring changes in the same parts of a system), a technique that considers just two bug report fields (component and short description) performs just as well as REP, a technique



(a) Feedback (Fb) (b) F-score (F_1)

Fig. 5: NextBug Feedback (Fb) and F-score (F_1) without the component filter

optimized for detecting duplicate bug reports. Furthermore, both techniques have a runtime performance that supports their online integration with a issue tracker system.

E. Threats to Validity

In this comparative study we evaluated the recommendations provided by two techniques, in a well-defined moment, when a bug is marked as closed. We acknowledge that different results are possible if we consider the production of recommendations in a different moment.[6] However, we evaluate thousands of bugs, reported during almost four years for 69 systems, which reduces an eventual bias due to just considering the production of recommendations in a given moment of a bug's life cycle.

We also assume that the recommendations are useful if they share similar changed files with the query. There are at least two threats to this assumption. First, two bugs are considered similar even when they require changes in different parts of the same source code files. Therefore, it might happen that these parts are sufficiently different to be recommended to the same developer. Second, we consider a recommendation as useful when its file similarity with the query is greater than 0.5, as computed by the Overlap set similarity coefficient. Our selection of 0.5 was a choice to compare in a laboratory two bug recommendation techniques. We acknowledge that different results are possible by varying this threshold. However, instead of presenting results for different Overlap values, we ran the field study we describe in the following section to learn from Mozilla developers if the recommendations produced by NextBug are in fact useful.

IV. Field Study

Experimental studies take an optimistic view; any useful recommendation will be recognized by the recipient of the recommendation. To determine if developers would consider taking a recommendation provided by NextBug, we conducted a field study with Mozilla developers.

A. Study Design

For this study, we monitored bugs handled on Mozilla projects for a week. On each day, we collected data of the bugs fixed in the previous day and computed off-line recommendations of similar bugs, as provided by NextBug. We used $\tau = 0.3$ (similarity threshold) because it showed an interesting balance between precision and recall in the comparative study. Additionally, it does not generate too many recommendations to manage manually.

We analyzed 1,412 bugs handled in this week and we were able to compute recommendations for 421 bugs, which correspond to a feedback of 30% (using the feedback definition presented in Section III-C). For each bug with at least one recommendation, we sent an email, using information from the tracking system, to the developer responsible for its resolution. Figure 6 presents one of the emails we sent. In each email, the developer is presented with a list of up to three bugs and asked whether he/she would consider to work on one of them (Question #1). We also asked the developer why (or why not) they would work on the recommendations (Question #1a and #1b). Finally, we asked the developers whether they would consider it useful to extend the tracker system with a list of similar bugs (Question #2).

We sent only one email to any given developer, regardless of how many tasks he/she concluded in the week we studied. The intention was to avoid a perception of our mails as spam messages. We discarded the usage of a control group due to potential ethical issues. Using a control group would require sending meaningless recommendations to real developers, which could impact negatively their daily work and therefore contribute to a negative image of software engineering researchers among practitioners.

The study involved sending emails to 176 developers. We received 66 answers, which represents a response ratio of 37%. We classify the developers that answered our survey in three

[6]The recommendation list for a given bug may change as one recommended bugs are marked closed. Moreover, it is also possible that additional editions in the data of the pending bugs change the similarity results.

Hello Mr./Ms. [XXX],

The following bug assigned to you was resolved on Tuesday:

[789261] - WebIDL bindings for Window

We found that you might next consider to work on one of the following open bugs:

I. [976307] - ES objects created by WebIDL bindings should be created in the compartment of the callee
II. [986455] - Support implementing part of C++ WebIDL interfaces in JS
III. [979835] - Port BoxObject to WebIDL

Looking at these suggestions, would you:

1. Consider working on one of these suggested bugs?
1a. If so, which of the bugs would you select to work on?
1b. If not, why are these bugs not of interest to work on?

2. Consider useful a Bugzilla extension presenting bugs similar to a browsed one (i.e., bugs that would probably require changes similar to the ones performed when fixing a given bug)?

Fig. 6: E-mail sent to a Mozilla developer

groups: newbies (developers that worked on only one or two bugs in the past), junior (developers that worked on more than two bugs and less than 27 bugs), and experts (developers that worked in more than 27 bugs in the past). To define the thresholds used in this classification we used the first and second quartiles (newbies), third quartile (junior), and fourth quartile (experts) of the distribution of the number of bugs fixed per developer in the dataset used in our comparative study. These groups correspond to 3% (newbies), 8% (junior), and 89% (experts) of the developers we received answers.

B. Results

Table II summarizes the field study results. Both questions were initially proposed to receive a positive (yes) or negative (no) answer. However, in some cases the developers did not answer the questions (blank) or answered in an unclear manner (unclear). The percentage of valid answers (yes or no) is 97% and 79% for Question #1 and Question #2, respectively. In the following subsections, we analyze the valid answers received for each question.

TABLE II: Field study results

	Answer			
	Yes	**No**	**Blank**	**Unclear**
Question #1	39 (59%)	25 (38%)	–	2 (3%)
Question #2	44 (67%)	8 (12%)	10 (15%)	4 (6%)

Question #1 Would you consider working on one of the recommended bugs?

For this first question, 59% (39 out of 66 developers) answered they would indeed consider taking one of the recommendations. Moreover, 86% of the answers (57 out of 66 answers) were provided by expert developers, which increase their confidence. Table III presents detailed information on two recommendations that received a positive feedback from Mozilla developers. We can observe that the cosine similarity between the query and the recommendations is high (greater than 0.33) and the bugs are in fact semantically related to the queries. For example, the first query and associated recommendations denote problems in the library for Firefox marketplace payments. The second query and recommendations are related to the airplane mode feature, from Firefox OS. In fact, the developer who worked on the second query answered that he replied to Rec. #2.1 thanks to our email.

TABLE III: Examples of useful recommendations, according to Mozilla developers (including the textual similarity between the query and the provided recommendations)

	Short Description	**Sim.**
Query #1	fxpay: make a payment with the example app (ID 989136)	
Rec. #1.1	fxpay: save a receipt to device on purchase (ID 991994)	0.35
Rec. #1.2	fxpay: fixup transaction state (ID 987758)	0.34
Rec. #1.3	'Payment Cancelled' message is displayed after 'Payment Complete' message (ID 972108)	0.33
Query #2	Airplane mode icon in status bar is not responsive (ID 1014262)	
Rec. #2.1	Airplane mode icon can co-exist with the wifi icon (ID 1008945)	0.49
Rec. #2.2	Intermittently the user is unable to turn/off Airplane mode on (ID 1003528)	0.48
Rec. #2.3	Airplane mode does not display an 'on' message to user (ID 1010551)	0.39

For the bugs in Table III, we received these comments:

"The suggestions seem pretty accurate (...) the #3 bug is new to me — I didn't know about that one" (Subject #47 on Recs. for Query #1)

"The suggested bugs you present are definitely bugs I'd be likely to pick up after the one I solved" (Subject #40 on Recs. for Query #2)

Table IV presents two examples of incorrect recommendations. In the first example, both the query and the recommendation are related to a specific component, called `Compositor`. However, the query denotes a performance bug and the recommendation denotes a crash. In the second example, Query #4 denotes a fairly simple bug in the Javascript engine and Rec. #4.1 is a more complex but not critical bug, according to the developer who completed the query.

TABLE IV: Examples of incorrect recommendations, according to Mozilla developers (including the textual similarity between the query and the provided recommendations)

	Short Description	**Sim.**
Query #3	Add compositor benchmark (ID 1014042)	
Rec. #3.1	Compositor crash during shutdown crash while debugging (ID 977641)	0.36
Query #4	Use a magic number to identify crashes related to any stack traversal during bailouts (ID 1015145)	
Rec. #4.1	IonMonkey bailouts should forward bailout reason string to bailout handler (ID 1015323)	0.35

We also analyzed the comments related to the incorrect answers, trying to extract small phrases and sentences that contribute to organize the developers' reasons in categories. This analysis resulted in four categories, as follows:

- Recommendations that do not make sense (e.g., *"one bug is a performance bug, the other is a stability bug", Subject #9*).
- Recommendations that might be correct, but they are not a priority at this moment (e.g., *"bugs may interest me but are definitely not my current focus", Subject #1*).
- Recommendations to developers who are not an expert on the component (e.g., *"I fixed that bug because someone broke the build, I'm not interested in the sandbox", Subject #38*).
- Recommendations to paid Mozilla developers, who follow a work schedule (e.g., *"Our manager determines the next bugs we work on", Subject #42*).

The percentage of answers in each category is presented in Table V. Only 20% of the recommendations were in fact ranked as denoting a different bug. In fact, 40% of the negative answers are not exactly because the recommendations are meaningless, but because the developers decided to focus on another bug, which he/she judged with a higher priority.

TABLE V: Reasons for not following a recommendation

Recommendations that do not make sense	20%
Recommendations that might be correct, but are not a priority	40%
Recommendations to non-expert developers	16%
Recommendations to developers with a work schedule	12%
Other reasons	12%

Question #2: Would you consider useful a Bugzilla extension with recommendations?

For this second question, 44 developers (67%) answered that a Bugzilla extension including similar bug recommendations would be useful. All newbies and junior developers answered this second question positively. Among the experts, 82% answered the question positively. We analyzed the comments related to the positive answers which resulted into the following categories:

- The extension is specially good for new contributors (e.g., *"would be immensely helpful for new contributors as they don't know the project very well", Subject #10*).
- The extension would increase developers' productivity (e.g., *"we could do more work in less time", Subject #27*).
- The extension would be useful as a customized bug search engine (e.g., *"might be useful for keeping relevant bugs that I might miss on my radar", Subject #39*).

The percentage of answers in each category is presented in Table VI. We can observe that 11 developers (25%) highlighted the benefits of the recommendations to new Mozilla contributors. Other 14% of the answers mention the increase in productivity and 13% envisioned using NextBug as an advanced bug search engine.

Only 8 developers (12%) answered that a Bugzilla extension including bug recommendations would *not* be useful (all of them are expert developers). We organized the negative answers in two categories:

TABLE VI: Usefulness of a Bugzilla extension

Support new contributors when searching for bugs	25%
Increase productivity	14%
Support to customized bug searches	13%
No reason given	25%
Other reasons	23%

- The extension is not useful for developers who follow a well defined work schedule (e.g., *"in my case a product process is driving the prioritization", Subject #48*).
- The extension is not useful for developers that work on projects with few and usually well-known bugs (e.g., *"I don't manage a lot of open bugs at the same time, so I don't need a list of suggestions", Subject #22*).

Each of these categories received 25% of the answers. Furthermore, 50% of the respondents did not give a clear reason or did not provide an answer at all.

C. Threats to Validity

Differently from the comparative study, we did not test several similarity thresholds, which would be a difficult test in a study with real developers. However, we at least tried to select a threshold showing an interesting balance between likelihood and feedback. We did not send multiple mails to the same developer, to avoid a perception of our messages as spam. Due to this decision, we in fact reduced our sample to 42% of the bugs with recommendations completed in the week monitored in the study. Despite this fact, we were able to receive feedback from 66 unique developers, which is a good response rate (37%). Finally, our study participants might not be representative of the whole population of Mozilla developers and of software developers, in general.

V. Discussion

In this section, we discuss the main benefits of recommending similar bugs (Section V-A), and the open source systems that most benefit of these recommendations (Section V-B).

A. Main Benefits

In the field study, six developers (14%) mentioned that a Bugzilla extension with recommendations would contribute to an increase in productivity. We received answers like this one:

"It would be useful for developers, since they'll be able to be much more productive in their contributions" (Subject #7).

Additionally, similar bugs recommendations can keep contributors working on the system, by indicating new bugs they could work on. In this way, a system like NextBug can contribute to expand the workforce, which is important for the long-term survival of open source systems [1]. For example, 2,221 developers were responsible for the 130,495 bugs we initially considered in the comparative study with Mozilla data. However, 928 developers (42%) worked on at most two

bugs and did not return to work on the system. Therefore, NextBug can help to keep such developers involved with the project. In fact, 11 developers (25%) mentioned this fact like in the following answer:

"A recommendation engine would be immensely helpful for new contributors to the project as they often don't know the project very well and find skulking through the bug tracker tedious and energy-sucking." (Subject #10).

In fact, all newbies and junior developers answered positively to the question about the usefulness of a Bugzilla extension with information on similar bugs.

Similarly, one of the Mozilla's project manager commented that a system like NextBug can be adapted to recommend mentors for pending bugs:

"In the spirit of the old medical-training adage, "see one, do one, teach one", it would be quite valuable to Mozilla to identify community members who have completed a bug who could then get a suggestion based on their previous contributions about mentoring. To be able to say, based on that contribution, you would be a good mentor for these bugs over here, that would be quite valuable to us" (Mozilla's Project Manager).

B. Target Systems

We evaluated NextBug with 69 systems from the Mozilla ecosystem. Open source systems are natural candidates for similar bug recommendations due to the uncoordinated and decentralized nature of their development process, which depends on a large base of contributors, most of them working as volunteers. Some systems, like Mozilla, due include paid developers whose workflows are established by project managers. In our study, these developers usually ranked the recommendations as not useful, as stated in the following comment:

*"I'm a paid contributor to Mozilla, and as such, we have processes used by ourselves and our manager which determine the next bugs we work on. So while your approach looks like it did indeed select bugs somewhat similar to the one you mentioned, the process used to choose which bug to *actually* work on next is quite different than simply 'it is similar'. Your approach may work well for volunteers who have full control over exactly what they choose to do." (Subject #42).*

For open source development that function with a management hierarchy, recommendations of similar bugs may not be useful. Therefore, NextBug should be used in projects where such a management hierarchy and process oversight is not in place. For example, we can mention GNOME and KDE.

VI. Related Work

In large software projects, some bugs are not new but are duplicates of bugs already logged in the bug repository. Optimally, during a triage process, duplicated bugs are filtered out to help ensure developers are able to focus on unique bugs [1], [11], [12]. To automate the work of detecting duplicated bug reports, several approaches have been proposed, typically using text-based information retrieval (IR) techniques to compare bug reports [13]–[15]. Besides textual features, some approaches also consider other bug reports fields (e.g., component, version, priority, etc) [5], execution traces [16], and contextual information [17]. There are also approaches that combine IR-based techniques with topic-based techniques [18]. Finally, some approaches rely on supervised machine learning [5], [19], [20]. In this paper, we assessed a novel application for duplicated bug detectors based on IR techniques. Instead of detecting duplicated bugs during a triage phase, the technique is used to recommend similar (although not exactly identical) bugs to developers.

The benefits of handling software maintenance in large working units is reported in the literature. For example, Banker and Slaughter showed the economies achieved when maintenance is handled in batch mode, i.e., when individual requests are packaged and implemented as part of larger projects [21], [22]. According to their simulation models, this strategy can reduce maintenance costs by 36%. The benefits of periodic maintenance policies—combined with policies to replace a system—is also reported by Tan and Mookerjee [23]. However, both studies are solely based on analytical models. Junio et al. [24] evaluate the advantages on grouping maintenance requests to achieve scale economics. They propose a process, called PASM, that fosters the implementation of maintenance requests in clusters. Marques-Neto et al. [25] made further analysis on the use of the PASM process in a real software organization and evaluated its impact on maintenance services. However, by handling maintenance as projects, a cost of opportunity usually appears, representing the cost of delaying the maintenance work to create larger working units. To avoid this cost, NextBug provides a list of bugs similar to the one browsed by a developer. Therefore, the decision on following or not the recommendations is delegated to the developers.

The negative impacts of context switches on productivity are well-known [26]. After analyzing 10,000 programming sessions of 86 programmers, Parnin and Rugaber observed that only 7% of the sessions do not require navigation to other locations prior to editing [3]. One approach introduced to reduce the cost of context switching is Eclipse Mylyn [2]. As a developer works on an indicated task, Mylyn tracks the parts of artifacts touched and changed, building a degree-of-interest model that describes how important each part of each artifact is to the task. This model can be stored per task and when a task switch occurs, Mylyn can re-display to a developer the parts of artifacts earlier worked on as part of that task.

Cassandra is a task recommender that aims to schedule the maintenance work in order to minimize conflicting changes in parallel software development [27]. The system relies on Mylyn to get contextual data, which is communicated to a centralized scheduler component. This component identifies potential conflicts in order to recommend task orders that restrict dependent tasks or tasks that share common files from

being concurrently edited. Therefore, Cassandra tries to reduce conflict in parallel work whereas the recommender evaluated in this paper attempts to foster a given developer to work on multiple tasks (sequentially our concurrently).

In recent years, there have been many research efforts focused on bug repository information, including assigning severity levels to defect reports [28], distinguishing bugs from enhancements [29], and summarizing bug reports [30]. Weiß et al. propose the use of existing bug reports to automatically estimate effort for new bugs, in terms of persons-hours [31]. To make the predictions, the authors rely on a textual similarity measure to identify those reports that are most closely related. There are also many works on techniques to assign bug reports to suitable developers, which are usually performed by training and employing a classifier or ranking function [10], [32], [33]. These works may also consider fuzzy-sets [34], mining source code and commits artifacts [35], different methods for term weighting and filtering [36], and user activity in the bug tracking system [37]. On the other hand, Wang et al. [38] propose an unsupervised approach employing an activity model and a dynamic cache which measures user activity scores to prioritize developers to assign bugs.

In a previous paper, we reported the design and implementation of NextBug [4]. This paper also includes a first evaluation of the tool. To construct an oracle for this evaluation we consider that two bugs are similar if they are handled by the same developer. However, it is possible that similar bugs are handled by different developers. For the current paper, we extend this preliminary evaluation by considering a more robust oracle. In this oracle, two bugs are similar if they require changes to similar source code files. Moreover, we compare NextBug with a technique originally proposed to detect duplicated bug reports. Finally, we report the results of a field study with Mozilla developers.

VII. Conclusion

When developers work on bugs that touch different parts of a software system, they may spend time between each bug making a context switch. To tackle this problem, NextBug recommender was proposed to identify similar bugs and encourage a developer to work in a more productive manner.

In this paper, we performed a comparative study to verify whether NextBug is more suited for similar bug recommendations than REP (a state-of-the-art technique for duplicate bug retrieval). The recommendations were validated if they modified the same files changed in the bug used as query. Although REP provides more recommendations, NextBug shows comparable results for the remaining metrics. NextBug precision for top-1 recommendation is 70%, and REP is 71%. Considering likelihood for the top-3 recommendations, the values are 82% and 81% for NextBug and REP, respectively. These results indicate that for detecting similar bugs, a technique that considers just the bug components and short descriptions (like NextBug) performs as well as a technique optimized for identifying identical bug reports, which usually consider other fields and compute their recommendations using more complex ranking functions. We also performed a week-long field study by sending e-mails with recommended bugs produced by NextBug to Mozilla developers. We received a feedback of 66 developers (37%) and most of them confirmed that NextBug recommendations indeed express meaningful next bugs for developers to work on.

As future work, we plan to investigate alternatives to improve NextBug results, like building a thesaurus with common words and also with domain-specific words which could contribute to increase NextBug accuracy. We are planning a second field study, when we intend to install NextBug in a real bug tracking system to investigate whether developers really follow similar bug recommendations. We also plan to investigate proactive recommendation styles, e.g., notifying developers when a new bug similar to one he/she is currently working on appear in the tracking platform.

Acknowledgments

Our research is supported by CNPq, CAPES, FAPEMIG, and NSERC. We would like to thank the Mozilla Foundation for providing us with the issues' data from their ecosystem. We also appreciate and thank the Mozilla developers who took their time to answer our survey.

References

[1] 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, Jul. 2002.

[2] M. Kersten and G. C. Murphy, "Using task context to improve programmer productivity," in *14th Symposium on Foundations of Software Engineering (FSE)*, 2006, pp. 1–11.

[3] C. Parnin and S. Rugaber, "Resumption strategies for interrupted programming tasks," *Software Quality Journal*, vol. 19, no. 1, pp. 5–34, 2011.

[4] H. Rocha, G. Oliveira, H. Marques-Neto, and M. Valente, "NextBug: a Bugzilla extension for recommending similar bugs," *Journal of Software Engineering Research and Development (JSERD)*, vol. 3, no. 1, p. 14, 2015.

[5] C. Sun, D. Lo, S.-C. Khoo, and J. Jiang, "Towards more accurate retrieval of duplicate bug reports," in *26th International Conference on Automated Software Engineering (ASE)*, 2011, pp. 253–262.

[6] R. J. Walker, S. Rawal, and J. Sillito, "Do crosscutting concerns cause modularity problems?" in *20th Symposium on the Foundations of Software Engineering (FSE)*, 2012, pp. 1–11.

[7] C. J. V. Rijsbergen, *Information retrieval*, 2nd ed. Butterworth-Heinemann, 1979.

[8] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," in *26th International Conference on Software Engineering (ICSE)*, 2004, pp. 563–572.

[9] G. Shani and A. Gunawardana, "Evaluating recommendation systems," in *Recommender Systems Handbook*, 2011, pp. 257–297.

[10] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in *28th International Conference on Software engineering (ICSE)*, 2006, pp. 361–370.

[11] Y. C. Cavalcanti, P. A. Mota Silveira Neto, D. Lucrédio, T. Vale, E. S. Almeida, and S. R. Lemos Meira, "The bug report duplication problem: an exploratory study," *Software Quality Journal*, vol. 21, no. 1, pp. 39–66, 2013.

[12] J. Xie, M. Zhou, and A. Mockus, "Impact of triage: A study of Mozilla and Gnome," in *Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2013, pp. 247–250.

[13] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of duplicate defect reports using natural language processing," in *29th International Conference on Software Engineering (ICSE)*, 2007, pp. 499–510.

[14] N. Jalbert and W. Weimer, "Automated duplicate detection for bug tracking systems," in *38th International Conference on Dependable Systems and Networks (DSN)*, Jun. 2008, pp. 52–61.

[15] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo, "A discriminative model approach for accurate duplicate bug report retrieval," in *32nd International Conference on Software Engineering (ICSE)*, 2010, pp. 45–54.

[16] 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 *30th International Conference on Software Engineering (ICSE)*, 2008, pp. 461–470.

[17] A. Alipour, A. Hindle, and E. Stroulia, "A contextual approach towards more accurate duplicate bug report detection," in *10th Working Conference on Mining Software Repositories (MSR)*, 2013, pp. 183–192.

[18] A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, D. Lo, and C. Sun, "Duplicate bug report detection with a combination of information retrieval and topic modelling," in *27th International Conference on Automated Software Engineering (ASE)*, 2012, pp. 70–79.

[19] K. Liu, H. B. K. Tan, and M. Chandramohan, "Has this bug been reported?" in *20th International Symposium on the Foundations of Software Engineering (FSE)*, 2012, pp. 28:1–28:4.

[20] Y. Tian, C. Sun, and D. Lo, "Improved duplicate bug report identification," in *16th European Conference on Software Maintenance and Reengineering (CSMR)*, 2012, pp. 385–390.

[21] R. D. Banker and C. F. Kemerer, "Scale economies in new software development," *IEEE Transactions on Software Engineering*, vol. 15, no. 10, pp. 1199–1205, Oct. 1989.

[22] R. D. Banker and S. A. Slaughter, "A field study of scale economies in software maintenance," *Management Science*, vol. 43, pp. 1709–1725, 1997.

[23] Y. Tan and V. Mookerjee, "Comparing uniform and flexible policies for software maintenance and replacement," *IEEE Transactions on Software Engineering*, vol. 31, no. 3, pp. 238–255, 2005.

[24] G. A. Junio, M. N. Malta, H. Mossri, H. T. Marques-Neto, and M. T. Valente, "On the benefits of planning and grouping software maintenance requests," in *15th European Conference on Software Maintenance and Reengineering (CSMR)*, 2011, pp. 55–64.

[25] H. Marques-Neto, G. J. Aparecido, and M. T. Valente, "A quantitative approach for evaluating software maintenance services," in *28th Annual ACM Symposium on Applied Computing (SAC)*, 2013, pp. 1068–1073.

[26] A. N. Meyer, T. Fritz, G. C. Murphy, and T. Zimmermann, "Software developers perceptions of productivity," in *22th International Symposium on Foundations of Software Engineering (FSE)*, 2014, pp. 26–36.

[27] B. K. Kasi and A. Sarma, "Cassandra: proactive conflict minimization through optimized task scheduling," in *35th International Conference on Software Engineering (ICSE)*, 2013, pp. 732–741.

[28] T. Menzies and A. Marcus, "Automated severity assessment of software defect reports," in *24th International Conference on Software Maintenance (ICSM)*, 2008, pp. 346–355.

[29] G. Antoniol, K. Ayari, M. D. Penta, F. Khomh, and Y.-G. Guéhéneuc, "Is it a bug or an enhancement?: a text-based approach to classify change requests," in *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, 2008, p. 23.

[30] R. Lotufo, Z. Malik, and K. Czarnecki, "Modelling the 'hurried' bug report reading process to summarize bug reports," in *28th International Conference on Software Maintenance (ICSM)*, 2012, pp. 430–439.

[31] C. Weiß, R. Premraj, T. Zimmermann, and A. Zeller, "How long will it take to fix this bug?" in *4th Workshop on Mining Software Repositories (MSR)*, 2007, pp. 1–8.

[32] G. A. D. Lucca, M. D. Penta, and S. Gradara, "An approach to classify software maintenance requests," in *18th International Conference on Software Maintenance (ICSM)*, 2002, pp. 93–102.

[33] J. Anvik and G. C. Murphy, "Reducing the effort of bug report triage: recommenders for development-oriented decisions," *ACM Transactions on Software Engineering Methodology (TOSEM)*, vol. 20, no. 3, pp. 10:1–10:35, Aug. 2011.

[34] A. Tamrawi, T. T. Nguyen, J. M. Al-Kofahi, and T. N. Nguyen, "Fuzzy set and cache-based approach for bug triaging," in *13th Conference on Foundations of Software Engineering (FSE)*, 2011, pp. 365–375.

[35] H. Kagdi, M. Gethers, D. Poshyvanyk, and M. Hammad, "Assigning change requests to software developers," *Journal of Software: Evolution and Process*, vol. 24, no. 1, pp. 3–33, 2012.

[36] R. Shokripour, J. Anvik, Z. M. Kasirun, and S. Zamani, "Why so complicated? simple term filtering and weighting for location-based bug report assignment recommendation," in *10th Working Conference on Mining Software Repositories (MSR)*, 2013, pp. 2–11.

[37] H. Naguib, N. Narayan, B. Brugge, and D. Helal, "Bug report assignee recommendation using activity profiles," in *10th IEEE Working Conference on Mining Software Repositories (MSR)*, 2013, pp. 22–30.

[38] S. Wang, W. Zhang, and Q. Wang, "FixerCache: Unsupervised caching active developers for diverse bug triage," in *8th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2014, pp. 25:1–25:10.