The Journal of Systems and Software 143 (2018) 44–58

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Characterizing and predicting blocking bugs in open source projects

Harold Valdivia-Garcia[a,*], Emad Shihab[b], Meiyappan Nagappan[c]

[a] Bloomberg LP, United States
[b] Concordia University, Montreal, Canada
[c] University of Waterloo Waterloo, Ontario, Canada

ARTICLE INFO

Article history:
Received 20 March 2017
Revised 17 February 2018
Accepted 21 March 2018
Available online 6 April 2018

Keywords:
Process metrics
Code metrics
Post-release defects

ABSTRACT

Software engineering researchers have studied specific types of issues such reopened bugs, performance bugs, dormant bugs, etc. However, one special type of severe bugs is *blocking bugs*. *Blocking bugs* are software bugs that prevent other bugs from being fixed. These bugs may increase maintenance costs, reduce overall quality and delay the release of the software systems. In this paper, we study *blocking bugs* in eight open source projects and propose a model to predict them early on. We extract 14 different factors (from the bug repositories) that are made available within 24 hours after the initial submission of the bug reports. Then, we build decision trees to predict whether a bug will be a *blocking bugs* or not. Our results show that our prediction models achieve F-measures of 21%–54%, which is a two-fold improvement over the baseline predictors. We also analyze the fixes of these *blocking bugs* to understand their negative impact. We find that fixing *blocking bugs* requires more lines of code to be touched compared to *non-blocking bugs*. In addition, our file-level analysis shows that files affected by *blocking bugs* are more negatively impacted in terms of cohesion, coupling complexity and size than files affected by *non-blocking bugs*.

1. Introduction

Software systems are becoming an important part of daily life for businesses and society. Most organizations rely on such software systems to manage their day-to-day internal operations, and to deliver services to their customers. This ever growing demand for new and better software products is skyrocketing the software production and maintenance cost. In 2000, Erlikh (2000) reported that approximately 90% of the software life-cycle cost is consumed by software maintenance activities. Two years later, a study conducted by the National Institute of Standards and Technology (NIST) found that software bugs cost $59 billions annually to the US economy (Tassey, 2002).

Therefore, in recent years, researchers and industry have put a large amount of effort in developing tools and prediction models to reduce the impact of software defects (e.g., D'Ambros et al., 2009; Graves et al., 2000; Moser et al., 2008). This work usually leverages data from bug reports in bug tracking systems to build their prediction models. Other work proposed methods for detecting duplicate bug reports (Runeson et al., 2007; Wang et al., 2008; Bettenburg et al., 2008), automatic assignment of bug severity/priority (Sharma et al., 2012; Lamkanfi et al., 2010), predicting fixing time (Marks et al., 2011; Panjer, 2007; Weiss et al., 2007; Giger et al., 2010) and assisting in bug triaging (Anvik et al.,Zou et al., 2011; Anvik and Murphy, 2011). More recently, prior work focused on specific types of issues such as reopened bugs, performance bugs and enhancement requests (Shihab et al., 2013; Zimmermann et al., 2012; Zaman et al., 2012; Antoniol et al., 2008).

In the normal flow of the bug process, someone discovers a bug and creates the respective bug report,[1] then the bug is assigned to a developer who is responsible for fixing it and finally, once it is resolved, another developer verifies the fix and closes the bug report. Sometimes, however, the fixing process is stalled because of the presence of a *blocking bug*. Blocking bugs are software defects that prevent other defects from being fixed. In this scenario, the developers cannot go further fixing their bugs, not because they do not have the skills or resources (e.g., time) needed to do it, but because the components they are fixing depend on other components that have unresolved bugs. These blocking bugs considerably lengthen the overall fixing time of the software bugs and increase the maintenance cost. In fact, we found that blocking bugs can take up 2 times longer to be fixed compared to *non-blocking*

* Corresponding author.
E-mail addresses: hvaldiviagar@bloomberg.net (H. Valdivia-Garcia), eshihab@cse.concordia.ca (E. Shihab), mei.nagappan@uwaterloo.ca (M. Nagappan).

[1] We use the terms "bug" or "bug report" to refer to an issue report (e.g., corrective and non-corrective requests) stored in the bug tracking system.

https://doi.org/10.1016/j.jss.2018.03.053
0164-1212/

bugs. For example, in one of our case studies, the median number of days to resolve a blocking bug is 129, whereas the median for non-blocking bugs is 69 days.

In our earlier work we found that the manual identification of blocking bugs takes 3–18 days on median (Valdivia-Garcia and Shihab, 2014). To reduce such impact, we built prediction models to flag blocking bugs early on for developers. In particular, we mined the bug repositories from six open source projects to extract 14 different factors related to the textual information of the bug, the location the bug is found and the people who reported the bug. Based on these factors and employing a decision tree-based technique (C4.5), we built our prediction models. Then, we compared our proposed models with many other machine learning techniques. In addition, we performed a Top Node analysis (Hassan and Zhang, 2006) in order to determine which factors best identify blocking bugs.

In this paper, we extended the work on blocking bugs in a number of ways. First, to reduce the threat to external validity, we added another 2 projects to our data set. Second, we enhanced our prediction models by using bug report information available within 24 hours after the initial submission of the bug reports. This change has a significant impact on the practical value of our work, since it means that our *new approach* can be applied much earlier than our previously proposed approach. Third, we analyzed the fixes of the blocking bugs to empirically examine their negative impact on the bug-fixing process. In particular, we link the bug-fixes to their corresponding bug-reports. Then, we divide the bug-fixes into blocking/non-blocking bug-fixes in order to compare their size. We also compared the files related to blocking and non-blocking bugs in terms of cohesion, coupling, complexity and lines of code. We note that our examination of the fixes is not done to improve the predictions, nor are we suggesting that fixing information can be used to predict blocking bugs; we study the fixes of blocking bugs to empirically validate their impact. In particular, we would like to answer the following research questions:

RQ1) **What is the impact of blocking bugs?** By analyzing bug reports and bug-fix commits, we find that blocking bugs take up 2 times longer and require 1.2–4.7 times more lines of code to be fixed than non-blocking bugs.

RQ2) **Do files with blocking bugs have higher complexity than files with non-blocking bugs?** We find that files affected by blocking bugs are bigger (in LOC), have higher complexity, higher coupling and less cohesion than not affected by non-blocking bugs.

RQ3) **Can we build highly accurate models to predict whether a new bug will be a blocking bug?** We use 14 different factors extracted from bug databases to build accurate prediction models that predict whether a bug will be a blocking bug or not. Our models achieve F-measure values between 21%–54%. Additionally, we find that the bug description, the comments and the experience of the reporter in identifying previous blocking bugs are the best indicators of whether or not a bug will be blocking bug.

The rest of the paper is organized as follows. Section 2 describes the approach used in this work, including the data collection, preprocessing and a brief description of the machine learning techniques used to predict blocking bugs. Section 3 presents the findings of our case study. We discuss the implications of relaxing the data collection process in Section 4. Section 5 highlights the threats to validity. We discuss the related work in Section 6. Section 7 concludes the paper and discusses future work.

2. Approach

In this section, we first provide a definition of blocking bugs. Second, we present details of the data collection process. We leveraged data from three sources: bug reports, bug-fixing commits and source-code files. Third, we discuss the bug report factors used in our prediction models. Forth, we briefly discuss the machine learning techniques, as well as, the evaluation criteria used to examine the performance of our prediction models.

2.1. Defining blocking and non-blocking bugs

When a user or developer finds a bug in a software system, she/he creates the respective report (bug report) in the bug tracking system. Typically, a bug assigned to a developer who is responsible for fixing it. Once the bug is marked as resolved, another developer verifies the fix and closes the bug report. There are cases in which the fixing of a bug prevents (blocks) other bugs (in the same or related component) from being fixed. We refer to such bugs as **blocking bugs**. Developers of blocked bugs will record the blocking dependency in the *"Blocks"* field of the bug that is blocking them. More precisely, in this work we consider a **blocking bug** as a bug report whose *"Blocks"* field contains at least one reference to another bug. Similarly, we consider a **non-blocking bug** as a bug report whose *"Blocks"* field is empty.

2.2. Data collection

We used the bug report, bug-fix and file history from eight different projects listed in Table 1. We chose these projects because they are mature and long-lived open sources projects, with a large amount of bug reports. Below we explain how we get the bug report and bug-fix data sets from the studied projects.

2.2.1. Bug report collection

We collected bug reports from the bug repository of each project. We only considered those bug reports with status equal to verified or closed. Bug reports closed in less than one day were also filtered out, because we want to analyze non-trivial bug reports. The left-hand side of Table 2 shows a summary of our data set of bug reports. We extracted 857,581 bug reports and discarded 247,781 of them. In brief, after the preprocessing step, we have that: (a) the total number of valid bugs was 609,800, of which 77,448 were blocking bugs and 532,352 were non-blocking bugs; (b) in all projects, the percentages of blocking bugs range from 6%–21% with an overall percentage of 12% and (c) the number of bugs **blocked** by blocking bugs is $\approx$ 57,000 (details in RQ1).

2.2.2. Bug-fix collection

We summarize the extracted bug-fixing commits in the right-hand side of Table 2. We link the bug-reports (in the bug repositories) to their bug-fixing commits (in the code-repositories) using an approach similar to previous studies (Rahman et al., 2012; 2013). First, we checked out the code repositories of each of the projects. The projects studied in this work are comprised of many products and components that use tens or even hundreds code-repositories (e.g., the Fedora website[2] lists 18,000 GIT repositories). However, processing the commits from all of these repositories would be impractical and of little benefit, since many of them have a small number of commits. To select the most representative code-repositories, we use the following two approaches:

- When we were able to identify the products and their code repositories, we manually downloaded the repositories of the

[2] Fedora Git Repositories: http://pkgs.fedoraproject.org/ .

Table 1
Description of the case study projects.

Project	Description
Chromium	Web browser developed by Google and used as the development branch of Google Chrome.
Eclipse	A popular multi-language IDE written in Java, well known for its system of plugins that allows customization of its programming environment.
FreeDesktop	Umbrella project hosting sub projects such as Wayland (display protocol to replace X11), Mesa (free implementation of the OpenGL specification), etc.
Mozilla	Framework and umbrella project that hosts and develops products such as Firefox, Thunderbird, Bugzilla, etc.
NetBeans	Another popular IDE written in Java. Although it is meant for java development, it also provides support for PHP and C/C++ development.
OpenOffice	Office suite initiated by Sun Microsystem and currently developed by Apache.
Gentoo	Operating system distribution built on top of either GNU/Linux or FreeBSD. At the time of writing this paper, Gentoo contains over 17,000 packages.
Fedora	GNU/Linux distribution developed by the Fedora-Project under the sponsorship of Red Hat.

Table 2
Summary of the collected bug reports.

	Bug-report dataset					**Bug-fix dataset**	
Project	# Bugs collected	# Bugs discarded	# Bugs studied	# Blocking bugs	# Non-blocking bugs	# Commits collected	# Commits linked to bugs
Chromium	206,125	149,057	57,068	3,468 [6.1%]	53,600 [93.9%]	223,403	78,472
Eclipse	142,923	13,122	129,801	8,022 [6.2%]	121,779 [93.8%]	422,912	115,119
FreeDesktop	5,844	552	5,292	605 [11.4%]	4,687 [88.6%]	1,002,143	10,773
Mozilla	74,982	6,156	68,826	13,994 [20.3%]	54,832 [79.7%]	214,114	22,210
NetBeans	80,473	3,069	77,404	5,101 [6.6%]	72,303 [93.4%]	210,481	13,720
OpenOffice	87,578	12,639	74,939	4,164 [5.6%]	70,775 [94.4%]	2,038	1137
Gentoo	10,575	3,875	6,700	531 [7.9%]	6,169 [92.1%]	196,561	17,421
Fedora	249,081	59,311	189,770	41,563 [21.9%]	148,207 [78.1%]	114,048	4,493
All Projects	857,581	247,781	609,800	77,448 [12.7%]	532,352 [87.3%]	2,385,700	263,345

20 most buggiest products. For example, the Bugzilla repository of Eclipse lists ≈ 230 different products, out of which we downloaded the code-repositories of the 20 products (84 repositories) with the highest number of bug-reports.
- On the other hand, when we were not able to match the products and the code-repositories, we downloaded all the code-repositories, ranked them by the number of commits and selected the 100 largest repositories. We also tried different number of repositories (50, 100 and 150), however in most of the cases the number of links only slightly improved (less than 1%) after 100 repositories.

In total, we downloaded more than 400 repositories. We refer the reader to our online appendix Valdivia-Garcia (2018) for a detailed list of the code-repositories used in this study. Once we obtained all the commits, we extracted those commits that contain bug-related words (e.g., bug, fixed, failed, etc) and potential bugs identifiers (e.g., bug#700, rhbz:800, etc) in their commit messages. To validate the collected commits, we checked that the bug-identifiers in the commits are present in our bug report data set. In total, we extracted ≈ 2.4 million commits, out of which approximately 263,345 commits were successfully linked to one or more bug-reports in our data set. Of these linked commits, 61,052 (23%) were commits fixing blocking bugs and about 202,293 (77%) were commits fixing non-blocking bugs.

2.2.3. Code-metrics collection

We used UNDERSTAND from Scitools[3] to extract four metrics from the source-code files in the code repositories: Lack of Cohesion, Coupling Between Objects, Cyclomatic Complexity and LOC. In our analysis, we take into account Java, C, C++, Python, Javascript, PHP, Bash and Patch source code files.

From the bug-fixing commits obtained in the previous section, we identified 402,423 buggy files. Then, we analyzed the distribution of the number of bugs per file and we found that ≈ 90% of the buggy files have at most 5 bugs and usually just 1 bug on median. Therefore, in this work, we split the buggy files into two groups: (a) files affected by at least one blocking bug (**blocking files** for brevity) and (b) files affected only by non-blocking bugs (**non-blocking files** for brevity). Table 3 shows the distribution of the blocking files and non-blocking files across all of the projects. We can see that 39% of the files are blocking files (156,507 files), whereas 61% are non-blocking files (245,916 files).

Table 3
Distribution of the number of blocking and non-blocking files.

Project	# Blocking Files	# Non-blocking Files	# Buggy Files
Chromium	34,430 [36%]	60,282 [64%]	94,712
Eclipse	74,580 [43%]	97,375 [57%]	171,955
FreeDesktop	1,074 [22%]	3,774 [78%]	4,848
Mozilla	34,939 [78%]	9,612 [22%]	44,551
NetBeans	3,876 [19%]	16,833 [81%]	20,709
OpenOffice	1,752 [4%]	48,183 [96%]	49,935
Gentoo	4,182 [33%]	8,510 [67%]	12,692
Fedora	1,674 [55%]	1,347 [45%]	3,021
All	156,507 [39%]	245,916 [61%]	402,423

To better understand the files affected by blocking and non-blocking bugs, we analyzed the distribution of their programming languages. In Table 4, we show the percentage of blocking files (third column) and non-blocking files (fourth column) across the top programming languages in each of the projects. For example, in Fedora 49% of the blocking files and 19% of the non-blocking files are written in Bash. Additionally, from the fifth column, we can observe that about 98% of the buggy files in Fedora are Patch or Bash files. As we will discuss in RQ2, this situation will prevent us from extracting two of the four code metrics for Fedora.

2.3. Factors used to predict blocking bugs

Since our goal is to be able to predict blocking bugs, we extracted different factors from the bug reports so the blocking bugs can be detected early on. In addition, we would like to determine which factors best identify these blocking bugs. We consider 14 different factors to help us discriminate between blocking and non-blocking bugs. To come up with a list of factors, we sur-

[3] http://www.scitools.com .

Table 4
Distribution of source code files across different programming languages. In each of columns three to five, we report the percentage of files that belong to a particular programming language.

Project	Language	% Blocking Files	% Non-blocking Files	% Buggy Files
Chromium	C++	86%	77%	81%
	JS	6%	10%	8%
	C	4%	5%	5%
	Others	4%	8%	6%
Eclipse	Java	99%	99%	99%
	Others	1%	1%	1%
FreeDesktop	C	88%	84%	84%
	C++	12%	14%	14%
	Others	0%	2%	2%
Mozilla	C++	40%	32%	39%
	JS	28%	48%	31%
	C	26%	13%	24%
	Others	6%	7%	6%
NetBeans	Java	100%	97%	98%
	Others	0%	3%	2%
OpenOffice	C++	97%	81%	82%
	Java	2%	17%	16%
	Others	1%	2%	2%
Gentoo	Python	68%	4%	31%
	C	4%	42%	26%
	Bash	14%	28%	22%
	Patch	7%	13%	10%
	Others	7%	13%	11%
Fedora	Patch	49%	79%	60%
	Bash	49%	19%	38%
	Others	2%	2%	2%

veyed prior work. For example, Sun et al. (2011) included factors such product, component, priority, etc in their models to detect duplicate bugs. Lamkanfi et al. (2010, 2011) used textual information to predict bug severities. Wang et al. (2008) and Jalbert and Weimer (2008) used text mining to identify duplicate bug reports. Zimmermann et al. (2012) showed that the reporter's reputation is negatively correlated with reopened bugs in Windows Vista. Furthermore, many of our factors are inspired in the metrics used by our prior work (Shihab et al., 2013), predicting reopened bugs. We list each factor and provide a brief description for each below:

1. **Product:** The product where the bug was found (e.g., Firefox OS, Bugzilla, etc). Some products are older or more complex than others and therefore, are more likely to have blocking bugs. For example, Firefox OS and Bugzilla are two Mozilla products with approximately the same number of bugs ($\approx$880), however there were more blocking bugs in Firefox OS (250 bugs) than in Mozilla (30 bugs).
2. **Component:** The component in which the bug was found (e.g., Core, Editor, UI, etc). Some components are more/less critical than others and as a consequence more/less likely to have blocking bugs than others. For example, it might be the case that bugs in critical components prevent bugs in other components from being fixed. Note that we were not able to have this factor for Chromium because its issue tracking system does not support it.
3. **Platform:** The operating system in which the bug was found (e.g., Windows, Android, GNU/Linux etc). Some platforms are more/less prone to have bugs than others. It is more/less likely to find blocking/non-blocking bugs for specific platforms.
4. **Severity:** The severity describes the impact of the bug. We anticipate that bugs with a high severity tend to block the development and debugging process. On the other hand, bugs with a low severity are related to minor issues or enhancement requests.
5. **Priority:** Refers to the order in which a bug should be attended with respect to other bugs. For example, bugs with low priority values (i.e., P1) should be prioritized instead of bugs with high priority values (i.e., P5). It might be the case that a high/low priority is indicative of a blocking/non-blocking bugs.
6. **Number in the CC list:** The number of developers in the CC list of the bug. We think that bugs followed by a large number of developers might indicate bottlenecks in the maintenance process and therefore are more likely to be blocking bugs.
7. **Description size:** The number of words in the description. It might be the case that long/short descriptions can help to discriminate between blocking and non-blocking bugs.
8. **Description text:** Textual content that summarize the bug report. We think that some words in the description might be good indicators of blocking bugs.
9. **Comment size:** The number of words of all comments of a bug. Longer comments might be indicative of bugs that get discussed heavily since they are more difficult to fix. Therefore, they are more likely to be blocking bugs.
10. **Comment text:** The comments posted by the developers during the life cycle of a bug. We think that some words in the comments might be good indicators of blocking bugs.
11. **Priority has increased:** Indicates whether the priority of a bug has increased after the initial report. Increasing priorities of bugs might indicate increased complexity and can make a bug more likely to be a blocking bug. Note that we were unable to obtain this information for Chromium.
12. **Reporter name:** Name of the developer or user that files the bug. We include this factor to investigate whether bugs filed by a specific reporter are more/less likely to be blocking bugs.
13. **Reporter experience:** Counts the number of previous bug reports filed by the reporter. We conjecture that more/less experienced reporters may be more/less likely to report blocking bugs.
14. **Reporter blocking experience:** Measures the experience of the reporter in identifying blocking bugs. It counts the number of blocking bugs filed by the reporter previous to this bug.

In order to extract information for the factors, we first obtained the closing-dates and blocking-dates of the bug-reports. **Closing-date** refers to the latest date in which a bug was closed. To obtain this information, we inspect the history of the bugs looking for the date of the last appearance of the tag *"status"* with a value equal to "closed". **Blocking-date** refers to the earliest date in which a bug was marked as blocking bug. To calculate this information, we look for the date of the first appearance of the tag *"Blocks"* in the history of the bugs.

For the non-blocking bugs, we extracted the last values of the factors prior to their *closing-dates and within 24 hours after the submission*. On the other hand, for the blocking bugs, we extracted the last values of the factors prior to their *blocking-dates and within 24 hours after the submission*. The rationale for this approach is that, although the data after the blocking-date is useful information about the fixing process in general, it is not useful to identify a blocking bug because we already know that the bug is a blocking bug (i.e., by then no prediction is needed). Since our aim is to identify potential blocking bugs early on, then we can only rely on data before the blocking phenomenon happens. That way we can shorten the overall fixing-time.

As we mentioned above, these 14 factors have been used in prior studies and most of them are easy to extract through software repositories. Because our goal is to help developers to identify blocking bugs *early on*, we only use bug report information available within 24 hours after the initial submission of the bug reports. When a factor was empty, we set its value to *NA* (or *zero* for numeric factors). That said, it is important to note that 3 of our factors (product, component and reporter's name) are project-specific. Therefore, if a practitioner would like to predict block-

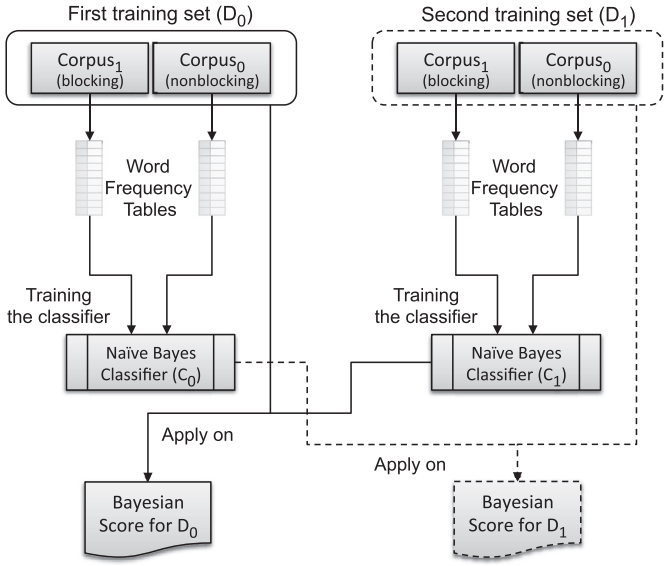

Fig. 1. Converting textual factor into Bayesian-score.

ing bugs in a cross-project setting, she/he might not able to reuse models on new projects. In that situation, the simpler approach would be to remove the project-specific factors from the model or adapt these factors from their specific project in order to have a more flexible model.

Finally, another important observation to note is that the description text and the comment text factors need special treatment before being included in our prediction models. We describe this special preprocessing in detail in the next sub-section.

2.4. Textual factor preprocessing

The description and comments in bug reports are two rich sources of unstructured information that require special preprocessing. These factors contain discussions about the bugs and can also provide snapshots of the progress and status of such bugs. One way to deal with text based factors is using a vector representation. In this kind of representation, a new factor is created for each unique word in the data set. Similar to prior work (Ibrahim et al., 2010; Shihab et al., 2013), we followed this simple approach. In Fig. 1, we show our adapted approach to convert textual factors into numerical values. We used a Naive–Bayes classifier to calculate the Bayesian-score of these two factors. Basically this metric indicates the likelihood that a description or comment belongs to certain kind of bug (i.e., blocking or non-blocking).

We divide the entire data set into two training sets (D_0 and D_1) using stratified random sampling. This ensures that we have the same number of blocking and non-blocking bugs in both training sets. We train a classifier (C_0) with the first training set and use it to obtain the Bayesian-scores on the second training set. We also do the same in the opposite direction. We build a classifier (C_1) using the second training set and apply it on the first training set. This strategy is used in order to avoid the classifiers from being biased toward their training sets; otherwise, it will lead to optimistic (unrealistic) values for the Bayesian-scores.

In our classifier implementation, each training set is split into two corpora ($corpus_1$ and $corpus_0$). The first corpus contains the descriptions/comments of the blocking bugs. The second corpus contains the description/comments of the non-blocking bugs. We create a word frequency table for each corpus. The textual content is tokenized in order to calculate the occurrence of each word within a corpus. Based on these two frequency tables, the next step is to calculate the probabilities of all the words to be in $corpus_1$ (i.e., blocking bugs), because we are interested in identifying these kinds of bugs. The probability is calculated as follow: if a word is in $corpus_1$ and not in $corpus_0$, then its probability is close to 1. If a word is not in $corpus_1$ but in $corpus_0$, then its probability is close to 0. On the other hand, if the word is in both corpora, then its probability is given by $p(w) = \frac{\%w \text{ in } corpus_1}{\%w \text{ in } corpus_1 + \%w \text{ in } corpus_0}$.

Once the classifiers are trained, we can obtain the Bayesian-score of a text based factor by mapping its words to their probabilities and combining them. The formula for the Bayesian-score is $p(text) = \frac{\prod p(w_i)}{\prod p(w_i) + \prod (1 - p(w_i))}$. For this calculation, the fifteen most relevant words are considered (Graham, 2003). Here, *"relevant"* means those words with probability close to 1 or 0.

2.5. Prediction models

For each of our case study projects, we use our proposed factors to train a decision tree model to predict whether a bug will be a blocking bug or not. We also compare our prediction model with four other classifiers namely: Naive Bayes, kNN, Zero-R, Logistic Regression, Random Forests and Stacked Generalization.

2.5.1. Decision tree model

We use a tree-based learning algorithm to perform our predictions. One of the benefits of decision trees is that they provide explainable models. Such models intuitively show to the users (i.e., developers or managers) the decisions taken during the prediction process. The C4.5 algorithm (Quinlan, 1993) belongs to this type of data mining technique and like other tree-based classifiers, it follows a greedy divide and conquer strategy in the training stage. The algorithm recursively splits data into subsets with rules that maximize the information gain. The rules are of the form $X_i < b$ if the feature is numeric or into multiple subsets if the feature is nominal. In Fig. 2, we provide an example of a tree generated from the extracted factors in our data set. The sample tree indicates that a bug report will be predicted as blocking bug if the Bayesian-score of its comment is > 0.74, there are more than 6 developers in the CC list and the number of words in the comments is greater than 20. On the other hand, if the Bayesian-score of its comment is ≤ 0.74 and the reporter's experience is less than 5, then it will be predicted as a non-blocking bug.

2.5.2. Naive–Bayes model

We use this machine learning method for two purposes: to convert textual information into numerical values (i.e., to obtain the probability that a description/comment belongs to a blocking-bug), and to build a prediction model and compare its performance with that of our decision tree model. This simple model is based on the Bayes theorem and the assumption that the factors are randomly independent. For a given record x, the model predicts the class k that maximizes the conditional joint distribution of the data set. Mathematically, the model can be written as:

$$f(x) = \arg\max_k \frac{P(C = k) \prod_i P(x_i | C = k)}{P(X = x)}$$

Here, the prior-probability $P(C = k)$ can be estimated with the percentage of training records labeled as k (e.g., percentage of blocking or non-blocking). The conditional probabilities $P(x_i | C = k)$ can be estimated with $\frac{N_{k,i}}{N_k}$, where the numerator is the number of records labeled as k for which the ith-factor is equal to x_i and the denominator is the number of records labeled as k. The probability $P(X = x)$ can be neglected because it is constant with respect to the classes.

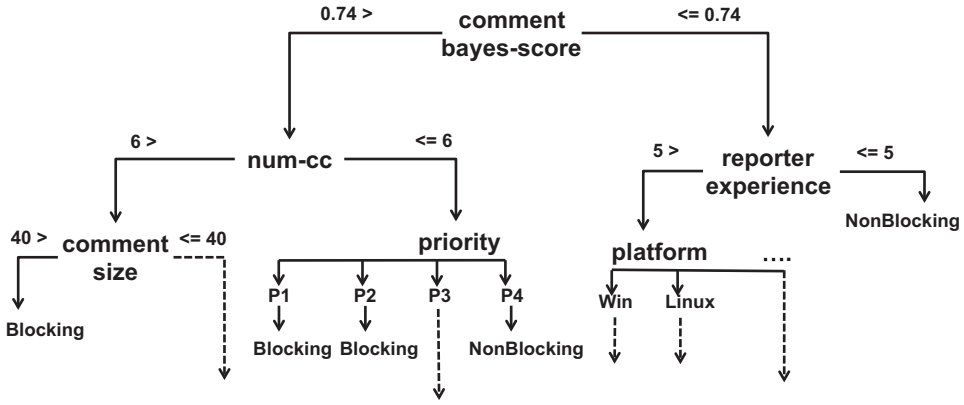

Fig. 2. Example of a Decision Tree.

2.5.3. k-nearest neighbor model

The k-nearest neighbor model is a simple, yet powerful memory-based technique, which has been used with relative success in previous bug prediction works (Weiss et al., 2007; Lamkanfi et al., 2011). The idea of the method is as follows: given an unseen record $\hat{x}$ (e.g., an incoming bug report), we calculate the distance of all records x in the training set (e.g., already-reported bugs) to $\hat{x}$, then we select the k closest instances and finally classify $\hat{x}$ to the most frequent class among these k neighbors. In this work, we considered $k = 5$ as the number of neighbors, used the euclidean metric for numerical factors and the overlap metric for nominal factors. Under the overlap metric, the distance is zero if the values of the factors are equal and one otherwise.

2.5.4. Zero-R model

Zero-R (no rule) is the simplest prediction model because it always predicts the majority class in the training set. We use this classifier as one of our baseline models in the comparison section.

2.5.5. Logistic regression

Logistic regression is statistical binary classification model extensively used in the literature on software bug prediction (Khoshgoftaar and Seliya, 2004; Antoniol et al., 2008; Zimmermann et al., 2012). For a given record $\mathbf{x} = x_1, x_2, \cdots, x_p$, this prediction model estimates the probability that such a record belongs to the class $k = 1$ (e.g., blocking-bug) using the following equation:

$$P(k = 1|\mathbf{x}) = \frac{e^{\beta_0+\beta_1 x_1+\cdots+\beta_p x_p}}{1 + e^{\beta_0+\beta_1 x_1+\cdots+\beta_p x_p}}$$

where the regression coefficients β_i are found during the training phase. For a detailed description of the logistic regression model, we refer readers to Basili et al. (1996).

2.5.6. Random forests model

Random Forests (Breiman, 2001) is an ensemble classification approach that makes its prediction based on the majority vote of a set of weak decision trees. This approach reduces the variance of the individual trees and makes the model more resilient to noise in the data set. In general, the random forests model outperforms simple decision trees in terms of prediction accuracy (Caruana and Niculescu-Mizil, 2006).

2.5.7. Stacked generalization

Stacked Generalization (Wolpert, 1992) is an ensemble classification approach, which attempts to increase the performance of individual machine learning methods by combining their outputs (i.e., individual predictions) using another machine learning method referred to as the meta-learner. In this work, we use C4.5, Naive–Bayes and kNN algorithm as our individual models, and Logistic regression as the meta-learner.

2.6. Performance evaluation

A common metric used to measure the effectiveness of a prediction model is its accuracy (fraction of correctly classified records). However, this metric might not be appropriate when the data set is extremely skewed towards one of the classes (Monard and Batista, 2002). If a classifier tends to maximize the accuracy, then it can perform very well by simply ignoring the minority class (Hulse et al., 2007; Weiss, 2004). Since our data set suffers from the class imbalance problem, the accuracy is not enough and therefore we include three other performance measures: precision, recall and f-measure. These measures are widely used to evaluate the quality of models trained on imbalanced data.

1. **Precision:** The ratio of correctly classified blocking bugs over all the bugs classified as blocking.
2. **Recall:** The ratio of correctly classified blocking bugs over all of the actually blocking bugs.
3. **F-measure:** Measures the weighted harmonic mean of the precision and recall. It is calculated as $F\text{-}measure = \frac{2*\text{Precision}*\text{Recall}}{\text{Precision}+\text{Recall}}$.
4. **Accuracy:** The ratio between the number of correctly classified bugs (both the blocking and the non-blocking) over the total number of bugs.

A precision value of 100% would indicate that every bug we classified as blocking bug was actually a blocking bug. A recall value of 100% would indicate that every actual blocking bug was classified as blocking bug.

We use stratified 10-fold cross-validation (Efron, 1983) to estimate the accuracy of our models. This validation method splits the data set into 10 parts of the same size preserving the original distribution of the classes. At the *i-th* iteration (i.e., fold), it creates a testing set with the *i-th* part and a training set with the remaining 9 parts. Then, it builds a decision tree using the training set and calculate its accuracy with the testing set. We report the average performance of the 10 folds. Since our data sets have a low number of blocking bugs, the stratified sampling prevents us from having parts without blocking bugs. Additionally, we use re-sampling on the *training data only* in order to reduce the impact of the class imbalance problem (i.e., the fact that there are many non-blocking bugs and very few blocking bugs) of our data sets.

3. Case study

This section reports the results of our study on eight open source projects and answers our three research questions. First, we

Table 5
Median fixing time in days and the result of the Wilcoxon rank-sum test for blocking and non-blocking bugs.

Project	t_{block}	$t_{nonblock}$ (X)
Chromium ***	35 [1.3X]	28
Eclipse ***	129 [1.9X]	69
FreeDesktop ***	67 [1.6X]	43
Mozilla ***	75 [1.4X]	52
NetBeans ***	204 [1.4X]	149
OpenOffice ***	129 [1.1X]	113
Gentoo ***	80 [1.6X]	52
Fedora ***	119 [1.1X]	107

(***)$p < 0.001$.

characterized the impact of blocking bugs in terms of their fixing time, blocking dependency and bug-fixing commits (i.e., bug-fix size). Second, we inspected the files affected by blocking and non-blocking bugs and measure their complexity to better understand the blocking phenomenon. Third, we built different prediction models to detect whether a bug will be or not a blocking bug and performed Top Node analysis to determine which of the collected factors are good indicators to identify blocking bugs.

RQ1. What is the impact of blocking bugs?

Motivation. Since blocking bugs delay the repair of other bugs (i.e., *blocked bugs*), they are harmful for the maintenance process. For example, if blocking bugs take longer than other ordinary bugs, then the overall fixing time of the system might increase. Similarly, the presence of blocking bugs that block a large number of other bugs (high dependency) might become bottlenecks for maintenance, and impact the quality of the system. Although there are different ways to define the impact of software bugs on software projects, in this RQ we are interested in quantifying the effects caused by blocking bugs during bug triaging. Therefore, in this RQ, we define the *impact* in terms of two proxy metrics collected at bug-report level, namely fixing-time and degree of dependency

Approach. First, we calculate the fixing time for both blocking and non-blocking bugs as the time period between the date when a bug is reported until its closing date. Then, we performed an unpaired Wilcoxon rank-sum test (also called Mann–Whitney U test) for the alternative hypothesis H_a: $t_{block} > t_{nonblock}$, in order to determine whether blocking bugs take longer to be fixed compared to non-blocking bugs. On the other hand, we analyze the degree of blocking dependency as the number of bugs that depend on the same blocking bugs.

Results. Fixing time. Table 5 reports the median fixing-time for *blocking/non-blocking* bugs. For all of the projects, we observe that the fixing-time for *blocking* bugs is 1.1–1.9 times longer than for the *non-blocking* bugs. In addition, the results of the Wilcoxon test confirm that there is a statistically significant difference between the blocking and non-blocking bugs for all of the projects ($p\text{-value} < 0.001$), meaning that the fixing-time for blocking bugs is statistically significantly longer than the fixing-time for non-blocking bugs.

Dependency of blocking bugs. In our study, we found that blocking bugs represent 12% of all bugs in our data set (77,448 bugs). In order to assess the impact of the dependency of these blocking bugs, we extracted the list of blocked bugs contained in the *"Blocks"* field of each blocking bug. In total, we identified 57,015 different bug reports that were blocked by blocking bugs. At the time of the data collection, many of these blocked bugs were still in progress (and therefore were not included in our data set). Hence, we cannot claim that they account for about 9% of our data set. Table 6 reports the distribution of the degree of dependency between one and six. Furthermore, we include a category "≥ 7" for those blocking bugs that block seven or more bugs. At first sight, it is easy to see that approximately 89–98% of the blocking bugs for all projects only block 1 or 2 bugs. As a consequence, blocking bugs with high dependency are uncommon. To better understand the severity of these bugs with degree of dependency greater than or equal to 7, we performed a manual inspection, and we found inconclusive results. For example, in the Eclipse project, many bugs with high dependency were actual enhancements with low priority (e.g., P3 or P4) instead of real defects. On the other hand, in NetBeans, we found that indeed these blocking bugs were real defects with high priority (e.g., P1 or P2).

Discussion Although, we found that blocking bugs take longer to be fixed compared to non-blocking bugs, the evidence is still unclear whether or not blocking bugs are more complex to fix. Blocking bugs may be easy to fix, but take a long time to find the right developers to solve them, or many blocking bugs are actually enhancements that while desirable, are not a priority, so the developers postpone them in favor of more important bugs. Therefore, we analyze the size of bug-fixes, to determine whether blocking bugs require more effort to fix than non-blocking bugs. First, we calculate the bug-fix size as the number of lines modified (LM) from all the commits related to the bug. Then, we check whether blocking bug-fixes are larger than the non-blocking bug-fixes by using a Wilcoxon rank-sum test for the hypothesis H_a: $LM_{block} > LM_{nonblock}$.

In Table 7, we report the median bug-fix size (code-churn) of blocking and non-blocking bugs. We can observe that for all of the projects, blocking bug-fixes are 1.2 – 4.7 times larger than non-blocking bug-fixes. The result of the Wilcoxon rank-sum test verify that blocking bugs have statistically significantly larger bug-fixes than non-blocking bugs.

The time to address a blocking bug is 1.1–1.9 times longer than the time it takes to address a non-blocking bug. Simultaneously, fixing blocking bugs requires 1.2 – 4.7 times more lines of code to be modified than fixing non-blocking bugs.

RQ2. Do files with blocking bugs have higher complexity than files with non-blocking bugs?

Motivation We found that fixing blocking bugs require more effort and time (RQ1). However, it is not clear whether files with blocking bugs (**blocking files**) are different from files with non-blocking bugs (**non-blocking files**). In this RQ, we would like to analyze and quantify the blocking phenomenon at file level.

Approach

To answer this question, first we extract four metrics from the source-code files in the code-repositories: size (LOC), Cyclomatic Complexity (CC), Lack of Cohesion (LCOM) and Coupling Between Objects (CBO).

Results lack of cohesion. Table 8 reports the median of LCOM for blocking/non-blocking files. We see that blocking files have slightly higher LCOM (1.02–1.18 times higher) than non-blocking files. We compared these two groups of files using a Wilcoxon rank-sum test in order to determine if the difference is statistically significant. For

Table 6
Degree of blocking dependency.

Degree	Chromium	Eclipse	FreeDesktop	Mozilla	NetBeans	OpenOffice	Gentoo	Fedora
1	74.1%	86.3%	87.4%	69.7%	89.9%	90.8%	85.7%	78.5%
2	20.1%	9.5%	10.4%	18.9%	7.8%	6.9%	9.4%	15.1%
3	3.5%	2.3%	2.0%	5.7%	1.3%	1.3%	2.8%	3.9%
4	1.4%	0.8%	0.2%	2.4%	0.5%	0.6%	1.3%	1.3%
5	0.3%	0.3%	0.0%	1.2%	0.2%	0.1%	0.2%	0.5%
6	0.3%	0.2%	0.0%	0.8%	0.1%	0.1%	0.0%	0.2%
≥ 7	0.2%	0.6%	0.0%	1.3%	0.2%	0.1%	0.6%	0.5%

Table 7
Median bug-fix size and the result of the Wilcoxon rank-sum test for blocking and non-blocking bugs.

Project	LM_{block}	$LM_{nonblock}$ (X)
Chromium ***	205 [4.7X]	44
Eclipse ***	107 [3.3X]	32
FreeDesktop ***	25 [1.2X]	20
Mozilla ***	66 [2.4X]	28
NetBeans ***	52 [2.7X]	19
OpenOffice **	77 [2.1X]	38
Gentoo ***	52 [2.6X]	20
Fedora ***	84 [1.4X]	58

(***) $p < 0.001$; (**) $p < 0.01$.

Table 8
Median lack of cohesion for blocking and non-blocking files.

Project	$LCOM_{block}$	$LCOM_{nonblock}$ (X)
Chromium ***	67% [1.10X]	61%
Eclipse ***	58% [1.16X]	50%
FreeDesktop	71% [0.86X]	83%
Mozilla	87% [1.02X]	85%
NetBeans ***	79% [1.03X]	77%
OpenOffice ***	71% [1.18X]	60%
Gentoo	–	–
Fedora	–	–

(***) $p < 0.001$; (**) $p < 0.01$; (*) $p < 0.05$.

Table 9
Median coupling for the blocking and non-blocking files.

Project	CBO_{block}	$CBO_{nonblock}$ (X)
Chromium ***	10 [1.43X]	7
Eclipse ***	11 [1.22X]	9
FreeDesktop	12 [1.26X]	9.5
Mozilla	8 [1.14X]	7
NetBeans ***	23 [1.15X]	20
OpenOffice ***	11 [1.38X]	8
Gentoo	–	–
Fedora	–	–

(***) $p < 0.001$; (**) $p < 0.01$; (*) $p < 0.05$.

Table 10
Median cyclomatic complexity for blocking and non-blocking files.

Project	CC_{block}	$CC_{nonblock}$ (X)
Chromium ***	9 [1.8X]	5
Eclipse ***	12 [1.5X]	8
FreeDesktop ***	58 [1.6X]	37
Mozilla ***	11 [1.2X]	9
NetBeans ***	32 [1.3X]	24
OpenOffice ***	53 [7.6X]	7
Gentoo	18 [0.5X]	36.5
Fedora	–	–

(***) $p < 0.001$; (**) $p < 0.01$; (*) $p < 0.05$.

four projects (Chromium, Eclipse, Netbeans and OpenOffice), we find that files with blocking bugs have statistically less cohesion than files with non-blocking bugs. For FreeDesktop and Mozilla, we find no evidence that blocking files have higher LCOM than non-blocking files. Although these projects have a relative large number of buggy files, the UNDERSTAND tool was able to extract the LCOM metric from only a small fraction of the buggy files. For both FreeDesktop and Mozilla, we obtained the LCOM metric from 7% and 25% of the buggy files respectively. In contrast, we obtained the LCOM metric from about 44%–93% of buggy files for the other projects. This is not surprising since, most of the buggy files in FreeDesktop and Mozilla are written in C and Javascript. From Table 4, we can see that for FreeDesktop, about 84% of the buggy files are written in C, whereas for Mozilla about 55% of the buggy files are written in C and Javascript.

Coupling between objects. In Table 9, we show the median of CBO for blocking/non-blocking files. For four projects (Chromium, Eclipse, Netbeans and OpenOffice), we find that blocking files are coupled to other classes $1.15 - 1.43$ times more than non-blocking files. The result of the Wilcoxon rank-sum test shows that, there is a statistically significant difference in terms of CBO between blocking and non-blocking files. Similar to the previous metric, we find no evidence that blocking files have higher CBO than non-blocking files for FreeDesktop and Mozilla.

Cyclomatic complexity. Prior work showed that OO metrics such as LCOM and CBO are significantly associated with bugs (Basili et al., 1996; Chidamber et al., 1998; Subramanyam and Krishnan, 2003; Gyimothy et al., 2005). These OO metrics are useful for architectural and design evaluation (Chowdhury and Zulkernine, 2011). However, first, they cannot be extracted from non-object oriented languages (e.g., C, Bash). Second, they might not be easily computed by practitioners. In such cases, other code metrics that assess the quality of the software systems should be considered (e.g., CC and LOC). In Table 10 we compare the median CC between blocking and non-blocking files. For the first six projects, we find that blocking files have $\approx 1.2 - 7.6$ times more execution paths than non-blocking files. The Wilcoxon rank-sum test confirms that the difference is significant. For Gentoo, there is no evidence that $CC_{block} > CC_{nonblock}$. However, this does not necessarily mean that blocking/nonblocking files have the same complexity. After performing the opposite hypothesis ($CC_{block} < CC_{nonblock}$), we find that blocking files have statistically less complexity than non-blocking files. After a manual inspection, we find that blocking and non-blocking files in Gentoo are quite different in terms of functionality provided and programming language distribution. Approximately 68% of the blocking files comes

Table 11
Median LOC for blocking and non-blocking files.

Project	LOC_{block}	$LOC_{nonblock}$ (X)
Chromium ***	142 [1.6X]	89
Eclipse ***	122 [1.4X]	88
FreeDesktop ***	588 [1.4X]	409
Mozilla ***	174 [1.4X]	127
NetBeans ***	284 [1.3X]	223
OpenOffice ***	513 [3.7X]	140
Gentoo	121 [0.9X]	130
Fedora ***	755 [12.2X]	62

(***) $p < 0.001$; (**) $p < 0.01$; (*) $p < 0.05$.

Table 12
Performance of blocking files prediction models.

Project	Precision	Recall	F-measure	Acc.
Chromium	71.0%	62.1%	**66.1%**	65.7%
Eclipse	57.9%	61.7%	**59.6%**	64.9%
FreeDesktop	35.7%	70.0%	**45.3%**	71.6%
Mozilla	95.5%	78.8%	**86.3%**	77.4%
NetBeans	39.7%	66.8%	**49.8%**	71.5%
OpenOffice	48.6%	96.4%	**64.5%**	96.7%
Gentoo	76.5%	73.3%	**74.8%**	78.2%
Fedora	84.0%	70.5%	**76.0%**	71.0%

from Portage (Gentoo's package management system), which is mostly written in Python, whereas 40% of the non-blocking files comes from Quagga (routing suite) and X-Server (window system server) which are mostly written in C. For Fedora, we did not have enough data to extract the CC metric. Approximately 98% of the files in Fedora are Patch/Bash files and our metric extraction tool does not support these kind of files.

Lines of code. Although CC is a good measure of structural complexity of a program, it cannot be easily calculated for Bash/Patch files. On the other hand, LOC can be calculated easier than CC for any kind of source code file. Table 11 presents the median LOC of blocking and non-blocking files. Similar to our previous findings, we observe that for most of the projects (the first six projects and Fedora), blocking files have statistically more lines of code (1.3X–12.2X) than non-blocking files. The only exception was Gentoo, for which we find that blocking files are smaller than non-blocking files.

Discussion. Our findings so far indicate that there is a negative impact on the quality of the files affected by blocking bugs. Therefore, practitioners should plan to allocate more QA effort when fixing blocking files. In order to help with the resource allocation, we would like to provide practitioners with a subset of files that are most susceptible to blocking bugs. More precisely, we would like to investigate whether we can build accurate models (trained on file metrics) to predict which buggy files will contain blocking bugs in the future. First, we extract two process metrics (Num. lines modified and Num. commits) and four code metrics (LOC, Cyclomatic, Coupling and Cohesion) for both blocking and non-blocking files analyzed in this RQ. Then, we train decision tree models using such file-metrics and evaluate their performance using the precision, recall and F-measure metrics. For Gentoo and Fedora, we do not consider Cyclomatic, Coupling and Cohesion metrics, since most of the files in these projects are Patch/Bash files. In Table 12, we report the models' performance for each of the projects. The results indicate that our blocking files prediction models can achieve moderate and high F-measure values ranging from 45.3%–86.3%, while at the same time achieving high accuracy values ranging from 64.9%–96.7%. It is important to emphasize that our models are not general models that aim to predict buggy files, but specialized models to predict whether a *buggy file* will be a blocking file. Therefore, our proposed models should be used in conjunction with traditional bug prediction/localization models to first identify buggy files (Nguyen et al., 2011; Zhou et al., 2012; Kim et al., 2013).

> *Files affected by blocking bugs have*
> *1.02–1.18 times less cohesion,*
> *1.15–1.43 times higher coupling,*
> *1.2–7.6 times higher complexity and*
> *1.3–12.2 times more lines of code*
> *than files affected by non-blocking bugs.*

RQ3. Can we build highly accurate models to predict whether a new bug will be a blocking bug?

Motivation. We observed that blocking bugs not only take much longer and require more lines of code to be fixed than non-blocking bugs, but also they negatively impact the affected files in terms of cohesion, coupling, complexity and size. Because of these severe consequences, it is important to identify blocking bugs in order to reduce their impact. Therefore, in this RQ, we want to build prediction models that can help developers to flag blocking bugs early on, so they can shorten the overall fixing time. Additionally, we want to know if we can accurately predict these blocking bugs using the factors that we proposed in Section 2.3.

Approach. We use decision trees based on the C4.5 algorithm as our prediction model, because it is an explainable model that can easily be understood by practitioners. We use stratified 10-fold cross-validation to estimate the accuracy of our models. To evaluate their performance, we use the precision, recall and F-measure metrics. The reported performances of the models are the average of the 10 folds. *Baseline:* In order to have a point of reference for our performance evaluation, we use a random classifier that has a 50/50 chance of predicting two outcomes (e.g., blocking and non-blocking bugs). Prior studies have also used this theoretical model as their baseline (Rahman et al., 2012; Mende and Koschke, 2009; DAmbros et al., 2012; Kamei et al., 2013). Given a 50/50 random classifier, if an infinite number of random predictions are performed, then the precision will be to the percentage of blocking bugs in the data set, and the recall will be to 50%. Additionally, we further compare them to six other machine learning techniques.

Results. In Table 13, we present the performance results of our prediction models. Our models present precision values ranging from 13.7%–45.8%. Comparing these results with those of the baseline models (6.1%–21.9%), our models provide a approximately two-fold improvement over the baseline models in terms of precision.

In terms of recall, our models present better results for six projects with values ranging from 52.9%–66.7%. For the other projects (Eclipse and Gentoo), the recalls were bellow the baseline recall (50%) with values of $\approx$ 47%–49%. Although, we achieved low recall values for some of our projects, what really matters for comparing the per-

Table 13
Performance of the decision tree models.

Project	Decision Tree model			Baseline model		
	Precision	Recall	F-measure	Precision	Recall	F-measure (X)
Chromium	15.7%	59.5%	**24.8% [2.3X]**	6.1%	50%	**10.8%**
Eclipse	14.0%	49.5%	**21.9% [2.0X]**	6.2%	50%	**11.0%**
FreeDesktop	24.8%	60.3%	**35.2% [1.9X]**	11.4%	50%	**18.6%**
Mozilla	36.1%	63.4%	**46.0% [1.6X]**	20.3%	50%	**29.0%**
NetBeans	15.7%	52.9%	**24.2% [2.1X]**	6.6%	50%	**11.6%**
OpenOffice	14.7%	54.2%	**23.1% [2.3X]**	5.6%	50%	**10.0%**
Gentoo	13.7%	47.7%	**21.2% [1.5X]**	7.9%	50%	**13.7%**
Fedora	45.8%	66.7%	**54.3% [1.8X]**	21.9%	50%	**30.5%**

formance of the two models is the F-measure, which is a trade-off between precision and recall.

Our results show that the F-measure values of our prediction models represent an improvement over those of the baseline models for all of the projects. Our F-measure values range from 21.2%–54.3%, whereas the F-measure values of the baseline models range from 10.8%–30.5%. The improvement ratio of our F-measure values vary from $\approx$ 1.5–2.3 folds.

The above results give an idea of the effectiveness of our models with respect to a random classifier. However, there are other popular machine learning techniques besides decision trees that can be used to predict blocking bugs. In Table 14, we compare the performance of our model to six other machine learning techniques namely: Zero-R, Naive–Bayes, kNN, Logistic Regression, Stacked Generalization and Random Forests. The *Zero-R* model presents the highest accuracy across most of the projects (except for Fedora). This happens because the Zero-R always predicts the majority class (e.g., non-blocking bugs), which in our case account for approximately 87% of the bugs in most of the projects. Clearly, it is useless to have a highly accurate model that cannot detect blocking bugs. Therefore, we use the F-measure metric to perform the comparisons. The *Naive–Bayes* model is only slightly better for Chromium, Mozilla and Gentoo with F-measure values ranging from 22.1%–46.6%. In the other five projects, Naive–Bayes performs worse than our model (specially for OpenOffice). The *kNN* model is slightly worse for all of the projects. For example, in Mozilla, kNN achieves a F-measure of 43%, whereas our model achieves a F-measure of 46%. The *Logistic Regression* model performs slightly worse for FreeDesktop, OpenOffice and Fedora, whereas in the other projects, it performs better than our model. For example, in Mozilla, Logistic Regression and Decision Trees achieve F-measures of 49% and 46% respectively. *Random Forests* and *Stacked Generalization* models perform better in all of the projects. In particular, Random Forests significantly outperforms our models with an improvement of 7%–9% for four projects (Chromium, Eclipse, NetBeans and OpenOffice). For example, for the Chromium project, we observe that the F-measure improves from 24.8%–31.7%. However, these two ensemble models do not provide easily explainable models. Practitioners often prefer easy-to-understand models such as decision trees because they can explain why the predictions are the way they are. What we observe is that the decision trees are close to the Random Forests (or Stacked Generalization) in terms of F-measure in many projects, however if one is more concerned about accuracy to detect blocking bugs, the Random Forests would be the best prediction model. If one wants accurate models that are easily explainable, then they would need to sacrifice a bit of accuracy and use the decision tree model.

Table 14
Predictions different algorithms.

Project	Classif.	Precision	Recall	F-measure	Acc.
Chromium	Zero-R	NA	0.0%	0.0%	**93.9%**
	Naive–Bayes	19.6%	51.3%	28.4%	84.3%
	kNN	13.2%	64.8%	21.9%	71.9%
	Logistic Regression	17.2%	61.2%	26.8%	79.7%
	Stacked Gen.	24.2%	41.5%	30.5%	88.5%
	Rand. Forest	27.1%	38.3%	**31.7%**	90.0%
	Decision Tree	15.7%	59.5%	24.8%	78.1%
Eclipse	Zero-R	NA	0.0%	0.0%	**93.8%**
	Naive–Bayes	13.2%	60.2%	21.6%	73.0%
	kNN	11.4%	60.2%	19.2%	68.8%
	Logistic Regression	12.5%	67.3%	21.1%	68.9%
	Stacked Gen.	20.6%	30.9%	24.7%	88.4%
	Rand. Forest	27.7%	30.6%	**29.1%**	90.8%
	Decision Tree	14.0%	49.5%	21.9%	78.1%
FreeDesktop	Zero-R	NA	0.0%	0.0%	**88.6%**
	Naive–Bayes	24.4%	59.5%	34.4%	73.9%
	kNN	20.4%	65.6%	31.1%	66.7%
	Logistic Regression	24.4%	65.5%	35.6%	72.9%
	Stacked Gen.	28.2%	49.1%	35.8%	79.8%
	Rand. Forest	31.9%	46.1%	**37.6%**	82.4%
	Decision Tree	24.8%	60.3%	35.2%	74.5%
Mozilla	Zero-R	NA	0.0%	0.0%	**79.7%**
	Naive–Bayes	35.0%	69.7%	46.6%	67.5%
	kNN	32.5%	63.4%	43.0%	65.7%
	Logistic Regression	38.1%	68.1%	49.0%	71.1%
	Stacked Gen.	39.1%	56.0%	46.0%	73.3%
	Rand. Forest	44.7%	53.2%	**48.6%**	77.1%
	Decision Tree	36.1%	63.4%	46.0%	69.7%
NetBeans	Zero-R	NA	0.0%	0.0%	**93.4%**
	Naive–Bayes	14.4%	61.3%	23.3%	73.3%
	kNN	13.0%	62.9%	21.5%	69.8%
	Logistic Regression	15.2%	63.5%	24.6%	74.3%
	Stacked Gen.	24.0%	37.1%	29.1%	88.1%
	Rand. Forest	30.5%	36.5%	**33.2%**	90.3%
	Decision Tree	15.7%	52.9%	24.2%	78.2%
OpenOffice	Zero-R	NA	0.0%	0.0%	**94.4%**
	Naive Bayes	6.4%	93.7%	12.0%	23.6%
	kNN	11.7%	59.8%	19.6%	72.8%
	Logistic Regression	13.8%	67.1%	22.9%	74.9%
	Stacked Gen.	23.6%	36.3%	28.6%	89.9%
	Rand. Forest	30.7%	36.8%	**33.5%**	91.9%
	Decision Tree	14.7%	54.2%	23.1%	80.0%
Gentoo	Zero-R	NA	0.0%	0.0%	**92.1%**
	Naive Bayes	15.9%	36.5%	22.1%	79.6%
	kNN	10.6%	55.9%	17.8%	59.0%
	Logistic Regression	17.1%	43.9%	24.6%	78.6%
	Stacked Gen.	15.4%	35.4%	21.5%	79.5%
	Rand. Forest	20.9%	29.9%	**24.3%**	85.1%
	Decision Tree	13.6%	47.7%	21.2%	72.0%
Fedora	Zero-R	NA	0.0%	0.0%	78.1%
	Naive Bayes	48.0%	59.7%	53.2%	77.0%
	kNN	38.5%	67.1%	48.9%	69.3%
	Logistic Regression	43.6%	70.2%	53.8%	73.6%
	Stacked Gen.	47.2%	62.6%	53.8%	76.5%
	Rand. Forest	53.5%	59.8%	**56.5%**	**79.8%**
	Decision Tree	45.8%	66.7%	54.3%	75.4%

Discussion. Besides warning about blocking bugs, we would like to advise developers to be careful of factors (in the bug reports) that potentially indicate the presence of blocking bugs. Therefore, we investigate which factor or group of factors have a significant impact on the determination of blocking bugs. We perform Top Node analysis in order to determine which factors are the best indicators of whether a bug will be a blocking bug or not. In the Top Node analysis, we examine the decision trees created by the 10-fold cross validation and we count the occurrences of the factors at each level of the trees. The most relevant

Table 15
Top Node analysis results.

Level	Chromium		Eclipse	
	#	Attribute	#	Attribute
0	10	Description text	8	Rep. Blocking experience
			2	Description text
1	16	Rep. Blocking experience	15	Description text
	4	Comment size	4	Rep. Blocking experience
			1	Comment text
2	22	Reporter	22	Component
	9	Comment size	10	Reporter
	8	Rep. Blocking experience	8	Description text
	1	Description text		
Level	**FreeDesktop**		**Mozilla**	
	#	**Attribute**	**#**	**Attribute**
0	10	Description text	8	Comment text
			2	Description text
1	17	Reporter	18	Description text
	2	Rep. Blocking experience	2	Comment text
	1	Description text		
2	63	Rep. Blocking experience	14	Rep. Blocking experience
	36	Rep. experience	10	Component
	22	Comment size	9	Reporter
	18	Priority	1	Priority
Level	**NetBeans**		**OpenOffice**	
	#	**Attribute**	**#**	**Attribute**
0	10	Description text	7	Comment text
			3	Description text
1	1	Rep. Blocking experience	10	Description text
	19	Comment text	8	Rep. Blocking experience
			2	Num. CC
2	25	Component	11	Rep. Blocking experience
	8	Description text	8	Rep. experience
	3	Reporter	8	Num. CC
	2	Rep. Blocking experience	4	Reporter
Level	**Gentoo**		**Fedora**	
	#	**Attribute**	**#**	**Attribute**
0	10	Description text	6	Comment text
			4	Description text
1	10	Reporter	15	Rep. Blocking experience
			3	Description text
			2	Comment text
2	41	Rep. Blocking experience	15	Component
	37	Rep. experience	8	Rep. Blocking experience
	18	Description size	7	Reporter
	17	Comment text	5	Num. CC

factors are always close to the root node (level 0, 1 and 2). As we traverse down the tree, the factors become less relevant. For example, in Fig. 2, the comment is the most relevant factor because it is the root of the tree (level 0). The next two most relevant factors are num-CC and reporter's experience (both in level 1) and so on. In the Top Node analysis, the combination of the level in which a factor is found along with its occurrences determines the importance of such as factor. If, for example, the product factor appears as the root in seven of the ten trees and the platform factor appears as the root in the remaining, we would report product as the first most important factor and platform as the second most important factor.

Table 15 reports the Top Node analysis results for our eight projects. The description and the comments included in the bugs are the most important factors. For example, the description text is the most important factor in Chromium, FreeDesktop, NetBeans and Gentoo; and the second most important factor in Eclipse, Mozilla, OpenOffice and Fedora. Likewise, the comment text is the most important factor in Mozilla, OpenOffice and Fedora; and the third most important in NetBeans. Words such as *"dtrace"*, *"pthreads"*, *"scheduling"*, *"glitches"* and *"underestimate"* are associated with blocking bugs by the Naive Bayes Classifier. On the other hand, words such as *"duplicate"*, *"harmless"*, *"evolution"*, *"enhancement"* and *"upgrading"* are associated with non-blocking bugs.

The experience of reporting previous blocking bugs (Rep. Blocking experience) is the most important factor for Eclipse, and the second most important for Chromium and NetBeans. It also appears consistently in the second and third levels of all the projects.

Other factors such as priority, component, number of developers in the CC list, reporter's name, reporter's experience, and description-size are only present in the second and third levels of two or less projects. This means that among the factors reported in Table 15, such factors are the less important.

We can build prediction models that can achieve F-measure values ranging from 21% to 54% when detecting blocking bugs.

In addition, we find that the description and comment text are the most important factors in determining blocking bugs for the majority of the projects, followed by the Rep. Blocking experience.

Table 16
Performance of the decision tree models using data collected after 24 hours after the initial submission.

Project	Precision	Recall	F-measure
Chromium	9.1%	49.9%	**15.3%**
Eclipse	9.2%	47.0%	**15.4%**
FreeDesktop	20.4%	73.6%	**31.9%**
Mozilla	29.0%	76.7%	**42.1%**
NetBeans	12.8%	59.3%	**21.1%**
OpenOffice	15.9%	65.9%	**25.6%**
Gentoo	8.6%	39.0%	**14.1%**
Fedora	27.6%	67.2%	**39.2%**

Table 17
Number of different reporters (nominal levels).

Project	Num. unique reporters
Chromium	16,209
Eclipse	14,616
FreeDesktop	1475
Mozilla	9945
NetBeans	6056
OpenOffice	14,412
Gentoo	3333
Fedora	35,882

4. Relaxing the data collection process

4.1. Prediction models using data available after 24 hours after the bug report submission

So far, we trained our prediction models with bug report information collected within the 24 hours after the initial submission. One limitation of this approach is that a large number of bug reports do not have any information recorded for some of the factors. For example, we found that around 92%–98% of the bug reports have empty values for severity, priority, priority has increased, platform and product. Therefore, it is worth investigating whether relaxing the data collection period could improve the performance of our prediction models.

In Table 16, we present the performance of prediction models trained on data collected without the *"24 hours restriction"*. More precisely, for non-blocking bugs we used data before their blocking-dates and for blocking-bugs we used data before their blocking-date. From Table 16, it can be seen that the F-measures range from 14.1%–42.1%. These values are lower that the F-measures of our original models (21.2%–54.3%) presented in Table 13. This suggests that collecting data after the blocking-date and closing-date is not worth the effort. One possible explanation for the performance degradation of the prediction models is that relaxing the data collection process introduces noise into the data set.

4.2. Dealing with the Reporter's name factor

While building our prediction models, we faced computational issues caused by the reporter's name factor. In our data set, we found approximately 100,000 different reporters. We summarize the number of unique reporters for all of the projects in Table 17. Having a nominal factor with such high number of levels is computational expensive and impractical. For example, a logistic model trained on the Chromium data would create 16,209 dummy variable to account for the different levels of the nominal factor reporter's name. To overcome this issue and because we are interested in the impact of non-sporadic developers, we reduced the number of levels by considering the top K reporters (of each project) with the highest number of reported bugs. The remaining reporters were grouped into a level named "others". In our work, we considered a value of $K = 200$ (i.e., the top 200 reporters) for the prediction models in RQ3.

Instead of performing a sensitivity analysis to determine whether other values of K (e.g., 50, 100, 300, etc.) have a potential effect on the models' performance, we followed a slightly different approach. First, we removed the reporter's name from the data set, and then re-built the prediction models. Our experiments show that these models achieved F-measures of 19.7% to 53.2%, which are similar to the performance of models considering the reporter's name built in RQ3 (F-measures of 21.2% to 54.3%). These findings suggest that the reporter's name does not play a significant role in predicting blocking bugs. A detailed information about the models built in this section (precision, recall and F-measure) for all of the projects can be found in our online appendix (Valdivia-Garcia, 2018).

5. Threats to validity

Internal validity: We used standard statistical libraries and methods to perform our predictions and statistical analysis (e.g., Weka and R programming). We also rely on a commercial tool (Scitools Understand) to extract the code metrics. Although these tools are not perfect, they have been used by other researchers in the past for bug prediction (Kim et al., 2011; Bird et al., 2009; Bettenburg et al., 2012).

Construct validity: The main threat here is the quality of the ground truth for blocking bugs. We used the information in the "Blocks" field of the bug reports to determine blocking and non-blocking bugs. In some cases, developers could have mistakenly filled that field. We inspected a subset of the blocking bugs in each of our projects and we found no evidence of such a mistake.

For the nominal factor: reporter name, we considered the top $K = 200$ reporters and grouped the remaining reporters into one level. This approach significantly reduced the number of different levels for that factor. Although using a different number K for the top reporters may change our results, we found that reporter name does not play a significant role in the prediction models. In addition, we used the number of previous reported bugs as the experience of a reporter. In some cases, using the number of previous reported bugs may not be indicative of actual developer experience, however similar measures were used in prior studies (Shihab et al., 2013).

In RQ2, we used Lack of Cohesion, Coupling between Objects, Cyclomatic Complexity and LOC as proxy metrics to quantify the impact of blocking bugs at file level. Although these metrics have also been reported to be useful for architectural evaluation, other architectural and design metrics such code smell metrics may quantify differently the effects of blocking bugs on software systems.

Our data set suffers from the class imbalance problem. In most of the projects, the percentage of blocking bugs account for less than 12% of the total data. This causes the classifier not to learn to identify the blocking bugs very well. To mitigate this problem, we use re-sampling of our training data and stratified cross-validation. To calculate the Bayesian-scores, we filtered out all the words with less than five occurrences in the corpora. Increasing this threshold will produce different scores, however, it will introduce more noise. Furthermore, the Bayesian-score of a description/comment is based on the combined probability of the fifteen most important words of the description/comment. Changing this number may impact our finding.

Our work did not considered bugs with status other than resolved or closed, because we wanted to investigate only well

identified blocking and non-blocking bugs. However, unlike non-blocking bugs, the blocking bugs may not be restricted to verified or closed bugs. In most of the cases, bugs marked as blocking bugs remain that way until their closed-date. In the future, we plan to include these blocking bugs in order to improve the accuracy of our model.

Many of the projects do not follow any formal guidelines to label bug reports in the commits. To extract the links between bug reports and commits, we tried to match the bug-IDs in the messages of the commits with different regular expressions that may not consider all possible patterns. Therefore, our data set might not be complete and/or contain false positive bug-fixes. To reduce the impact of this threat, we manually inspected a subset of the linked commits and their respective bug reports generated by each regular expression. Additionally, we might miss actual bug-fixes in which the developer did not include the related bug-report. Although more sophisticated methods (Wu et al., 2011; Nguyen et al., 2012) can improve the identification of bug-fixes, our approach was able to extract a large number of bug-fixes (263,345) which is a rich data set suitable for the purpose of this study.

In Software Bug Prediction research area, there are two well-known model evaluation settings: Cross-validation (D'Ambros et al., 2010; Giger et al., 2011; Tantithamthavorn et al., 2017) and Forward-release/Cross-release validation (Premraj and Herzig, 2011; Kamei et al., 2010,Ma et al.) (i.e., train models with data from previous releases and test them with data of the next release). Although, Forward-release is closer to what can be deployed in a real environment, both approaches (and most of the studies in the Software Bug Prediction area) assume little or no autocorrelation in the dataset. In other words, the instances in the dataset do not have any significant temporal dependency among them. Since we are using cross-validation to evaluate our models performance, we are implicitly assuming no autocorrelation in our dataset and as a result, our prediction models do no account for this kind of correlation. Therefore, if there was a high autocorrelation in our dataset, then other techniques such time series analysis could potentially improve the performance of our models.

External validity: Including more software systems improves the generality of research findings (which is a difficult and important aspect in SE research). In this work, we increase the generality of our results by studying 609,800 bug reports and 263,345 bug-fixing commits from eight projects. That said, not always having a set of diverse projects is better because it might introduce outliers that can impact the generality of the findings. To combat this, we considered long-live and large open-source mostly written in Java and C++.

6. Related work

Re-opened bug prediction: Similar to our work, however focusing on different types of bugs, prior work by Shihab et al. (2013) studied re-opened bugs on three open-source projects and proposed prediction models based on decision trees in order to detect such type of bugs. In their work, they used 22 different factors from 4 dimensions to train their models. Xia et al. (2013) compared the performance of different machine learning methods to predict re-opened bugs. They found that Bagging and Decision Table algorithms presents better results than decision trees when predicting re-opened bugs. Zimmermann et al. (2012) also investigated and characterized re-opened bugs in Windows. They performed a survey to identify possible causes of reopened bugs and built statistical models to determine the impact of various factors. The extracted factors in our data sets are similar to those used in the previous works (specially in Shihab et al., 2013; Xia et al., 2013). Additionally, we also use decision trees as our prediction models. However our work differs in that we are not interested in predicting reopened bugs, but instead in predicting blocking bugs.

Fix-time prediction: A prediction model for estimating the bug's fixing effort based on previous bugs with similar textual information has been proposed by Weiss et al. (2007). Given a new bug report, they use kNN along with text similarity techniques for finding the bugs with closely related descriptions. The average effort of these bugs are used to estimate the fixing effort of the given bug report. Panjer (2007) used decision trees and other machine learning methods to predict the lifetime of Eclipse bugs. Since the classifiers do not deal with a continuous response variable, they discretized the lifetime into seven categories. Their models considered only primitive factors taken directly from the bug database (e.g., fixer, severity, component, number of comments, etc.) and achieved accuracies of 31–34%. Marks et al. (2011) used Random Forest to predict bug's fixing time. Using the bugs from Eclipse and Mozilla, they examined the fixing time along 3 dimensions: Location, reporter and description. Following an approach similar to Panjer, Marks et al. discretized the fixing time into 3 categories (within 1 month, within 1 year, more than a year). For both projects their method was able to yield an accuracy of about 65%. In our work, we also used decision trees as prediction models, but instead of predicting the bug's lifetime, we try to predict blocking bugs. Bhattacharya and Neamtiu (2011) performed multivariate regression testing to determine the relationship strength between various bug report factors and the fixing time. They found that the dependency among software bugs (i.e., blocking dependency) is an important factor that contributes to predict the fixing time. Our work is not directly related to bug-fixing time prediction, but the results in Bhattacharya and Neamtiu (2011) motivate the study and characterization of blocking bugs.

Severity/Priority prediction: Other works focused on the prediction of specific bug report fields (Lamkanfi et al., 2010; 2011; Menzies and Marcus, 2008; Sharma et al., 2012). Lamkanfi et al. (2010) trained Naive–Bayes classifiers with textual information from bug reports on Eclipse and Mozilla to determine the severity of such bugs. In another paper (Lamkanfi et al., 2011), the authors compared the performance of four machine learning algorithms (Naive–Bayes, Naive–Bayes Multinomial, kNN and SVM) for predicting the bug severity and found that Naive Bayes Multinomial is the fastest and most accurate. Menzies and Marcus (2008) used a rule-based algorithm for predicting the severity of bug reports using their textual descriptions. They evaluated their method using data from a NASA's bug tracking system. Sharma et al. (2012) evaluated different classifiers for predicting the priority of bugs in OpenOffice and Eclipse. Their prediction models achieved accuracies above 70%. Our work differs from the previous studies in that we used that information to predict blocking bug rather than the severity/priority. In fact, we used the severity and priority of the bug reports in our factors.

Bug triaging and Duplicate bug detection: Other studies use textual information from bug reports such as summary, description and execution trace for semi-automatic triage process (Cubranic and Murphy, 2004,Anvik et al.,Anvik and Murphy, 2011; Rahman et al., 2009) and bug duplicate detection (Runeson et al., 2007; Wang et al., 2008; Bettenburg et al., 2008; Jalbert and Weimer, 2008; Sun et al., 2011). The key idea in the majority of these works is to apply natural language processing (NLP) and information retrieval techniques in order to find a set of bug reports that are similar to a target bug (new bug). Based on this suggested list of similar bugs, the triager can, for example, recommend the ap-

propriate developer to incoming bugs or filter out those already-reported bugs. Similar to these works, we included textual-based factors (comments and description) in our prediction models with the difference that instead of using a vector space representation, we converted them into numerical factors following the same approach used by Shihab et al. (2013) and Ibrahim et al. (2010).

Bug localization: Prior studies have proposed method to localize buggy files of a given new bug report (Nguyen et al., 2011; Zhou et al., 2012; Kim et al., 2013). Nguyen et al. (2011) proposed BugScout, a new topic model based on Latent Dirichlet Allocation that can assist practitioners in automatically locating buggy files associated to a bug report. They exploited the technical aspects shared by the textual content of files between code and bug reports in order to correlate buggy files and bugs. Zhou et al. (2012) proposed BugLocator, a method based on a revised Vector Space Model for locating source code files relevant to a initial bug report. To rank potential buggy files, the method uses (a) text similarity between a new bug report and the source code files, (b) historical data of prior fixed reports and (c) source code size. Kim et al. (2013) proposed a two-phase machine learning model to suggests the files that are likely to be fixed in response to a given bug report. In the first phase, their model assesses whether the bug report has sufficient information. In the second phase, the model proceeds to predict files to be fixed only if it believes that the bug report is predictable. To train the model, the authors considered only basic metadata and initial comments posted within 24 hours from the bug submission. Our work differs from these previous studies in that their goal is to recommend relevant files related to a given bug report, whereas our main goal is to predict whether a given bug report will be a blocking bug or not. That said, since these bug localization techniques use textual information to do their recommendations, they can easily be used in conjunction with our prediction models to identify potential buggy files with blocking bugs (as we pointed out at the end of RQ2).

7. Conclusion and future work

Blocking bugs increase the maintenance cost, cause delays in the release of software projects, and may result in a loss of market share. Our empirical study shows that blocking bugs take up 2 times longer and require 1.2–4.7 times more lines of code to be fixed than non-blocking bugs. On further analysis, we found that files affected by blocking bugs are more negatively impacted in terms of cohesion, coupling complexity and size than files affected by non-blocking bugs. For example, we find that files with blocking bugs are 1.3–12.2 times bigger (in LOC) than files with non-blocking bugs. Based on our findings, we suggest that practitioners should allocate more QA effort when fixing blocking bugs and files related to them.

Since these bugs have such severe consequences, it is important to identify them early on in order to reduce their impact. In this paper, we build prediction models based on decision trees to predict whether a bug will be a blocking bug or not. As our data set, we used 14 factors extracted from the bug repositories of eight large open source projects. The results of our investigation shows that our models achieve 13%–45% precision, 47%–66% recall and 21%–54% F-measure when predicting blocking bugs. On the other hand, our Top Node analysis shows that the most important factors to determine blocking bugs are the description, comments and the reporter's blocking experience. In the future, we plan to model the blocking dependency of the bug reports as a graph structure and study it using network analysis. Particularly, we are interested in deriving network measures to incorporate them in our prediction models and examine whether they improve the prediction performance (Zimmermann et al. followed a similar approach in Zimmermann and Nagappan, 2008). We also plan to extend this work by performing feature selection on our factors. Employing feature selection may improve the performance of our models since it removes redundant factors. From the architectural point of view, we would like to exploit the source code topology to identify hotspots in the architecture of software systems caused by blocking bugs. Furthermore, we plan to perform qualitative analyses similar to Tan et al. (2014) at factor and file level to better understand (a) the influence of certain factors and (b) the characteristics of buggy files affected by blocking bugs.

Supplementary material

Supplementary material associated with this article can be found, in the online version, at 10.1016/j.jss.2018.03.053

References

Antoniol, G., Ayari, K., Penta, M.D., Khomh, F., Guéhéneuc, Y.G., 2008. Is it a bug or an enhancement?: a text-based approach to classify change requests. In: Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds. ACM, p. 23.

Anvik, J., Hiew, L., Murphy, G. C.,. Who should fix this bug? In: Proceedings of the 28th International Conference on Software Engineering, pp. 361–370.

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

Basili, V.R., Briand, L.C., Melo, W.L., 1996. A validation of object-oriented design metrics as quality indicators. IEEE Trans. Softw. Eng. 22 (10), 751–761.

Bettenburg, N., Nagappan, M., Hassan, A.E., 2012. Think locally, act globally: Improving defect and effort prediction models. In: Proceedings of the 9th IEEE Working Conference on Mining Software Repositories. IEEE Press, pp. 60–69.

Bettenburg, N., Premraj, R., Zimmermann, T., Kim, S., 2008. Duplicate bug reports considered harmful really. In: Software Maintenance, 2008. ICSM 2008. IEEE International Conference on, pp. 337–345.

Bhattacharya, P., Neamtiu, I., 2011. Bug-fix time prediction models: Can we do better? In: Proceedings of the 8th Working Conference on Mining Software Repositories. ACM, pp. 207–210.

Bird, C., Nagappan, N., Gall, H., Murphy, B., Devanbu, P., 2009. Putting it all together: Using socio-technical networks to predict failures. In: Software Reliability Engineering, 2009. ISSRE'09. 20th International Symposium on. IEEE, pp. 109–119.

Breiman, L., 2001. Random forests. Mach. Learn. 45 (1), 5–32.

Caruana, R., Niculescu-Mizil, A., 2006. An empirical comparison of supervised learning algorithms. In: Proceedings of the 23rd International Conference on Machine Learning. ACM, pp. 161–168.

Chidamber, S.R., Darcy, D.P., Kemerer, C.F., 1998. Managerial use of metrics for object-oriented software: an exploratory analysis. Softw. Eng. IEEE Trans. 24 (8), 629–639.

Chowdhury, I., Zulkernine, M., 2011. Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. J. Syst. Archit. 57 (3), 294–313.

Cubranic, D., Murphy, G.C., 2004. Automatic bug triage using text categorization. In: SEKE 2004: Proceedings of the Sixteenth International Conference on Software Engineering and Knowledge Engineering. KSI Press, pp. 92–97.

D'Ambros, M., Lanza, M., Robbes, R., 2009. On the relationship between change coupling and software defects. In: Working Conference on Reverse Engineering, pp. 135–144.

D'Ambros, M., Lanza, M., Robbes, R., 2010. An extensive comparison of bug prediction approaches. In: 2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010). IEEE, pp. 31–41.

DAmbros, M., Lanza, M., Robbes, R., 2012. Evaluating defect prediction approaches: a benchmark and an extensive comparison. Empirical Softw. Eng. 17 (4–5), 531–577.

Efron, B., 1983. Estimating the error rate of a prediction rule: improvement on cross-validation. J. Am. Stat. Assoc. 78 (382), 316–331.

Erlikh, L., 2000. Leveraging legacy system dollars for e-business. IT Prof. 2 (3), 17–23.

Giger, E., Pinzger, M., Gall, H., 2010. Predicting the fix time of bugs. In: Proceedings of the 2Nd International Workshop on Recommendation Systems for Software Engineering. ACM, pp. 52–56.

Giger, E., Pinzger, M., Gall, H.C., 2011. Comparing fine-grained source code changes and code churn for bug prediction. In: Proceeding of the 8th working conference on Mining software repositories - MSR '11. ACM Press, p. 83.

Graham, P., 2003. A plan for spam. Available on: http://paulgraham.com/spam.html.

Graves, T.L., Karr, A.F., Marron, J.S., Siy, H., 2000. Predicting fault incidence using software change history. IEEE Trans. Softw. Eng. 26 (7), 653–661.

Gyimothy, T., Ferenc, R., Siket, I., 2005. Empirical validation of object-oriented metrics on open source software for fault prediction. IEEE Trans. Softw. Eng. 31 (10), 897–910.

Hassan, A.E., Zhang, K., 2006. Using decision trees to predict the certification result of a build. In: Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on. IEEE, pp. 189–198.

Hulse, J.V., Khoshgoftaar, T.M., Napolitano, A., 2007. Experimental perspectives on learning from imbalanced data. In: Proceedings of the 24th International Conference on Machine Learning. ACM, pp. 935–942.
Ibrahim, W., Bettenburg, N., Shihab, E., Adams, B., Hassan, A., 2010. Should i contribute to this discussion? In: Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on, pp. 181–190.
Jalbert, N., Weimer, W., 2008. Automated duplicate detection for bug tracking systems. In: Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on, pp. 52–61.
Kamei, Y., Matsumoto, S., Monden, A., Matsumoto, K.i., Adams, B., Hassan, A.E., 2010. Revisiting common bug prediction findings using effort-aware models. In: 2010 IEEE International Conference on Software Maintenance. IEEE, pp. 1–10.
Kamei, Y., Shihab, E., Adams, B., Hassan, A.E., Mockus, A., Sinha, A., Ubayashi, N., 2013. A large-scale empirical study of just-in-time quality assurance,. Softw. Eng. IEEE Trans. 39 (6), 757–773.
Khoshgoftaar, T.M., Seliya, N., 2004. Comparative assessment of software quality classification techniques: an empirical case study. Empirical Softw. Eng. 9 (3), 229–257.
Kim, D., Tao, Y., Kim, S., Zeller, A., 2013. Where should we fix this bug? a two-phase recommendation model. Softw. Eng. IEEE Trans. 39 (11), 1597–1610.
Kim, S., Zhang, H., Wu, R., Gong, L., 2011. Dealing with noise in defect prediction. In: Software Engineering (ICSE), 2011 33rd International Conference on. IEEE, pp. 481–490.
Lamkanfi, A., Demeyer, S., Giger, E., Goethals, B., 2010. Predicting the severity of a reported bug. In: Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on, pp. 1–10.
Lamkanfi, A., Demeyer, S., Soetens, Q., Verdonck, T., 2011. Comparing mining algorithms for predicting the severity of a reported bug. In: Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on, pp. 249–258.
Ma, W., Chen, L., Yang, Y., Zhou, Y., Xu, B.,. Empirical analysis of network measures for effort-aware fault-proneness prediction. Inf. Softw. Technol.
Marks, L., Zou, Y., Hassan, A.E., 2011. Studying the fix-time for bugs in large open source projects. In: Proceedings of the 7th International Conference on Predictive Models in Software Engineering. ACM, pp. 11:1–11:8.
Mende, T., Koschke, R., 2009. Revisiting the evaluation of defect prediction models. In: Proceedings of the 5th International Conference on Predictor Models in Software Engineering. ACM, p. 7.
Menzies, T., Marcus, A., 2008. Automated severity assessment of software defect reports. In: Software Maintenance, 2008. ICSM 2008. IEEE International Conference on, pp. 346–355.
Monard, M.C., Batista, G., 2002. Learning with skewed class distributions, advances in logic. Artif. Intell. Robot. 173–180.
Moser, R., Pedrycz, W., Succi, G., 2008. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In: ICSE '08: Proceedings of the 30th international conference on Software engineering, pp. 181–190.
Nguyen, A.T., Nguyen, T.T., Al-Kofahi, J., Nguyen, H.V., Nguyen, T.N., 2011. A topic-based approach for narrowing the search space of buggy files from a bug report. In: Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on. IEEE, pp. 263–272.
Nguyen, A.T., Nguyen, T.T., Nguyen, H.A., Nguyen, T.N., 2012. Multi-layered approach for recovering links between bug reports and fixes. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE '12. ACM, New York, NY, USA, pp. 63:1–63:11.
Panjer, L.D., 2007. Predicting eclipse bug lifetimes. In: Mining Software Repositories, 2007. ICSE Workshops MSR '07. Fourth International Workshop on, p. 29.
Premraj, R., Herzig, K., 2011. Network versus code metrics to predict defects: A replication study. In: 2011 International Symposium on Empirical Software Engineering and Measurement. IEEE, pp. 215–224.
Quinlan, J.R., 1993. C4.5: Programs for Machine Learning. Morgan Kaufmann Publishers Inc..
Rahman, F., Posnett, D., Devanbu, P., 2012. Recalling the "imprecision" of cross-project defect prediction. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering - FSE '12. ACM Press, p. 1.
Rahman, F., Posnett, D., Herraiz, I., Devanbu, P., 2013. Sample size vs. bias in defect prediction. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013. ACM Press, p. 147.
Rahman, M.M., Ruhe, G., Zimmermann, T., 2009. Optimized assignment of developers for fixing bugs an initial evaluation for eclipse projects. In: Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement. IEEE Computer Society, pp. 439–442.
Runeson, P., Alexandersson, M., Nyholm, O., 2007. Detection of duplicate defect reports using natural language processing. In: Software Engineering, 2007. ICSE 2007. 29th International Conference on, pp. 499–510.
Sharma, M., Bedi, P., Chaturvedi, K., Singh, V., 2012. Predicting the priority of a reported bug using machine learning techniques and cross project validation. In: Intelligent Systems Design and Applications (ISDA), 2012 12th International Conference on, pp. 539–545.
Shihab, E., Ihara, A., Kamei, Y., Ibrahim, W., Ohira, M., Adams, B., Hassan, A., Matsumoto, K.i., 2013. Studying re-opened bugs in open source software. Empirical Softw. Eng. 18 (5), 1005–1042.
Subramanyam, R., Krishnan, M.S., 2003. Empirical analysis of ck metrics for object-oriented design complexity: implications for software defects. IEEE Trans. Softw. Eng. 29 (4), 297–310.
Sun, C., Lo, D., Khoo, S.C., Jiang, J., 2011. Towards more accurate retrieval of duplicate bug reports. In: Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on, pp. 253–262.
Tan, L., Liu, C., Li, Z., Wang, X., Zhou, Y., Zhai, C., 2014. Bug characteristics in open source software. Empirical Softw. Eng. 19 (6), 1665–1705.
Tantithamthavorn, C., McIntosh, S., Hassan, A.E., Matsumoto, K., 2017. An empirical comparison of model validation techniques for defect prediction models. IEEE Trans. Software Eng. 43 (1), 1–18.
Tassey, G., 2002. The economic impacts of inadequate infrastructure for software testing. Technical Report.
Valdivia-Garcia, H., 2018. Characterizing and prediction blocking bugs in open source projects - appendix. https://github.com/harold-valdivia-garcia/blocking-bugs/blob/master/jss-appx.pdf.
Valdivia-Garcia, H., Shihab, E., 2014. Characterizing and predicting blocking bugs in open source projects. In: Proceedings of the 11th Working Conference on Mining Software Repositories. ACM, pp. 72–81.
Wang, X., Zhang, L., Xie, T., Anvik, J., Sun, J., 2008. An approach to detecting duplicate bug reports using natural language and execution information. In: Software Engineering, 2008. ICSE '08. ACM/IEEE 30th International Conference on, pp. 461–470.
Weiss, C., Premraj, R., Zimmermann, T., Zeller, A., 2007. How long will it take to fix this bug? In: Proceedings of the Fourth International Workshop on Mining Software Repositories. IEEE Computer Society, p. 1.
Weiss, G.M., 2004. Mining with rarity: a unifying framework. SIGKDD Explor. Newsl. 6 (1), 7–19.
Wolpert, D.H., 1992. Stacked generalization. Neural Netw. 5 (2), 241–259.
Wu, R., Zhang, H., Kim, S., Cheung, S.C., 2011. Relink: recovering links between bugs and changes. In: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering. ACM, pp. 15–25.
Xia, X., Lo, D., Wang, X., Yang, X., Li, S., Sun, J., 2013. A comparative study of supervised learning algorithms for re-opened bug prediction. In: Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on, pp. 331–334.
Zaman, S., Adams, B., Hassan, A.E., 2012. A qualitative study on performance bugs. In: Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on. IEEE, pp. 199–208.
Zhou, J., Zhang, H., Lo, D., 2012. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In: Software Engineering (ICSE), 2012 34th International Conference on. IEEE, pp. 14–24.
Zimmermann, T., Nagappan, N., 2008. Predicting defects using network analysis on dependency graphs. In: Proceedings of the 30th International Conference on Software Engineering, pp. 531–540.
Zimmermann, T., Nagappan, N., Guo, P.J., Murphy, B., 2012. Characterizing and predicting which bugs get reopened. In: Proceedings of the 2012 International Conference on Software Engineering, pp. 1074–1083.
Zou, W., Hu, Y., Xuan, J., Jiang, H., 2011. Towards training set reduction for bug triage. In: Proceedings of the 2011 IEEE 35th Annual Computer Software and Applications Conference. IEEE Computer Society, pp. 576–581.