Characterizing and Predicting Blocking Bugs in Open Source Projects

Harold Valdivia Garcia and Emad Shihab
Department of Software Engineering
Rochester Institute of Technology
Rochester, NY, USA
{hv1710, emad.shihab}@rit.edu

ABSTRACT

As software becomes increasingly important, its quality becomes an increasingly important issue. Therefore, prior work focused on software quality and proposed many prediction models to identify the location of software bugs, to estimate their fixing-time, etc. However, one special type of severe bugs is *blocking bugs*. *Blocking bugs* are software bugs that prevent other bugs from being fixed. These blocking bugs may increase maintenance costs, reduce overall quality and delay the release of the software systems.

In this paper, we study *blocking-bugs* in six open source projects and propose a model to predict them. Our goal is to help developers identify these blocking bugs early on. We collect the bug reports from the bug tracking systems of the projects, then we obtain 14 different factors related to, for example, the textual description of the bug, the location the bug is found in and the people involved with the bug. Based on these factors we build decision trees for each project to predict whether a bug will be a *blocking bug* or not. Then, we analyze these decision trees in order to determine which factors best indicate these blocking bugs. Our results show that our prediction models achieve F-measures of 15-42%, which is a two- to four-fold improvement over the baseline random predictors. We also find that the most important factors in determining blocking bugs are the comment text, comment size, the number of developers in the CC list of the bug report and the reporter's experience. Our analysis shows that our models reduce the median time to identify a blocking bug by 3-18 days.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Software Quality Analysis

Keywords

Process Metrics, Code Metrics, Post-release Defects

MSR'14, May 31 – June 1, 2014, Hyderabad, India
ACM 978-1-4503-2863-0/14/05
http://dx.doi.org/10.1145/2597073.2597099

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 [1] reported that approximately 90% of the software life-cycle cost is consumed by software maintenance activities. Two year 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 [2].

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., [3, 4, 5]). 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 [6, 7, 8], automatic assignment of bug severity/priority [9, 10], predicting fixing time [11, 12, 13, 14] and assisting in bug triaging [15, 16, 17]. More recently, prior work focused on specific types of issues such reopened bugs, performance bugs and enhancement requests [18, 19, 20, 21]

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 take approximately two to three times longer to be fixed compared to non-blocked bugs. For example, in one of our case studies, the median number of days to resolve a blocking bug is 48, whereas the median for *non-blocking* bugs is 16 days.

To reduce the impact of blocking bugs, we build prediction models in order to flag the blocking bugs early on for developers. In particular, we mine the bug repositories from six open source projects namely: Chromium, Eclipse, FreeDesktop, Mozilla, NetBeans and OpenOffice to extract 14 different factors related to the textual information of the bug, the location the bug is found and the people

[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.

who reported the bug. We determine the ground truth (i.e., information of whether a bug is blocking or non-blocking) from the bug repositories. Based on these factors and employing machine learning techniques such as decision trees (C4.5), Naive Bayes, kNN, Random Forests and Zero-R, we build our prediction models. Additionally, we perform a analysis [22] in order to determine which factors best identify blocking bugs. In particular, we would like to answer the following research questions:

RQ1 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 15%-42%.

RQ2 Which factors are the best indicators of blocking bugs?
We find that the bug comments, the number of developers in the CC list and the bug reporter 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. Section 3 discusses and characterizes blocking bugs. Section 4 presents our case study. We compare the performance of different classifiers in Section 5. We discuss the related work in Section 6. Section 7 highlights the threats to validity. Section 8 concludes the paper and discusses future work.

2. APPROACH

In this section, we describe our approach as shown in Figure 1. First, we discuss the data used in our case study and we list the factors extracted from the bugs reports. Second, we describe the prediction models and the performance metrics used in our study.

2.1 Data Used in the Case Studies

In order to perform our study, we used the bug reports from six different projects namely: Chromium, Eclipse, FreeDesktop, Mozilla, NetBeans and OpenOffice. Chromium is a web browser developed by Google and used as the development branch of Google Chrome. Eclipse is a popular multi-language IDE written in Java, well known for its system of plugins that allows customization of its programming environment. FreeDesktop is an umbrella project hosting sub projects such as Xorg (official implementation of the X Window System), Mesa (free implementation of the OpenGL specification), etc. Mozilla is a framework and umbrella project that hosts and develops products such as Firefox, Thunderbird, Bugzilla, etc. NetBeans is also another popular IDE written in Java. Although it is meant for java development, it also provides support for PHP and C/C++ development. OpenOffice is an office suite initiated by Sun Microsystem and currently developed by Apache. The reason we chose these projects is because they are mature and long-lived open sources projects, with a large amount of bug reports.

Table 1 shows the number of closed bugs for each project. For Chromium, we extracted all bugs published before *April 17, 2013*. The total number of extracted bugs was 206,125 bugs. We removed bugs with a status other than verified or closed. Bugs with empty fields were also filtered out, because our prediction models require instances with non-empty values. After this preprocessing step, the number of bugs was reduced to 39,619, of which 924 (2.3%) were blocking bugs and 38,695 were non-blocking bugs.

For the remaining five projects, we extracted bug reports that were verified or closed before *October 18, 2013*. Similar to the Chromium extraction, we filtered out those bugs with empty fields.

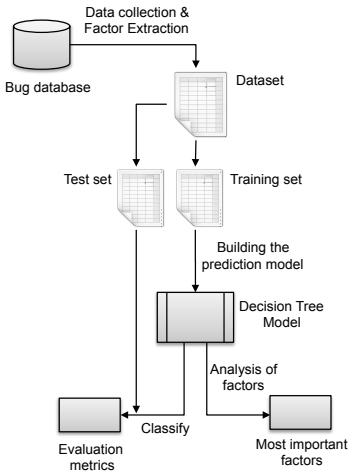

Figure 1: Approach overview

The total number of valid bugs for these five projects was 363,343. From the table, we can see that Eclipse is the largest project with 130,694 valid bugs. We can also notice that for most of the projects, the percentages of blocking bugs was less that 3% of the set of bugs. The only two exceptions were FreeDesktop and Mozilla with 8.9% and 12.5%, respectively.

2.2 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 surveyed prior work. For example, Sun *et al.* [23] included factors such product, component, priority, etc in their models to detect duplicate bugs. Lamkanfi *et al.* [10, 24] used textual information to predict bug severities. Wang *et al.* and Jalbert *et al.* [7], [25] used text mining to identify duplicate bug reports. Zimmermann *et al.* [19] 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 [18], 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, Thunderbird, 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, GNU/Linux, Android, 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. We excluded the severity for Chromium because, we found that around 98% of its bugs have an empty value for this factor.
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. For blocking bugs, we only counted the developers subscribed right before they were identified as being 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. For blocking bugs, we only consider comments posted before the bugs were marked as blocking.
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. Similar to comment size, for the blocking bugs, we only include the comments posted before the bug was marked as blocking.
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. Since we are interested in the impact of non-sporadic developers, we group reporters with less than five contributions into a category named "others". Additionally, we try to consolidate reporters with more than one email. We perform a semi-automatic inspection of the emails. The levenshtein distance is calculated for every pair of emails and then we check manually the pairs with a distance less than five.
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.

Once the aforementioned factors are extracted, we use them to build classifiers to predict blocking bugs. It is important to mention 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.

2.3 Textual Factor Preprocessing

Although all of the bugs reports in the bug repositories were extracted, we needed to discard a number of bug reports. The total number of extracted bug reports for all the projects were 582,938. After calculating the factors, filtering out those bugs with empty values and preprocessing the comments we kept only 402,962 valid bug reports. Some of these factors were extracted directly from the bug reports, however, other factors required special preprocessing.

Table 1: Dataset description

Project	Blocking	Non-blocking	Total
Chromium	924 [2.3%]	38,695 [97.7%]	39,619
Eclipse	3,654 [2.8%]	127,040 [97.2%]	130,694
FreeDesktop	424 [8.9%]	4361 [91.1%]	4,785
Mozilla	8,476 [12.5%]	59,121 [87.5%]	67,597
NetBeans	2,424 [3.2%]	74,307 [96.8%]	76,731
OpenOffice	2,520 [3.0%]	81,016 [97.0%]	83,536
All Projects	18,422 [4.6%]	384,540 [95.4%]	402,962

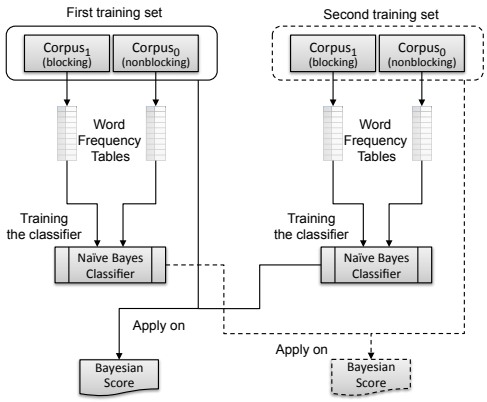

Figure 2: Converting textual factor into Bayesian-score

The description and comments in bug reports are two factors that required special preprocessing, since they represent a rich source of unstructured information. These factors contain discussions about the bugs and can also provide snapshots of the progress and status of such bugs. However, not all the comments are valid, at least for the purpose of this work. For example:

- After two months of closing the bug 302295 in Eclipse, a developer identified that the bug 309220 in Eclipse was its duplicate and posted the comment *"Bug 309220 has been marked as a duplicate of this bug"*.
- The bug 7430 in Eclipse is a blocking bug and has 22 comments before the closing date, but 8 of them were posted after it was marked as blocking. If we want to predict and warn about these blocking bugs, no information after the blocking-date (i.e., the earliest date on which the bug is marked as blocking bug) should be used.

To deal with cases like those described earlier, we followed two criteria. First, for the non-blocking bugs, we filtered out comments written after the closing-date. Second, for the blocking bugs, we kept only comments before the blocking-date. Once we collected the valid comments and the descriptions, the next step was to convert them into numerical values.

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 [18, 26], we followed this simple approach. In Figure 2, 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 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 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 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 [27]. Here, *"relevant"* means those words with probability close to 1 or 0.

2.4 Prediction Models

For each of our case study projects, we use our proposed factors to train a decision tree classifier 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, Random Forests and Zero-R.

2.4.1 Decision Tree Classifier

We use a tree-based classifier to perform our predictions. One of the benefits of tree-based classifiers 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 [28] 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 begins with an empty tree. At each level, the goal is try to find the feature that maximize the information gain. Consider for example a data set with p feature-columns: $X_1, X_2, \ldots X_p$ (e.g., severity, platform, comment-size, etc) and a class-column: C (e.g., Blocking/Non-Blocking). The C4.5 algorithm splits the data into two subsets with rules of the form $X_i < b$ if the feature is numeric or into multiple subsets if the feature is nominal. The algorithm is applied recursively to the partitions until every leaf contains only records of the same class, or no further splitting can be performed or the number of records in the leaf reaches a predefined threshold [28].

In Figure 3, 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

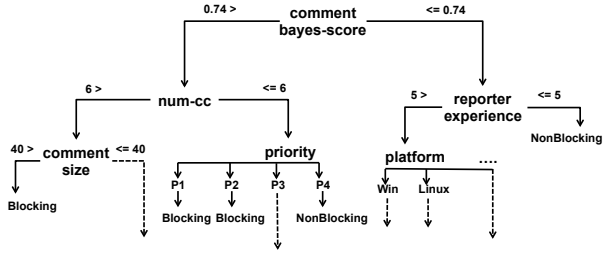

Figure 3: Example of a Decision Tree

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.

2.4.2 Naive Bayes Classifier

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.

The naive bayes algorithm classifies a new record $x = \langle x_1, \ldots x_p \rangle$ to the class k that maximizes the conditional probability:

$$P(C = k | X = \langle x_1, \ldots x_p \rangle)$$

In other words, it chooses the most probable class k based on the piece of information x (and the training set as we will see later). Under the assumption that the factors are random independent (which is uncommon in real world) and using the Bayes' theorem, the classifier can be re-written as:

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

Here, $P(C = k)$ is called the prior-probability and can be estimated with the percentage of training records labeled as k (e.g., percentage of blocking or non-blocking). The conditional probabilities $P(x_i | C = k)$ also known as the likelihood 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.

On the other hand, if we want the probability of a text to belong to a blocking bug, we need to calculate: $\frac{P(C=block)\, P(text|C=block)}{P(text)}$. In Section 2.3 (Textual Factor Preprocessing), we presented a more detailed discussion of this matter.

2.4.3 K-Nearest Neighbor Classifier

The k-nearest neighbor classifier is a simple, yet powerful memory based technique which has been used with relative success in previous bug prediction works [13, 24]. 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. For example, if two bug reports A and B have the same

platform, then the distance between $A_{platform}$ and $B_{platform}$ is zero. On the other hand, if the platforms are distinct, then the distance is one.

2.4.4 Random Forests Classifier

Random Forests [29] is an ensemble classifier 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 classifier more resilient to noise in the data set. For building a Random Forests of m decision trees, we need to generate m bootstrap samples taken from the training set (i.e., a random sample with replacement of the same size as the training set) and use each of them for training a tree. Unlike C4.5 models, weak trees in a Random Forest are not pruned and the splitting at each node only consider a random subset of the factors (which minimizes the correlation among the trees). In general, this classifier outperforms simple decision trees in terms of prediction accuracy [30].

2.4.5 Zero-R Classifier

Zero-R (no rule) is the simplest classifier 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 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 [31]. If a classifier tends to maximize the accuracy, then it can perform very well by simply ignoring the minority class [32, 33].

We use a confusion matrix to evaluate the effectiveness of the derived models. A confusion matrix stores the correct and incorrect decisions made by a classifier. For example, if a bug is classified as blocking when it truly is blocking, then the classification is a true positive (TP); if it is classified as blocking when actually it is non-blocking, then the classification is a false positive (FP); if it is classified as non-blocking when it is actually blocking, then the classification is a false negative (FN); finally, if it is classified as non-blocking and it truly is non-blocking, then the classification is a true negative (TN). Table 2 summarizes these four possible outcomes.

Using the values stored in the confusion matrix, we calculate the widely used Precision, Recall, F-measure and Accuracy metrics for the blocking bugs to evaluate the performance of the predictive models:

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

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

We use stratified 10-fold cross-validation [34] 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.

Table 2: Confusion matrix

		True class: Blocking	True class: Non-blocking
Classified as	Blocking	TP	FP
	Non-blocking	FN	TN

3. BUG CHARACTERIZATION

Before presenting our case study, we discuss three aspects of the blocking bugs that we consider important to better understand their impact.

3.1 Fixing Time

Blocking bugs are harmful for the maintenance process, they delay the repair of other bugs (i.e., *blocked bugs*) not because there is not enough resources to fix them, but because the components affected by these *blocked bugs* rely on other components that need to be fixed first. On the other hand, bugs without dependencies (i.e., **non-blocking**) on other bugs can be fixed right away and as a result, their fixing time might be shorter.

Table 3: Median fixing time in days

Project	Blocking	Non-blocking
Chromium	48	16
Eclipse	146	51
FreeDesktop	78	37
Mozilla	100	42
Netbeans	225	134
Open-Office	142	74

Table 3 reports the median of the fixing-time for **blocking/non-blocking** bugs. Through all the projects, we observe that the fixing-time for **non-blocking** bugs is shorter than for the **blocking** bugs. To see if there is a significant difference between the blocking and non-blocking bugs, we performed an unpaired Wildcoxon-test for the hypothesis $H_a : t_{blocking} > t_{nonblocking}$. Table 4 shows the fixing-time difference reported by the test. The *p-values* were significant for all projects (*p-value* < 0.001), meaning that the fixing-time for blocking bugs is statistically significantly longer than the fixing-time for non-blocking bugs.

The median time to address a blocking bug 15-40 days longer than the median time it takes to address a non-blocking bug.

3.2 Time to Identify a Blocking Bug

Consider, for example, bug 121120 in OpenOffice that was created on *2012-09-27* and reported as blocker for the first time on *2012-11-16*. This means, it took approximately 50 days to identify this bug as blocking bug.

Some blocking bugs are harder to identify than other blocking bugs. In this case, we can see this period of time as a measure

Table 4: Wildcoxon-Test for blocking and non-blocking bugs

Project	$t_{blocking} - t_{nonblocking}$
Chromium	15.9 ***
Eclipse	52.7 ***
FreeDesktop	19.2 ***
Mozilla	37.0 ***
Netbeans	16.8 ***
Open-Office	40.1 ***

(***) p < 0.001

Table 5: Median time to identify a blocking bug in days

Project	Time to identify a blocking bug
Chromium	8
Eclipse	17
FreeDesktop	9
Mozilla	18
NetBeans	5
OpenOffice	3

of the degree of difficulty to identify blocking bugs. For example, the bug 79342 in OpenOffice was reported as blocking in 30 days, while bug 80769 in OpenOffice was reported as blocking in 5 days.

To calculate the *time to identify a blocking bug* for the projects using Bugzilla, we look for the date of the first appearance of the tag "blocks" in the history of the bugs. When no information can be obtained from the bug's history (e.g., bug 376141 in Eclipse), we look up the histories of the related blocked bugs and select the date of the first reported dependency. For the Chromium project, we calculate this information by looking up the labels associated to the comments of the bugs and the comments of the related blocked bugs. In the Google's bug tracking system the history of a bug is embedded in the comments.

In Table 5, we present the median *time to identify a blocking bug* for our six case study projects. We observe that for the Netbeans and OpenOffice projects, half the of blocking bugs were reported as blocking in less than 3-5 days. For the Chromium and FreeDesktop projects, the median *time to identify a blocking bug* was 8-9 days, while, for Eclipse and Mozilla, it took 17-18 days on median.

The median time it takes to identify a blocking bug is 3-18 days

3.3 Degree of Blockiness

The degree of blockiness refers to the number of bugs that depends on the same bug and it can be seen as a measure of severity of a blocking bug. For example, the bug 309165 in Eclipse is blocking four other bugs, this means that its degree of blockiness is equal to four. The presence of bugs with high blockiness might become bottlenecks for the maintenance and evolution of the code. These blocking bugs could be the result of low quality design in critical components (e.g., tightly coupled), significant changes in the requirements, etc.

Table 6 reports the percentages of bugs by degree of blockiness. Only the percentages for degrees ≤ 7 were reported. We notice that the FreeDesktop project does only have bugs that block 1 or 2 or at most 3 bugs. The number of bugs with degree "1" is 1082 (74.8%) in Chromium, 3039 (83.1%) in Eclipse, 368 (86.7%) in FreeDesktop, 5671 (66.9%) in Mozilla, 2217 (91.4%) in NetBeans and 2304 (91.4%) in OpenOffice. At first sight, it is easy to see that approximately 96% of the bugs for all projects have a blockiness of

Table 6: Degree of blockiness

Blockiness	Chromium	Eclipse	FreeDesk	Mozilla	NetBeans	OpenOffice
1	74.84%	83.17%	86.79%	66.91%	91.46%	91.43%
2	18.78%	11.71%	11.56%	20.07%	6.64%	6.94%
3	3.46%	2.76%	1.65%	6.30%	1.20%	0.95%
4	1.45%	1.09%	0.00%	2.76%	0.29%	0.40%
5	0.76%	0.44%	0.00%	1.36%	0.25%	0.04%
6	0.48%	0.22%	0.00%	0.97%	0.00%	0.16%
$7 \geq$	0.23%	0.61%	0.00%	1.64%	0.17%	0.08%

"1" or "2". As a consequence, bugs with high degree of blockiness are uncommon.

We manually examined some of the bugs with blockiness > 7 for Eclipse and NetBeans. Many of the Eclipse bugs were enhancements with low priority (P3, P4) instead of real defects. On the other hand, for the NetBeans bugs, we found that indeed they were defects with high priority (P1, P2).

The majority (96%) of blocking bugs block at most two bugs.

4. CASE STUDY

This section reports the results of our study on six open source projects and answers our two research questions. First, we built different prediction models to detect whether a bug will be or not a blocking bug. Second, we performed Top Node analysis to determine which of the collected factors are good indicators to identify blocking bugs.

RQ1. Can we build highly accurate models to predict whether a new bug will be a blocking bug?

Motivation. We see that blocking bugs take much longer to be fixed than non-blocking bugs. We want to help developers flag blocking bugs early on, so that they can avoid that wasted of time and effort. Therefore, we want to build prediction models to help developers to identify these blocking bugs and we want to know if we can accurately predict these blocking bugs using the factors that we proposed.

Approach. Our prediction models are based on the C4.5 decision tree algorithm, 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 performance of the models are the average of the 10 folds. These metrics are compared to the performance of our baseline model. Here, our baseline corresponds to a model that randomly predicts blocking bugs. The precision for our baseline classifier is the percentage of blocking bugs in the data set. Since our baseline model is a random classifier with two possible outcomes (e.g., blocking/non-blocking), its recall is 50%.

Results. In Table 7, we present the performance results of our prediction models. The precision values achieved by our decision tree models are better than the precision values of the baselines for all the projects. Our models present precision values ranging from 9.1% to 29%. Comparing these results with those of the baseline models (2.3% - 12.54%), we observe that our prediction models provide a $\sim$2 to $\sim$5 fold improvement over the baseline models in terms of precision.

The recall values achieved by the decision trees of four of the six projects are better than the recall values of their corresponding baseline models. The only two exceptions are Chromium and Eclipse with recall values (49% and 47%) slightly below the baseline recall (50%). For the four other projects our models achieve recall values of 59.3%-76.7%. We notice that our prediction models provide a

Table 7: Performance of the decision tree models

	Decision Tree model			Baseline model		
Project	**Precision**	**Recall**	**F-measure**	**Precision**	**Recall**	**F-measure**
Chromium	9.1%	49.9%	15.3%	2.33%	50.0%	4.46%
Eclipse	9.2%	47.0%	15.4%	2.80%	50.0%	5.30%
FreeDesktop	20.4%	73.6%	31.9%	8.86%	50.0%	15.05%
Mozilla	29.0%	76.7%	42.1%	12.54%	50.0%	20.05%
NetBeans	12.8%	59.3%	21.1%	3.16%	50.0%	5.94%
OpenOffice	15.9%	65.9%	25.6%	3.02%	50.0%	5.69%

$\sim$0.9 to $\sim$1.5 fold improvement over the baseline models in terms of recall. Although, we achieved low recall values for some of our projects, what it really matters for comparing the performance of two models is the F-measure, which is a trade-off between precision and recall.

Similar to the previous metrics, the F-measure values of our prediction models present an improvement with respect to the F-measure values of the baseline models. Our F-measure values range from 15.3% to 42.1%, whereas that the F-measure values of the baseline models range from 4.4% to 20.05%. The improvement ratio of our F-measure values vary from $\sim$2 to $\sim$4.5 folds.

We can build highly accurate prediction models that can achieve F-measure values of 15%-42% when detecting blocking bugs.

RQ2. Which factors are the best indicators of blocking bugs?

Motivation. Besides warning about potential blocking bugs, we would like to identify the factor or group of factors that have a significant impact on the determination of these blocking bugs. With this information, we can advise developers of which factors to use in order to detect blocking bugs early on.

Approach. 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 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 Figure 3, 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.

Results. Table 8 reports the Top Node analysis results for our six projects. The comments included in the bugs and the sizes of those comments are the most important factors. In five of the projects, the comment text is the most important factor. The only exception is Mozilla in which the comment text is the second most important factor. The comment-size is the second most important factor in Chromium and OpenOffice and the third most important in Mozilla and 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 number in the CC list is the most important factor for Mozilla. It also appears in the second level (level 1) of NetBeans, OpenOffice and Chromium as the second, third and forth most important factor respectively.

The reporter's name is the second most important factor for Eclipse and FreeDesktop. It also appears in the third level (level 2) of Chromium, NetBeans and OpenOffice. Although its importance differs from project to project, it consistently appears in five of the six projects.

The description of a bug is a relevant factor only for Chromium (third most important). Other factors such as priority, product, reporter's experience, description-size, etc are only present in the second and third levels of two or less projects. In other words, among the factors reported in Table 8, they are the less important.

The Comment text is the most important factor in determining blocking bugs for all the projects except Mozilla project in which the most important factor is the Number in the CC list.

5. COMPARISON USING DIFFERENT CLASSIFIERS

Besides decision tree classifiers, there are other popular machine learning algorithms that can be used to predict whether a bug will be a blocking bug or not. In this section, we compare the performance of four other classifiers namely: Zero-R, Naive Bayes, kNN and Random Forests.

In Table 9, we report the precision, recall and F-measure achieved by these classifiers. If we look at the accuracies, we see that Zero-R presents the highest accuracy among all the projects. This happens because, first, the Zero-R algorithm always predicts the majority class (i.e., non-blocking bugs) and second, the percentages of non-blocking bugs account for approximately 95% of the total (in most of the projects). Clearly, it is useless to have a highly accurate model that cannot detect blocking bugs. For this reason, we use the F-measure metric to compare the performance of the four classifiers against the performance of our prediction model.

The Naive Bayes algorithm is only slightly better for Chromium and Eclipse with F-measure values equal to 16.9% and 15.5%. In the other projects, it performs worse than our model.

The models based on the kNN algorithm are slightly worse for all the projects. To give an example, consider the Mozilla project which has a F-measure of 34.9%. In this case, our model with a F-measure of 42.1% outperforms the kNN algorithm for over 7.2%.

The Random Forests classifier performs better in all the cases. For example, for the Chromium project, we observe that the F-measure improves from 15.3% to 22.8%. However, the Random Forests classifier does 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 C4.5 classifier is close to the Random Forests classifier in terms of F-measure, however if one is more concerned about accuracy to detect blocking bugs, the Random Forests would be the best classifier. If one wants accurate models that are easily explainable, then they would need to sacrifice a bit of accuracy and use the C4.5 classifier.

6. RELATED WORK

Re-opened bug prediction: Similar to our work, however focusing on different types of bugs, prior work by Shihab *et al.* [18] studied re-opened bugs on three open-source projects and proposed prediction models based on decision trees in order to detect such

Table 8: Top Node analysis results

Level	Chromium		Eclipse		FreeDesktop	
	#	Attribute	#	Attribute	#	Attribute
0	5	Comment text	10	Comment text	10	Comment text
	5	Comment size				
1	17	Description text	12	Reporter	4	Reporter
	3	Num. CC	8	Comment text	3	Num. CC
					2	Priority has Increased
2	15	Num. CC	704	Comment size	14	Rep. Blocking experience
	14	Reporter	613	Description size	13	Rep. experience
	3	Description text	536	Comment text	8	Description text
					8	Comment size
Level	**Mozilla**		**NetBeans**		**OpenOffice**	
	#	Attribute	#	Attribute	#	Attribute
0	10	Num. CC	10	Comment text	10	Comment text
1	15	Comment text	10	Num. CC	10	Comment size
	1	Comment size	10	Comment size	9	Num. CC
					1	Rep. Blocking experience
2	15	Comment size	12	Reporter	21	Comment text
	11	Priority	28	Comment text	10	Comment size
	1	Comment text			9	Reporter
	1	Product				

Table 9: Predictions different algorithms

Project	Classif.	Precision	Recall	F-measure	Acc.
Chromium	Zero-R	NA	0%	0%	**97.6%**
	Naive Bayes	10.0%	54.2%	16.9%	81.5%
	kNN	9.0%	47.1%	15.1%	81.5%
	Rand. Forest	**18.6%**	29.6%	**22.8%**	93.0%
	C4.5	9.1%	**49.9%**	15.3%	80.7%
Eclipse	Zero-R	NA	0%	0%	**97.2%**
	Naive Bayes	8.8%	**66.4%**	15.5%	79.7%
	kNN	8.1%	53.0%	14.0%	81.8%
	Rand. Forest	**16.5%**	24.0%	**19.5%**	94.5%
	C4.5	9.2%	47.0%	15.4%	85.5%
FreeDesktop	Zero-R	NA	0%	0%	**91.1%**
	Naive Bayes	18.7%	**74.3%**	29.9%	69.0%
	kNN	19.4%	72.6%	30.6%	70.7%
	Rand. Forest	**27.8%**	60.2%	**37.9%**	82.4%
	C4.5	20.4%	73.6%	31.9%	72.0%
Mozilla	Zero-R	NA	0%	0%	**87.4%**
	Naive Bayes	29.5%	68.0%	41.1%	75.6%
	kNN	23.3%	69.3%	34.9%	67.5%
	Rand. Forest	**36.1%**	53.6%	**43.2%**	82.3%
	C4.5	29.0%	**76.7%**	42.1%	73.6%
NetBeans	Zero-R	NA	0%	0%	**96.8%**
	Naive Bayes	9.9%	**73.3%**	17.3%	77.1%
	kNN	11.4%	59.3%	19.1%	84.2%
	Rand. Forest	**26.3%**	37.6%	**30.9%**	94.7%
	C4.5	12.8%	59.3%	21.1%	86.0%
OpenOffice	Zero-R	NA	0%	0%	**96.9%**
	Naive Bayes	12.9%	**78.1%**	22.1%	83.3%
	kNN	14.2%	66.1%	23.4%	87.0%
	Rand. Forest	**32.9%**	46.7%	**38.6%**	95.5%
	C4.5	15.9%	65.9%	25.6%	88.4%

type of bugs. In their work, they used 22 different factors from 4 dimensions to train their models. Xia *et al.* in [35] 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.* [19] 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 [18, 35]). 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.* [13]. 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 *et al.* in [12] 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.* [11] 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.

Severity/Priority prediction: Other works focused on the prediction of specific bug report fields [9, 10, 24, 36]. Lamkanfi *et al.* [10] trained Naive Bayes classifiers with textual information from bug reports on Eclipse and Mozilla to determine the severity of such bugs. In another paper [24], 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 *et al.* [36] 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.* [9] 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 [15, 17, 37, 38] and bug duplicate detection [6, 7, 8, 23, 25]. 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 appropriate 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 [18], [26].

Bhattacharya *et al.* [39] 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 [39] motivate the study and characterization of blocking bugs.

7. THREATS TO VALIDITY

Internal Validity. For the reporter name, we grouped the reporters with less than five contributions into one category. This approach significantly reduced the number of different reporters. Considering all of the reporters may introduce noise and therefore impact in our results. 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 [18]. Our data set suffers from the class imbalance problem. In most of the projects, the percentage of blocking bugs account for less than 5% 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.

External Validity. In this work, we studied 402,962 bug reports from six open source projects, therefore our findings may not generalize well to commercial software projects. In fact, although we examined large open source projects that cover a wide range of products and domains, there are other projects that use different software processes, bug tracking systems, etc and therefore our results may not generalize to all of them.

8. 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. 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 six large open source projects. The results of our investigation shows that our models achieve 9-29% precision, 47-76% recall and 15-42% 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 comments, comment-size, number of developers in the CC list and the reporter's 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 [40]). 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. Furthermore, we plan to perform more qualitative analysis on the different factors (e.g., severity and priority) in order to better understand the influence of such a factors. Our results show that blocking bugs take longer to be fixed than non-blocking bugs, however it is unclear if blocking bugs require more effort and resources than non-blocking bugs. To tackle this question, we plan to link bug reports with information from the version control systems, leverage metrics at commit level and perform a quantitative analysis that may help us to confirm or refute our intuition that blocking bugs indeed require more effort.

9. REFERENCES

[1] L. Erlikh, "Leveraging legacy system dollars for e-business," *IT Professional*, vol. 2, no. 3, pp. 17–23, May 2000.

[2] G. Tassey, "The economic impacts of inadequate infrastructure for software testing," Tech. Rep., 2002.

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

[4] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting fault incidence using software change history," *IEEE Transactions of Software Engineering*, vol. 26, no. 7, pp. 653–661, July 2000.

[5] R. Moser, W. Pedrycz, and G. Succi, "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*, 2008, pp. 181–190.

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

[7] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, "An approach to detecting duplicate bug reports using natural language and execution information," in *Software Engineering, 2008. ICSE '08. ACM/IEEE 30th International Conference on*, 2008, pp. 461–470.

[8] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim, "Duplicate bug reports considered harmful really," in *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, 2008, pp. 337–345.

[9] M. Sharma, P. Bedi, K. Chaturvedi, and V. Singh, "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*, 2012, pp. 539–545.

[10] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals, "Predicting the severity of a reported bug," in *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, 2010, pp. 1–10.

[11] L. Marks, Y. Zou, and A. E. Hassan, "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, 2011, pp. 11:1–11:8.

[12] L. D. Panjer, "Predicting eclipse bug lifetimes," in *Mining Software Repositories, 2007. ICSE Workshops MSR '07. Fourth International Workshop on*, 2007, pp. 29–29.

[13] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller, "How long will it take to fix this bug?" in *Proceedings of the Fourth International Workshop on Mining Software Repositories*. IEEE Computer Society, 2007.

[14] E. Giger, M. Pinzger, and H. Gall, "Predicting the fix time of bugs," in *Proceedings of the 2Nd International Workshop on Recommendation Systems for Software Engineering*. ACM, 2010, pp. 52–56.

[15] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in *Proceedings of the 28th International Conference on Software Engineering*. ACM, 2006, pp. 361–370.

[16] W. Zou, Y. Hu, J. Xuan, and H. Jiang, "Towards training set reduction for bug triage," in *Proceedings of the 2011 IEEE 35th Annual Computer Software and Applications Conference*. IEEE Computer Society, 2011, pp. 576–581.

[17] J. Anvik and G. C. Murphy, "Reducing the effort of bug report triage: Recommenders for development-oriented decisions," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 3, pp. 10:1–10:35, Aug. 2011.

[18] E. Shihab, A. Ihara, Y. Kamei, W. Ibrahim, M. Ohira, B. Adams, A. Hassan, and K.-i. Matsumoto, "Studying re-opened bugs in open source software," *Empirical Software Engineering*, vol. 18, no. 5, pp. 1005–1042, 2013.

[19] T. Zimmermann, N. Nagappan, P. J. Guo, and B. Murphy, "Characterizing and predicting which bugs get reopened," in *Proceedings of the 2012 International Conference on Software Engineering*, 2012, pp. 1074–1083.

[20] S. Zaman, B. Adams, and A. E. Hassan, "A qualitative study on performance bugs," in *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*. IEEE, 2012, pp. 199–208.

[21] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc, "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, 2008, pp. 23:304–23:318.

[22] A. E. Hassan and K. Zhang, "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, 2006, pp. 189–198.

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

[24] A. Lamkanfi, S. Demeyer, Q. Soetens, and T. Verdonck, "Comparing mining algorithms for predicting the severity of a reported bug," in *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*, 2011, pp. 249–258.

[25] N. Jalbert and W. Weimer, "Automated duplicate detection for bug tracking systems," in *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, 2008, pp. 52–61.

[26] W. Ibrahim, N. Bettenburg, E. Shihab, B. Adams, and A. Hassan, "Should i contribute to this discussion?" in *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, 2010, pp. 181–190.

[27] P. Graham, "A plan for spam," Available on: http://paulgraham.com/spam.html, Aug. 2003.

[28] J. R. Quinlan, *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., 1993.

[29] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[30] R. Caruana and A. Niculescu-Mizil, "An empirical comparison of supervised learning algorithms," in *Proceedings of the 23rd International Conference on Machine Learning*. ACM, 2006, pp. 161–168.

[31] M. C. Monard and G. Batista, "Learning with skewed class distributions," *Advances in Logic, Artificial Intelligence and Robotics*, pp. 173–180, 2002.

[32] J. Van Hulse, T. M. Khoshgoftaar, and A. Napolitano, "Experimental perspectives on learning from imbalanced data," in *Proceedings of the 24th International Conference on Machine Learning*. ACM, 2007, pp. 935–942.

[33] G. M. Weiss, "Mining with rarity: A unifying framework," *SIGKDD Explor. Newsl.*, vol. 6, no. 1, pp. 7–19, Jun. 2004.

[34] B. Efron, "Estimating the error rate of a prediction rule: improvement on cross-validation," *Journal of the American Statistical Association*, vol. 78, no. 382, pp. 316–331, 1983.

[35] X. Xia, D. Lo, X. Wang, X. Yang, S. Li, and J. Sun, "A comparative study of supervised learning algorithms for re-opened bug prediction," in *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*, 2013, pp. 331–334.

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

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

[38] M. M. Rahman, G. Ruhe, and T. Zimmermann, "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, 2009, pp. 439–442.

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

[40] T. Zimmermann and N. Nagappan, "Predicting defects using network analysis on dependency graphs," in *Proceedings of the 30th International Conference on Software Engineering*, 2008, pp. 531–540.