Filtering Bug Reports for Fix-Time Analysis

Ahmed Lamkanfi, Serge Demeyer
LORE - Lab On Reengineering — University of Antwerp, Belgium

***Abstract*—Several studies have experimented with data mining algorithms to predict the fix-time of reported bugs. Unfortunately, the fix-times as reported in typical open-source cases are heavily skewed with a significant amount of reports registering fix-times less than a few minutes. Consequently, we propose to include an additional filtering step to improve the quality of the underlying data in order to gain better results. Using a small-scale replication of a previously published bug fix-time prediction experiment, we show that the additional filtering of reported bugs indeed improves the outcome of the results.**

***Index Terms*—Software engineering, fix-time, bug reports, preprocessing, data mining**

I. Introduction

Many open-source projects provide a bug tracking system allowing their users to report bugs they have encountered. During bug triaging, a software development team must decide how soon bugs must be fixed. However, the number of reported bugs for popular open-source projects is usually quite high[1]. Therefore, tool support to aid a development team in estimating the time needed to fix a particular bug is desirable.

Several efforts have been presented to estimate or predict the time needed to fix a particular reported bug. These studies propose to use data mining techniques to predict the fix-time of a newly reported bug. Panjer et al. use logistic regression to predict the fix-time of a bug in Eclipse providing an accuracy up to 34.9% [1]. Bougie et al. replicated the previous study, but now on FreeBSD instead of Eclipse, obtaining an accuracy of only 19.49% [2]. Giger et al. used decision trees where the fix-time is predicted as either *slow* or *fast* [3]. Here, an accuracy of over 60-70% is obtained. The experiments proposed in these studies typically leverage a large history of reported bugs to gain insights and predict the time needed to fix a reported bug.

However, particularly when dealing with large data sets, it is common for the data set to contain irregularities [4]. This also holds when we are dealing with software repositories. Bird et. al. found strong evidence of a systematic bias in bug-fix data sets [5]. Similarly, Hindle et al. distinguish small commits from large commits in version control systems [6]. Subsequently, the fix-time information as extracted from the history of reported bugs are also prone to contain irregularities or outliers in the case of bug reports. These outliers may "confuse" the data mining techniques when predicting the fix-time. Removing outliers can improve the quality of the data and may have a positive impact on the accuracy of the predictions. In this paper, we investigate how the fix-times of reported bugs are distributed so that we can identify potential outliers. Our hypothesis is that these outliers may negatively impact the accuracy when predicting the fix-time of reported bugs. For this purpose, we make observations about the fix-time of 10 software systems drawn from two major open-source projects, namely Eclipse and Mozilla. Furthermore, we perform a small experiment comparing the accuracy before and after filtering out the outliers. In this study, we make the following contributions.

- We observe that the fix-times are heavily skewed resulting in a significant amount of reports registering fix-times less than a few minutes.
- We show that when we filter outliers from our collection of bug reports, the accuracy of the predictions of the fix-times tends to improve.

The paper is structured as follows. First, Section II provides some background about fix-time of bug reports and make our observations of the fix-time using a number of open-source software systems. The setup of our experiment to predict the fix-time of bug together with the evaluation of the accuracy of the predictions is then proposed in Section III. Finally, Section IV summarizes the results and points out future work.

II. Bug Reports Analysis

In this section, we analyze the fix-times of reported bugs from two major open-source projects: Eclipse and Mozilla. Based on the observations we make from our analysis, we propose a filtering step which we will discuss in Section III.

A. Fix-time

The software projects examined in this study, use Bugzilla[2] as their bug-tracking system. During the life-time of a bug, a status is assigned to each bug. A typical reported bug in Bugzilla starts with the *unconfirmed* status, which is then followed by *new* when a bug triager can confirm the bug. The triager subsequently assigns the bug to an appropriate developer changing the status of the bug to *assigned*. Whenever the bug is resolved, the status is set to *resolved*. In this study, the fix-time of a bug is considered as the time between the bug is reported and the time it was resolved.

$$fixtime(bug_i) = time\_resolved(bug_i) - time\_opened(bug_i)$$

[1] A software project like Eclipse received over 2.764 bug reports over a period of 3 months (between 01/10/2009-01/01/2010)

[2] Bugzilla: www.bugzilla.org

Some bugs that have not been appropriately fixed can be reopened at a later stage. These reopened bugs are also not considered in this study because the time that it took to reopen the particular bug, can bias the fix-times.

B. Case Selection

We use 10 software systems drawn from across two major open-source projects, namely Eclipse and Mozilla, to investigate the fix-times. Both projects use Bugzilla as their bug tracking system. Eclipse has also been used in other similar studies investigating the fix-times of reported bugs [1, 3] making it worthwhile to include in this study. Table I lists the selected software systems from the two projects together with the total number of available bugs and the respective reporting periods.

TABLE I: The Selected Software Systems with the Corresponding Number of Bugs and Reporting Periods.

Project	Nr. of Bugs	Period
Eclipse Platform	76.456	Oct. 2001 - Oct. 2007
Eclipse PDE	11.117	Oct. 2001 - Oct. 2007
Eclipse JDT	41.691	Oct. 2001 - Oct. 2007
Eclipse CDT	11.468	Oct. 2001 - Oct. 2007
Eclipse GEF	1.587	Oct. 2001 - Oct. 2007
Mozilla Core	143.542	Mar. 1997 - Jul. 2008
Mozilla Bugzilla	19.135	Mar. 2003 - Jul. 2008
Mozilla Firefox	79.272	Jul. 1999 - Jul. 2008
Mozilla Thunderbird	23.408	Jan. 2000 - Jul. 2008
Mozilla Seamonkey	85.143	Nov. 1995 - Jul. 2008

C. Observations

Next, we investigate the fix-times of the reported bugs based on the software systems from Table I. In Table II, we show some basic statistics about the minimum, median and maximum of the fix-times for each of the different cases we selected in our study. Furthermore, we use the visualization technique of so-called box-plots where various statistics of the fix-times are visualized using boxes: the minimum, lower quartile, median, upper quartile and the maximum. Figure 1 shows the box-plots obtained from the fix-times expressed in number of days of the different software systems. Since this number of days becomes very large with some bugs, we present the fix-times using a logarithmic scale in the box-plots.

For example, in the case of the Eclipse Platform case, we notice from Figure 1 that the fastest bugs were fixed in less than one minute. Approximately 25 % of the bugs are fixed within 1 day, 50 % are fixed between 1 and 100 days while another 25 % of the bugs took more than 100 days. The other selected software systems from Eclipse display a similar trend: many bugs are fixed within a few minutes when the bug is reported. The fact that bugs are fixed within a timespan of a few minutes is a conspicuous observation. This does not only hold for the Eclipse cases, but can be generalized to the reported bugs of the software systems from Mozilla. This observation can be explained by how developers contribute their code and how bugs are reported. In some cases, a developer has already the necessary source-code changes ready to fix a particular bug they have encountered and which has not yet been reported. After committing the source-code changes fixing a bug to the version control system, the developer files a report of that particular bug he/she just fixed in order to make sure that the bug is tracked by the bug tracking system. We have presented this observation to the developers of the Mozilla project and they have acknowledged this observation.

TABLE II: Different Statistics for the Selected Cases.

Project	Minimum	Median	Maximum
Eclipse Platform	10 seconds	8.5 days	9.1 years
Eclipse PDE	12 seconds	4.7 days	5.9 years
Eclipse JDT	10 seconds	4.3 days	7.9 years
Eclipse CDT	9 seconds	8.8 days	7.2 years
Eclipse GEF	8 seconds	13 days	7.2 years
Mozilla Core	11 seconds	13.8 days	11.5 years
Mozilla Bugzilla	3 seconds	2.7 days	10.7 years
Mozilla Firefox	13 seconds	7.6 days	11.1 years
Mozilla Thunderbird	18 seconds	32 days	10.3 years
Mozilla Seamonkey	14 seconds	5.8 days	12.7 years

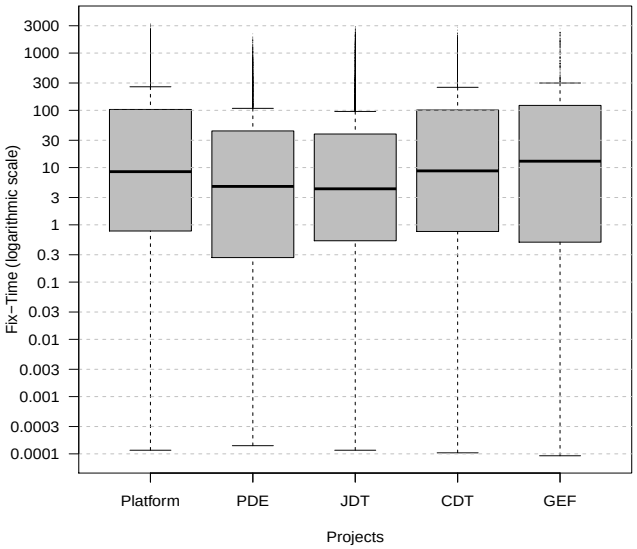
Furthermore, we also notice from Figure 1 another interesting fact: many bugs take a very long time to fix, e.g.: more than 100 days. However, it is not clear what causes these bugs to take this long to fix. From our discussions with the Mozilla developers, it was stated that many bugs are reported with incorrect information, more particularly, many bugs are reported specifying the wrong component of the software system. In Firefox for example, many bugs are reported with the *General Component* that rather belong to the *UI* component. This way, the developers loose the attention of these bugs with incorrect information and are thus not dealt with. However, the same bugs would be fixed much faster when the correct information would be provided.

In this study, we focus on the bug reports with conspicuous fast fix-times and propose to filter out these bug reports. These bugs are fixed in a different manner than most other reported bugs which tend to have a more "normal" lifecycle. In this study, we propose to use a threshold to filter out bug reports with fix-times below that particular threshold. We determine the threshold through the distribution analysis of the fix-times from Figure 1. In Section III, we choose a particular threshold for our experiment. This threshold allows us to distinguish "conspicuous" bugs from the "normal" bugs.

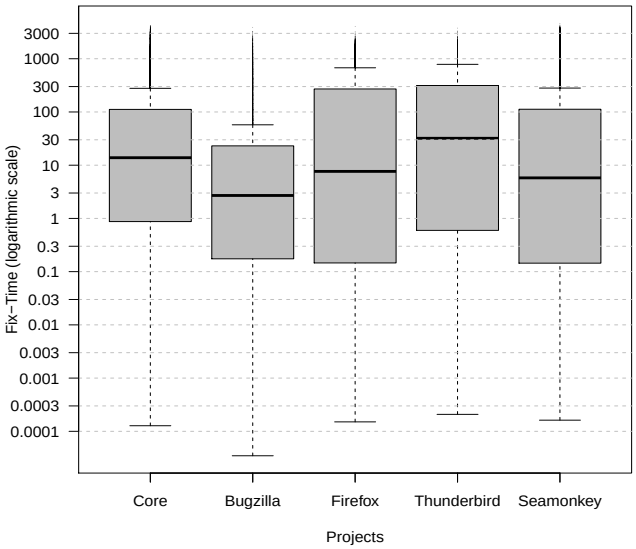
III. Experiment

In this section, we first provide the different steps involved in setting up the experiment. Here, we wish to evaluate the impact of introducing a filtering step for bug reports with fix-times below a certain threshold. Subsequently, we present the

Fig. 1: Box-plots of the Fix-Times expressed in Days



(a) Box-plots of the Fix-Times for Eclipse expressed in Days



(b) Box-plots of the Fix-Times for Mozilla expressed in Days

results achieved through the experiment where we compare the accuracy before and after the filtering step.

A. Setup Experiment

From our previous analysis in Section II, we have noticed that many reported bugs are fixed in just one day or even within a few minutes. As we previously observed in Section II-C, the process of fixing these bugs differs from how most other reported bugs are fixed. Therefore, we propose to make a distinction between these reports. Hence, we introduce a filtering step by excluding reports with conspicuous fast fix-times. We perform an experiment where we predict the fix-time of a reported bug. Our goal is to compare the accuracy of the predictions before and after introducing the filtering of bug reports. A threshold for the fix-time is used to distinguish the conspicuous reports. We experimented with thresholds ranging from a few minutes up to one day. When we use a low threshold, we hardly notice the impact on the accuracy while higher values for the threshold tend to have a higher impact. For the purpose of this study, we set the threshold for each software system separately to halfway the lower quartile of the box-plots from Figure 1, i.e.: for each software system, the threshold is set to $1/2 * qu_1$.

Since we would like to investigate the impact of the filtering step, we run the experiment in two phases: with and without filtering out the bug reports. Reports with fix-times below a threshold are not considered in the second phase. This way, we are able to compare the accuracy before and after the filtering. The outcome of these two steps are presented and discussed in III-B. Next, we provide more details about the set-up of the experiment.

First, we group the bug reports according to the actual fix-time into two categories:*Fast* and *Slow*. A similar grouping was proposed by Giger et al. [3], where the median value of the fix-times of all bug reports is used to distinguish the two categories as follows:

$$bugClass = \begin{cases} Fast & : \quad fixtime <= median \\ Slow & : \quad fixtime > median \end{cases}$$

A bug report is now categorized according to the median values we presented in Table II. The *bugClass* is the dependent variable since we are interested in predicting this field. Now, we can use the technique of classification to predict the *bugClass* of a particular bug. During the training phase, the classifier learns the characteristics of the fix-time by analyzing a history of bug reports where the fix-time is known in advance. This history of reports is also known as the *training set*. A separate *evaluation set* of reports is then used to evaluate the accuracy of the classifier. The classifier analyzes and learns the characteristics from a set of independent variables that we have extracted from the collection. The independent variables of a bug report are listed in Table III.

Many different classification algorithms exist. In this study, we use the popular Naïve Bayes classifier which found its way into many applications due to its simple principle but yet powerful accuracy [7]. Bayesian classifiers are based on a statistical principle. While training the classifier, each independent variable is assigned a probability that it belongs

TABLE IV: The different attributes extracted from the bug reports.

(a) Accuracy Results of Eclipse Projects

Project	AUC Before	AUC After
Eclipse Platform	0.692	0.700
Eclipse PDE	0.641	0.661
Eclipse JDT	0.646	0.649
Eclipse CDT	0.693	0.708
Eclipse GEF	0.663	0.732

(b) Accuracy Results of Mozilla Projects

Project	AUC Before	AUC After
Mozilla Core	0.663	0.686
Mozilla Bugzilla	0.722	0.733
Mozilla Firefox	0.623	0.653
Mozilla Thunderbird	0.657	*0.645*
Mozilla Seamonkey	0.698	0.706

TABLE III: The different attributes extracted from the bug reports.

Attribute	Description
day_opened	day in which the bug was reported
month_opened	month in which the bug was reported
year_opened	year in which the bug was reported
platform	hardware platform, e.g., PC, Mac
op_sys	the operating system on which the bug occurred
assigned_to	the developer to which the bug was assigned
reporter	the person who reported the bug
severity	the severity of the bug
priority	the priority of the bug
component	the component affected by the bug
fixtime	number of hours needed to fix the bug

to a certain *bugClass*. The *bugClass* of a new report is then predicted according to the independent variables appearing in the new report and their respective assigned probabilities.

In order to evaluate the accuracy of the predictions, we use *K-Fold Cross-validation*. Here, the available set of bug reports is split into *K* disjoint subsets. The classifier is then trained with *K - 1* of the subsets and subsequently evaluated with the last subset. These steps are repeated *K* times so that each single subset has been used to evaluate the predictions. In this study, we use *10-Fold Cross-validation*. Furthermore, the *Receiver Operating Characteristic* with the Area Under Curve (AUC) statistic to measure the accuracy of the predictions. If the *Area Under Curve* (AUC) is close to 0.5 then the classifier provides practically random predictions, whereas a number close to 1.0 means that the classifier predicts with the highest accuracy.

In this study, we carry out our experiment using the popular WEKA[3] tool implementing many classification algorithms including a Naïve Bayes classifier. Furthermore, WEKA implements *K-Fold Cross-validation* and also the *Receiver Operating Characteristic* with the *Area Under Curve* accuracy measure.

B. Results

Next, we present the results obtained from the experiment where we show the accuracy in terms of the AUC measure before and after we introduce the filtering step. Table IV summarizes these results for the different software systems selected from both Eclipse and Mozilla.

[3]WEKA: http://www.cs.waikato.ac.nz/ml/weka/

In case of the initial experiment where we do not apply any filtering on the bug reports, we notice from Table IV that the Area Under Curve accuracy measure ranges between 0.641-0.693 and 0.623-0.722 for Eclipse and Mozilla respectively. However, when we introduce the filtering step where bug reports are removed when the fix-time is below a threshold we determined previously, we observe that the accuracy of the predictions tend to change compared with no filtering. Here, the AUC measure ranges between 0.649-0.732 and 0.653-0.733 for Eclipse and Mozilla respectively. When we compare the AUC of before and after the filtering step, we notice a improvement of the accuracy of the predictions except in the case of *Mozilla Thunderbird* with a small degradation of the accuracy. However, in some cases like *Eclipse PDE*, *Eclipse GEF*, *Mozilla Core* and *Mozilla Firefox*, the improvements of the accuracy of the predictions are considerable.

This improvement of the accuracy indicates that the accuracy of the predictions improves when we filter out bug reports with conspicuous fix-times. These conspicuous reports are handled by the developers in a different manner resulting in different characteristics. The classifier might get "confused" when we do not distinguish conspicuous bug reports from other reports which would result in less accurate predictions.

IV. Conclusion and Future Work

Several studies use data mining techniques to predict the fix-time of a particular bug. However, real-world data often contain outliers which may "confuse" a data mining techniques. Therefore, in this paper, we proposed to investigate reported bugs and their corresponding fix-times. Here, we have observed that, for the cases of both Eclipse and Mozilla, a fraction of the bug reports indicate conspicuous fix-times where the bugs are often fixed within a few minutes. This can be explained by the fact that, in some cases, developers first fix a particular bug they have encountered and then afterwards file a report in the bug tracking system.

According to our observations, we proposed to filter out these conspicuous bug reports when we are using data mining techniques to predict the fix-times of reported bugs. We performed a small data mining experiment where we compared the accuracy of predictions of the fix-time before and after filtering the conspicuous reports. This experiment demonstrated that the filtering step allowed more accurate predictions of the

fix-time. Therefore, we conclude that more filtering of bug reports can have a positive impact when we try to predict the fix-time of a reported bug.

On-going and future work includes a more profound analysis of the characteristics of fix-times. This analysis can take other factors into account, e.g.: profile of developer, upcoming releases, severity of bugs, changes applied on the bug reports. The relation of these additional factors to the fix-times can be studied. Furthermore, more advanced outlier detection techniques can be applied on the bug data to identify conspicuous reports. Outlier detection techniques like *Chauvenet's Criterion* and *Grubbs' Test* can be applied on the bug reports. This way, it is possible to experiment with other outlier removal techniques to investigate the impact of the different outlier detection techniques on the accuracy of the predictions.

ACKNOWLEDGMENTS

This work has been carried out in the context of a Ph.D grant of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen). Additional sponsoring by (i) the Interuniversity Attraction Poles Programme - Belgian State – Belgian Science Policy, project *MoVES*;

REFERENCES

[1] L. D. Panjer, "Predicting eclipse bug lifetimes," in *Working Conference on Mining Software Repositories (MSR)*, 2007, p. 29.

[2] G. Bougie, C. Treude, D. M. Germán, and M.-A. D. Storey, "A comparative exploration of freebsd bug lifetimes," in *MSR*, 2010, pp. 106–109.

[3] 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*, ser. RSSE '10, 2010.

[4] J. I. Maletic and A. Marcus, "Data cleansing: Beyond integrity analysis," in *Fifth Conference on Information Quality (IQ 2000)*, 2000, pp. 200–209.

[5] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. T. Devanbu, "Fair and balanced?: bias in bug-fix datasets," in *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2009, Amsterdam, The Netherlands, August 24-28, 2009*, 2009, pp. 121–130.

[6] A. Hindle, D. M. Germán, and R. C. Holt, "What do large commits tell us?: a taxonomical study of large commits," in *Proceedings of the 2008 International Working Conference on Mining Software Repositories, MSR 2008, Leipzig, Germany, May 10-11, 2008, Proceedings*, 2008, pp. 99–108.

[7] I. Rish, "An empirical study of the naïve bayes classifier," in *Workshop on Empirical Methods in AI*, 2001.