The Journal of Systems and Software 136 (2018) 173–186

Contents lists available at ScienceDirect

The Journal of Systems and Software

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

Predicting bug-fixing time: A replication study using an open source software project☆

Shirin Akbarinasaji*, Bora Caglayan, Ayse Bener

Data Science Lab, Ryerson University, Toronto, Canada

ARTICLE INFO

Article history:
Received 2 January 2016
Revised 9 February 2017
Accepted 17 February 2017
Available online 24 February 2017

Keywords:
Replication study
Bug fixing time
Effort estimation
Software maintainability
Deferred bugs

ABSTRACT

Background: On projects with tight schedules and limited budgets, it may not be possible to resolve all known bugs before the next release. Estimates of the time required to fix known bugs (the "bug fixing time") would assist managers in allocating bug fixing resources when faced with a high volume of bug reports.

Aim: In this work, we aim to replicate a model for predicting bug fixing time with open source data from Bugzilla Firefox.

Method: To perform the replication study, we follow the replication guidelines put forth by Carver [J. C. Carver, Towards reporting guidelines for experimental replications: a proposal, in: 1st International Workshop on Replication in Empirical Software Engineering, 2010.]. Similar to the original study, we apply a Markov-based model to predict the number of bugs that can be fixed monthly. In addition, we employ Monte-Carlo simulation to predict the total fixing time for a given number of bugs. We then use the k-nearest neighbors algorithm to classify fixing times into slow and fast.

Result: The results of the replicated study on Firefox are consistent with those of the original study. The results show that there are similarities in the bug handling behaviour of both systems.

Conclusion: We conclude that the model that estimates the bug fixing time is robust enough to be generalized, and we can rely on this model for our future research.

1. Introduction

Software bugs are an inextricable part of software maintenance and development activities, and they may have continuous negative effects such as user inconvenience, inappropriate functionality, and security risk (Wikipedia, 2015b). A significant amount of time is spent by software developers in investigating and fixing bug reports. However, there is no consensus among the experts on how much time is spent on finding and fixing bugs. Different researchers estimate the testing effort between 45% and 80% of the overall effort (Kaner et al., 1999; Boehm et al., 1981; Zhang et al., 2012): Kaner et al. provided a figure of 45% of project time spent on testing (Kaner et al., 1999) and Bohem et al. assigned 50% of the effort to parameter and assembly testing (Boehm et al., 1981). However, Zhang et al. reported 80% of time spent on fixing and finding bugs (Zhang et al., 2012). The reason for such disagreement may be due to differences in the definition of the testing processes. In some cases, bug fixing and reporting is part of the testing process, and in others it may be after the testing phase. Therefore, depending on how each phase is defined, and which particular activities are included or excluded, the reported effort percentage may differ. Boehm et al. in their study indicated that the companies they analyzed initially spent more time on development, while they focused on fixing bugs at the post-release maintenance stage (Boehm and Basili, 2005). They also reported that noncritical software defects may not be as costly as major defects, however, there should be an emphasis on getting things 'right' sooner rather than later (Boehm and Basili, 2005). Zhang et al. also indicated that as software systems become more complex and they are built on multiple components and platforms, software companies receive more bugs over a period of time (Zhang et al., 2013). They argued that development teams may have to defer fixing bugs to the later releases due to reporting more bugs by users, and tight release schedule and budget. Therefore, estimating bug fixing time would aid software development managers to manage high volumes of bug reports in an issue tracking system by balancing soft-

☆ Fully documented templates are available in the elsarticle package on CTAN.

* Corresponding author.
E-mail addresses: shirin.akbarinasaji@ryerson.ca (S. Akbarinasaji), bora.caglayan@ryerson.ca (B. Caglayan), ayse.bener@ryerson.ca (A. Bener).

http://dx.doi.org/10.1016/j.jss.2017.02.021
0164-1212/

ware quality and bug fixing effort (Zhang et al., 2013; Akbarinasaji, 2015; Tom et al., 2013).

In handling bugs, software development teams decide to fix the bug in the current release or defer it to later releases. Such decisions are made by considering a set of factors including the risk of defects, their severity, customer pressure, and the effort required to fix the bugs (Snipes et al., 2012). Hence, software development managers need to make trade-off decisions considering project constraints such as schedule and resources to fix bugs now or delay bug fixing to the next release. The motivation behind the original study and our replication of their study is that having advance knowledge of bug fixing time would allow software quality team to prioritize their work (Hooimeijer and Weimer, 2007; Kim and Whitehead, 2006).

Recently, the analysis and prediction of bug fixing time has attracted attention from many researchers (Bhattacharya and Neamtiu, 2011; Zhang et al., 2013; Giger et al., 2010; Lamkanfi and Demeyer, 2012). Bhattacharya et al. reported the extraction of bug report attributes such as bug severity, bug priority, bug owner, number of developers, and number of comments to estimate the bug fixing time for both open source and commercial software projects (Bhattacharya and Neamtiu, 2011). We are interested in investigating bug fixing time to develop a model to manage deferred bugs in an issue tracking system. Therefore, in this paper, we replicated a study by Zhang et al. (2013) to validate the generalizability of their work.

We believe that the results of our replication study would be useful to both practitioners and researchers in two ways. First, we will verify if the results of the original study are valid and reliable on other software bug tracking systems. Because most research papers pertaining to bug fixing time estimation are isolated ones published by a particular research group or a laboratory, there is a necessity to generalize the model using a new dataset with totally different settings. In addition, the replication study would shed light on the ambiguous definition of the associated research theory and a few details about the original study that the authors were not able to report due to limitations of their study or confidentiality of the dataset (Caglayan et al., 2015). Additionally, the replication study would enrich the body of software engineering knowledge and aid examination of the novel approach in different contexts (Caglayan et al., 2015).

Second, the replication study will help gain research insight from the viewpoints of building upon new hypotheses regarding the effect of deferred bugs and testing, verifying/refuting said hypotheses to build new theories. We chose Zhang et al.'s research because their bug handling model categorizes the bugs into deferred and regular bugs, which makes their work unique for the purpose of our own future research motivation. In the original study, the authors proposed three research questions:

- How many bugs can be fixed in a given time?
- How long would it take to fix a given number of bugs?
- How long would it take to fix an individual bug?

We use the same approach to analyze data from Bugzilla Firefox. Then, we compare our findings with those of the original study to examine the external validity of their findings.

This paper is structured based on the replication study framework suggested by Carver (2010). In the related work section, we review the importance of replication studies in software engineering and provide an overview of fixing time prediction. In Section 3, we review the description of the original study, including the research questions, data collection procedures, and approach employed. In Section 4, we describe replication including the level of interaction with the original study and changes compared to the original study. Then, we compare our findings with those of the original study. Finally, we describe the threats to study validity from the replication viewpoint and conclude the paper with a summary of our findings and future work.

2. Related work

As this research is the replication study on predicting the bug fixing time, we reviewed the literature both on the value of replication studies in software engineering and also on bug fixing time prediction and estimation.

2.1. On the value of replication studies in software engineering

Replication is an integral part of software engineering (SE) experimentation to enrich the body of knowledge and to find the conditions that make an experiment steady (de Magalhães et al., 2015). La Sorte (1972) defined replication as *"a conscious and systematic repeat of an original study."* Juristo and Gómez (2012) identified six purposes of replication: control for sampling error, control for experiments, control for site, control for an artifactual result, determining operationalization limits, and determining the limits of population properties. They studied the concept of replication from both the statistical and the scientific perspectives, and identified several topologies for replication in software engineering. In this study, we did a replication "to control for experiments" as per Juristo's categorization. We verified that the different experiments did not influence the result. Shull et al. (2002) emphasized that the availability of documented experimental packages, which collect information on experiments, make replication more persuasive.

Brooks et al. (1996) classified replication into internal replication and external replication. Internal replication is performed by the primary researcher, whereas external replication is undertaken by independent researchers. They described the importance of external replication and presented a concrete example of it. In 2008, Shull et al. (2008) discussed successful replication studies and suggested that both replication with similar results and different results can be considered successful. In addition, they classified exact replication into dependent replication and independent replication. Exact replication is a group of experiments that follows the original study to the extent possible. Dependent studies are replication studies in which researchers attempt to retain all criteria of the primary experiment. However, independent replication studies proceed by changing one or more aspects of the original studies. Basili et al. (1999) discussed the necessity of performing replication studies. They categorized families of replicated studies into five groups: strict replication or replication that conforms exactly to the original study; replication in which the dependent variables of the original study are changed; replication in which the independent variables of the original study are changed and the focus is on evaluation; replications in which the contextual variables in the environment in which the original study was performed are changed; and replications in which the hypotheses of the original study are tested unchanged in order to extend the theory. Based on Basili et al.'s categorization, our study is an external non-strict replication. In 2007, Sjoberg et al. (2007) outlined the future of empirical methods in software engineering research and recommended that to avoid bias, it would be better if replication is performed by external researchers.

The first replication study in software engineering was performed in 1994 by Daly et al. on the verification of results in software maintenance (Daly et al., 1994). In 2014, Da Silva et al. (2014) presented a mapping study on the replication of empirical studies related to software engineering. They analyzed 37 papers to explore the classification schemes and guidelines for replication work in software engineering research. In another study, Carver (2010) published replication guidelines for reporting the details of

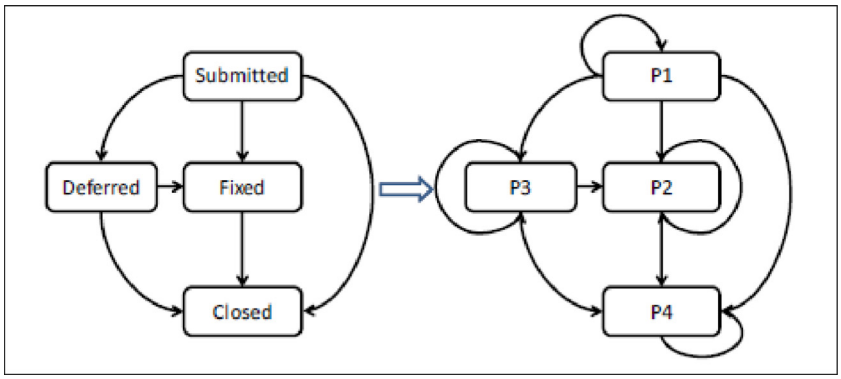


Fig. 1. Markov model (Zhang et al., 2013).

the original study and a comparison of the findings. In this study, we follow the same guidelines because they allow us to present the similarities and differences between the original study and the replicated one.

2.2. Bug fixing time prediction and estimation

The literature for predicting bug fixing time can be classified into two categories: studies that propose models for predicting the overall time required for fixing the bugs via classification and regression and studies focused on reducing the bug fixing time in initial bug triage processes by recommending ideal developers for bugs. In this study, we focus on the former type of studies.

Panjer (2007) employed Eclipse Bugzilla data to investigate the viability of using data mining tools to predict bug fixing time. He applied basic machine learning tools such as 0-R, 1-R, decision tree, naive Bayes, and logistic Regression in the initial stage of the bug life cycle. He reported that his model is able to predict 34.9% of the bugs correctly. Hooimeijer and Weimer (2007) employed linear regression to predict whether a bug report will be addressed by developers within a given time threshold or not. They found the minimum amount of necessary post-submission data for the model to improve model performance. In 2009, Anbalagan et al. studied 72,482 bug reports from Ubuntu to investigate the relationship between bug fixing time and the number of participants involved in bug fixing. They concluded that there is a strong linear relationship between the number of participants and the bug fixing time and that a linear model can be used for predicting fixing time. Giger et al. (2010) proposed a decision tree to classify bugs into “fast” or “slow”. They conducted experimental studies on bug reports from six open source projects by Eclipse, Mozilla, and Gnome. In addition, they showed that model performance can improve if the post-submission bug report data of up to one month is included in the features. Lamkanfi and Demeyer (2012) proposed the filtering of bug reports for fixing time analysis. They reported a slight improvement in fixing time prediction by filtering out conspicuous bugs (bugs with a very short lifecycle). In 2015, Habayeb (2015) employed a hidden Markov model for predicting bug fixing time based on the temporal sequence of developer activities. Habayeb performed an experiment on Firefox projects and compared her model with popular classification algorithms to show that the model outperformed existing ones significantly (Habayeb et al., 2015). In 2013, Zhang et al. (2013) performed an empirical study on bug fixing times in three projects of a company called CA Technologies. They proposed a Markov chain model to predict number of bugs that can be fixed in the future. In addition, they employed Monte Carlo simulation to predict the total fixing time for a given number of bugs. Then, they classified bugs as fast and slow based on different threshold times. In this paper, we replicate their work in order to gain insight for the purpose of our future research about predicting the bug fixing time in order to manage defect debt (i.e. lingering/ deferred defects in the system). In Section 4.1, we explain thoroughly why we chose Zhang et al.’s study.

3. Information about original study

In this section, we briefly summarize the original study used herein as the basis of our replication. The original study aimed to provide a better estimate of bug fixing efforts to manage software projects more effectively. It was conducted by Zhang et al. (2013) and three research questions were considered:

- How many bugs can be fixed in a given amount of time?
- How long would it take to fix a given number of bugs?
- How much time is required to fix a particular bug?

They conducted the experimental study on three commercial software projects by CA Technologies. CA Technologies is an IT management solutions company. To address their research questions, Zhang et al. (2013) defined a simplified bug handling process. Initially, bugs are identified and submitted into a bug tracking system either by the QA team or by end users. Once the bugs are confirmed, they are assigned to the developer team for resolution. They also verified whether the bugs were prioritized correctly. If the developer team was unable to resolve the bugs in a current release, the bugs were tagged as “Deferred”. Finally, if the developer team was able to fix the bugs and the QA team was satisfied with the resolution, the bugs were tagged as “Closed”. Fig. 1 (left) shows the simplified life cycle of bugs in CA Technologies’ system.

Owing to the limited availability of resources and time, the developer team may not fix all bugs upon their arrival. Hence, as an initial analysis, Zhang et al. explored the distribution of bug fixing time for those projects and performed some exploratory analysis on the fixing times of the bugs based on their severity and priority.

3.1. How many bugs can be fixed in a given amount of time?

To predict how many bugs can be fixed within a given time in the future, Zhang et al. proposed a Markov-based model. They extracted the resolution history of bugs from the company’s historical bug database to calculate a state-transition matrix. The transition matrix, which is shown below, yielded the probability distribution of bugs transitioning from one state to another. Fig. 1 shows the

mapping of the bug lifecycle model to the Markov chain model:

$$P = \begin{bmatrix} P_{11} & P_{12} & P_{13} & P_{14} \\ P_{21} & \cdots & \cdots & P_{24} \\ P_{31} & \cdots & \cdots & P_{34} \\ P_{41} & P_{42} & P_{43} & P_{44} \end{bmatrix} \quad (1)$$

In the transition matrix, P_{ij} is the probability that a bug moves from state i to state j. The distribution of bugs at time t is defined as the percentage of bugs in each state at time t ,α_t, and it is calculated as follows (Zhang et al., 2013):

$$\alpha_t = \alpha_0 * P^t \quad (2)$$

Note that the distribution of the bugs at time 0, α_0, is equivalent to the percentage of bugs in the initial state. Suppose that the total number of bugs at time t, total number of fixed bugs at time t, and the total number of bugs submitted are T_t ,N_t, and S_t, respectively. Then, the total number of bugs fixed at time $t+1$ is (Zhang et al., 2013)

$$N_{t+1} = N_t + (P_{12} + P_{13} * P_{32}) * (S_t + \Delta S_{t+1}) \quad (3)$$

where

$$S_t = T_t * \alpha_t^{submitted} = T_t * \alpha_0^{submitted} P^t \quad (4)$$

$$\Delta S_{t+1} = T_{t+1} - T_t \quad (5)$$

The total number of bugs at time t + 1 was predicted using auto regression models and historical data of the total number of bugs, $(T_{t+1} = \alpha * T_t + \beta)$. Note that Markov decision model is flexible to be trained based on any criteria for t. Based on the availability of data for calculating the transition matrix t can be either similar to or greater than or less than the release cycle. The result of this study was compared with the results of a simple average-based model for predicting the number of bugs in the next M months as follows (Zhang et al., 2013)

$$S_{t+M} = S_t + M * \overline{S} \quad (6)$$

where $\overline{S}$ is the average number of bugs fixed in each month. To evaluate the performance of their Markov-based model, Zhang et al. calculated MRE (magnitude relative error).

3.2. How long would it take to fix a given number of bugs?

Regarding the second research question, Zhang et al. (2013) proposed using Monte Carlo simulation. Originally, the best fitting distribution for bug fixing was estimated. Because the fixing time distributions of the CA Technologies projects were skewed, the authors employed lognormal, exponential, and Weibull distributions. To evaluate the best distribution function, the goodness of fit, R^2, and standard error of estimate, S_e, were monitored (Zhang et al., 2013):

$$S_e = \sqrt{\frac{\sum (y - y')^2}{n-2}} \quad (7)$$

where y' and y represent the predicted and actual fixing times, respectively, and n is the number of observations in the dataset.

After determining the best-fitting distribution, they randomly generated N fixing times from each distribution using parameters acquired from historical data. Each value corresponded to a bug fixing time. Then, these N values were summed and saved as the first run of the simulation. To predict the total bug fixing time, multiple iterations of the simulation were performed, and the average of the results of these iterations was taken as the final estimate. The result of their experiment was compared with that of an average-based model as follows (Zhang et al., 2013)

$$Y = N * \overline{X} \quad (8)$$

where $\overline{X}$ is arithmetic average of bug fixing time.

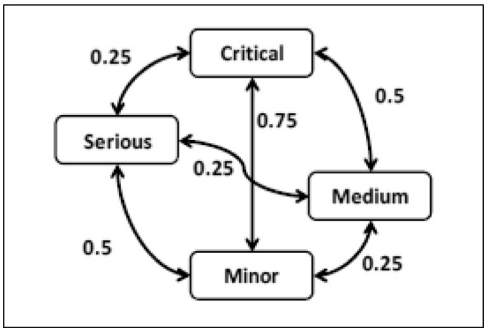


Fig. 2. Measuring severity distance (Zhang et al., 2013).

3.3. How much time is required to fix a particular bug?

The authors proposed the K-nearest neighbourhood classification to predict the bug fixing time for each bug. They assumed that similar bugs would behave similarly. They classified the bugs into quick and slow bugs under six different time thresholds (0.1, 0.2, 0.4, 1, and 2 time units). Each time unit was almost equivalent to the median time taken for fixing the bugs. Based on whether the fixing time for a given bug was less than or more than the threshold, it was labelled as a "quick" or a "slow" bug, respectively. Zhang et al. collected the following information for their KNN model:

1. Submitter of bugs
2. Owner of bugs or developer to whom a bug is assigned
3. Bug priority (critical, serious, medium, minor)
4. Bug severity (blocking, functional, enhancement, cosmetic, or request)
5. ESC is an indicator of whether a bug was discovered internally by the QA team or externally by end users.
6. Bug category
7. Bug summary

For measuring the distance between bug priorities, they defined a distance matrix based on Fig. 2. It was based on the assumption that the closer the levels of bugs, the smaller are the distances between the bugs' priorities. As can be seen in Fig. 2, the distance between critical and serious bugs is 0.25, and the distance between critical and minor bugs is 0.75. The severity distance matrix was also calculated in the same manner.

The distance between bug summaries was measured based on the bag of words model (Ko, 2012) and how different two sets of words were. They generated a standard set of words for each bug category and used it to eliminate the useless set of words from the summary. The distance was calculated by counting the bugs from a summary containing the standard words (Zhang et al., 2013).

$$d^s(\alpha, \beta) = 1 - \frac{\left|W(S_\alpha) \cap W(S_\beta) \cap W(C)\right|}{\left|W(S_\alpha) \cup W(S_\beta) \cap W(C)\right| + 1} \quad (9)$$

where S_α, S_β stands for summary of bug α, β and $W(S_\alpha)$ is the set of words extracted from the summary of bug α. $W(C)$ represents the set of standard words extracted from pre-defined category labels.

For other categorical features, the distance is as follows (Zhang et al., 2013):

$$d(v_1, v_2) = \begin{cases} 1 & v_1 \neq v_2 \\ 0 & v_1 = v_2 \end{cases} \quad (10)$$

The Euclidean distance was calculated to measure the distance between two bug summary reports and all other features, and KNN classification was applied to find the nearest neighbourhood of each particular bug. The proposed KNN scheme was compared

with existing methods in the literature, including decision tree, naive Bayes, Bayesian network, and RBF (radial basis function) network. To evaluate the performance of this method, the weighted F-measure was used as below (Zhang et al., 2013):

$$F = \frac{1}{\sum N_{c_i}} \cdot \sum F_{C_i}.N_{c_i} \quad (11)$$

where N_{c_i} denotes the size of each class in the dataset, and F_{C_i} is the associated F measure for the class.

4. Information about replication

4.1. Motivation for replication

A huge volume of bug reports in the issue tracking system compared to the available resources may cause a backlog of defects in the bug repository. Delaying bug fixing has a tremendous effect on software quality (Tom et al., 2013). A massive pileup of defects in the bug repository might cause the development team to spend all of their time fixing the deferred bugs, leaving no time to address the delivery of new features (Tate, 2005). Therefore, there is a necessity to prioritize defect reports and decide when to implement bug fixing activities. The key factor for scheduling the order of defect reports is a quantitative understanding of the time constraints for fixing the bugs and the effect of postponing bug fixing activities on the bug resolution time. Instead of starting afresh, we relied on a fixing time prediction model from the literature. We chose Zhang et al.'s study mainly because their bug handling model (Fig. 1) considers the "Deferred" state. Additionally, their research questions can serve as a preliminary analysis for our future research goal of managing deferred bugs. Therefore, the purpose of this replication is twofolds: first, it provides insight for our academic research program; second, it verifies the merits of the original study on an open source data.

4.2. Level of interaction with original researchers

According to Carver (2010) replication guidelines, the level of interaction between the researchers of the primary and the original studies should be reported. In this study, we had a personal communication with the main author of the original paper and sent him several emails to discuss details that were not reported in the original paper. So, the level of interaction in this study was more than only reading a paper and gathering information and it was closer to substantial (the researchers of the original study acted as consultants). However, based on the nature of open source bug tracking systems and differences in the levels and details of information in our dataset, we made some changes to the original study context. We discuss the changes to original study in the next section .

4.3. Changes to original study

In this section, we explain the deviation of our research from the original study due to nature of open source data:

- Different Dataset:
  The experiment of this study was performed on the Mozilla Firefox project. Firefox is a free and open source web browser created by the Mozilla community. The browser is available for all modern operating systems including Windows, OS, and Linux (Wikipedia, 2015a). It is among the most popular web browsers in the world because of its security, speed, and add-ons. The bug tracking system for Mozilla, called Bugzilla, was developed by Mozilla projects for open source web browsers. Originally, Zhang et al. performed an empirical study on three projects from CA Technologies. Therefore, the first difference is associated with the differences between open source and commercial bug tracking systems. Because commercial software guarantees specific support and complete customization to customer needs, this might affect the characteristics of the dataset. Another difference pertains to the purpose and the users of the projects. Firefox is a web browser for transferring, retrieving, and presenting information to and from the World Wide Web. The CA Technologies projects are related to IT management and solutions such as API management, automation, and project and portfolio management. Therefore, the nature of defects reported in the bug tracking system and the prioritization policy for fixing the defects may be completely different.
  One other difference in the datasets is related to the time frame and the number of the bugs extracted. We analyzed the Firefox bugs created over a period of 15 years from January 1, 2000 to November 30, 2015. Our database comprises 158,831 distinct bug IDs in total. In order to extract Firefox bug information from Bugzilla, we developed a python program to use REST API and restore the data into a relational database management system. Bugzilla was released and used for Mozilla project as a bug tracker since 1998. More information about the bug tracking system can be found in Bugzilla @ Mozilla[1] and the latest REST API is well documented in Bugzilla documentation.[2] In the primary study, the authors extracted bugs for projects A, B, C over 45 months, 30 months, and 29 months, respectively. However, owing to confidentiality, they could not report the total number of bugs and any statistics about them. They also reported having limited information on data as a threat to the validity of their study.
- Different Bug Handling Models and Maintenance Processes:
  The maintenance processes for all bugs in the Bugzilla database may not be similar. However, for most bugs, once reported in the system, their status changes to "UNCONFIRMED." The bugs remain unconfirmed until they receive sufficient votes or the developers are able to reproduce the bug; then, the status of the bug changes to "NEW." Furthermore, a bug may start its journey as "NEW" if the reporter has the authority to confirm it upon arrival. Once bug status is confirmed in the tracking system, the bug may either be assigned to specific developers or it may become available publicly to volunteer developers. The bug status changes to "ASSIGNED" after developers take possession of the bug. Then, developers resolve the bug and if the QA team is satisfied by the solution, the bug status changes to "RESOLVED" and "VERIFIED," respectively. Otherwise, the QA might "REOPEN" the bug, and its status would revert to either "ASSIGNED" or "RESOLVED." A bug should be "CLOSED" by the person who reports it. However, as Fig. 3 shows, bugs might bypass some states. For instance, a bug may start as "UNCONFIRMED" and then directly move to "RESOLVED" if a solution is obtained in one step. There is also the possibility that a bug goes backward in the Bugzilla lifecycle; for example, if the ownership of the bug is changed, then the bug status changes from "ASSIGNED" to "NEW" until somebody takes the responsibility to fix the bug. Fig. 3 shows all the possible edges between status nodes in the Bugzilla bug lifecycle (Barnson et al., 2001). Because we would like to gain insight about handling deferred bugs in this study, we affixed the "DEFERRED" status to the Bugzilla lifecycle. Identical to Zhang et al. (2013), a "DEFERRED" bug is one that is not fixed in the current release. Bug may either be assigned to developers but ignored by them or it may not have been assigned to a developer so far. Fig. 4 shows the

[1] https://bugzilla.mozilla.org/
[2] http://bugzilla.readthedocs.io/en/latest/api/

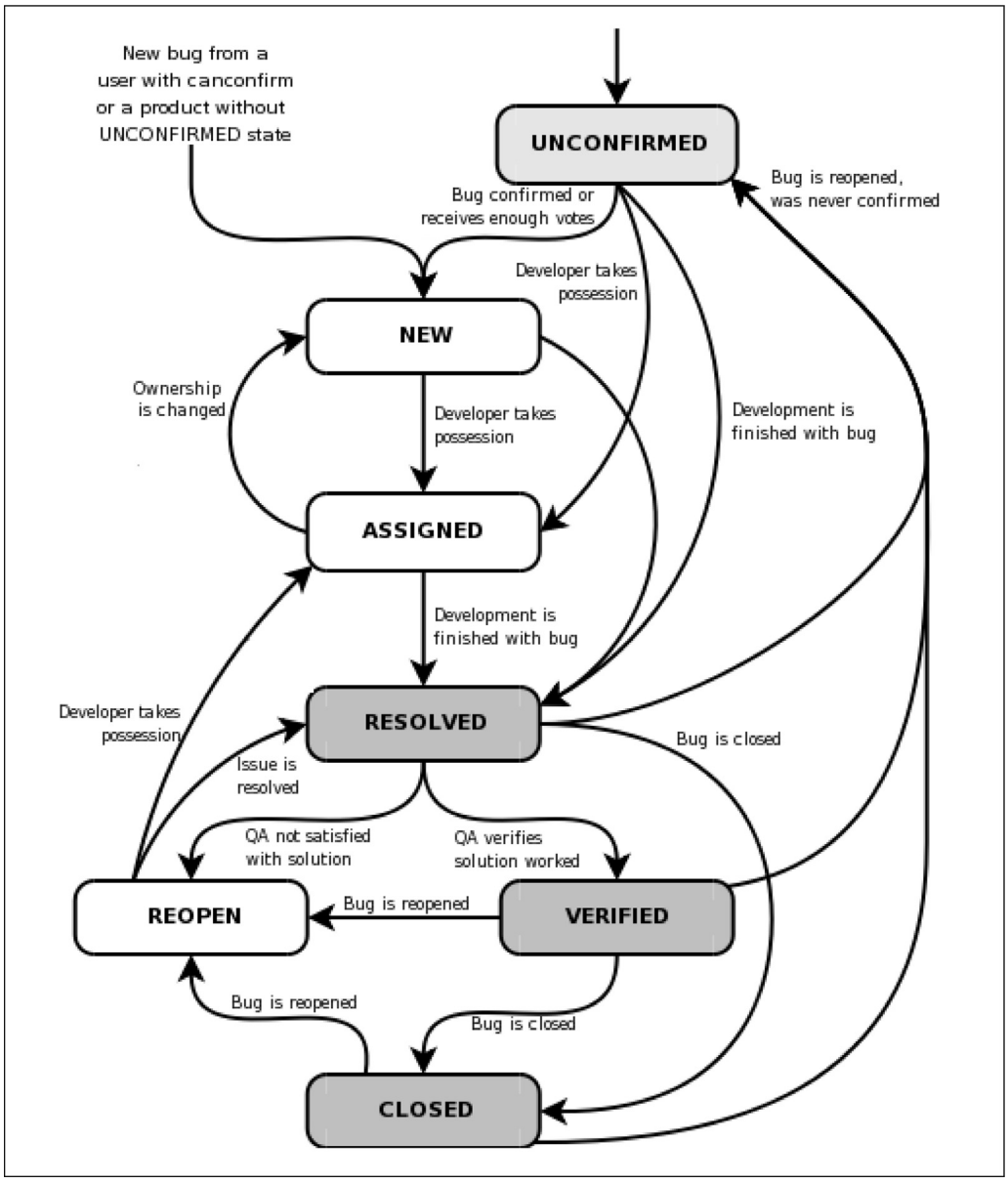


Fig. 3. Bugzilla lifecycle (Barnson et al., 2001).

modified model for the Bugzilla lifecycle. Similar to Zhang et al. (2013), we assumed a bug's next state is dependent only on its current state and Bugzilla defect handling processes can be mapped to a discrete Markov model. Fig. 4 shows the Bugzilla state transition graph.

As described in Section 3, the maintenance processes of CA Technologies are much simpler and are limited to only four states, namely, "Submitted," "Fixed," "Deferred," and "Closed." For Firefox, we defined eight states "Unconfirmed," "New," "Deferred," "Assigned," "Resolved," "Verified," "Reopen," and "Closed." Having different numbers of states would also affect the size of the transition matrix as follows:

$$P = \begin{bmatrix} P_{11} & P_{12} & P_{13} & \cdots & P_{18} \\ P_{21} & P_{22} & \cdots & \cdots & P_{28} \\ P_{31} & P_{32} & \cdots & \cdots & P_{38} \\ \cdots & & & & \\ P_{81} & P_{82} & P_{83} & \cdots & P_{88} \end{bmatrix} \quad (12)$$

where P_{ij} is defined as in the original study as the probability of transfer from state i to state j. Eventually, this difference in the state transition matrix influences the number of bugs that can be fixed at time $t+1$. In the original study, a bug may transfer to "Fixed" from "Submitted" or from "Submitted" to "Deferred" to "Fixed". In the replicated study, based on Fig. 4, a bug may transfer to "RESOLVED" from "UNCONFIRMED," "NEW," or "REOPEN." Therefore, we computed three probabilities initially: probabilities of transfer from "Unconfirmed" to "Resolved," "New" to "Resolved," and "Reopen" to "Resolved":

$$prob_{uncof-res} = P_{12} * P_{23} * P_{35} + P_{14} * P_{45} + P_{13} * P_{35} + P_{15} \quad (13)$$

$$prob_{new-res} = P_{23} * P_{35} + P_{25} \quad (14)$$

$$prob_{reop-res} = P_{73} * P_{35} + P_{74} * P45 + P_{75} \quad (15)$$

Therefore, the total number of "Resolved" bugs at time $t+1$ (N_{t+1}) is given as follows:

$$N_{t+1} = prob_{uncof-res} * (U_t + \delta U_{t+1}) + prob_{new-res} * (Ne_t + \delta Ne_{t+1})$$

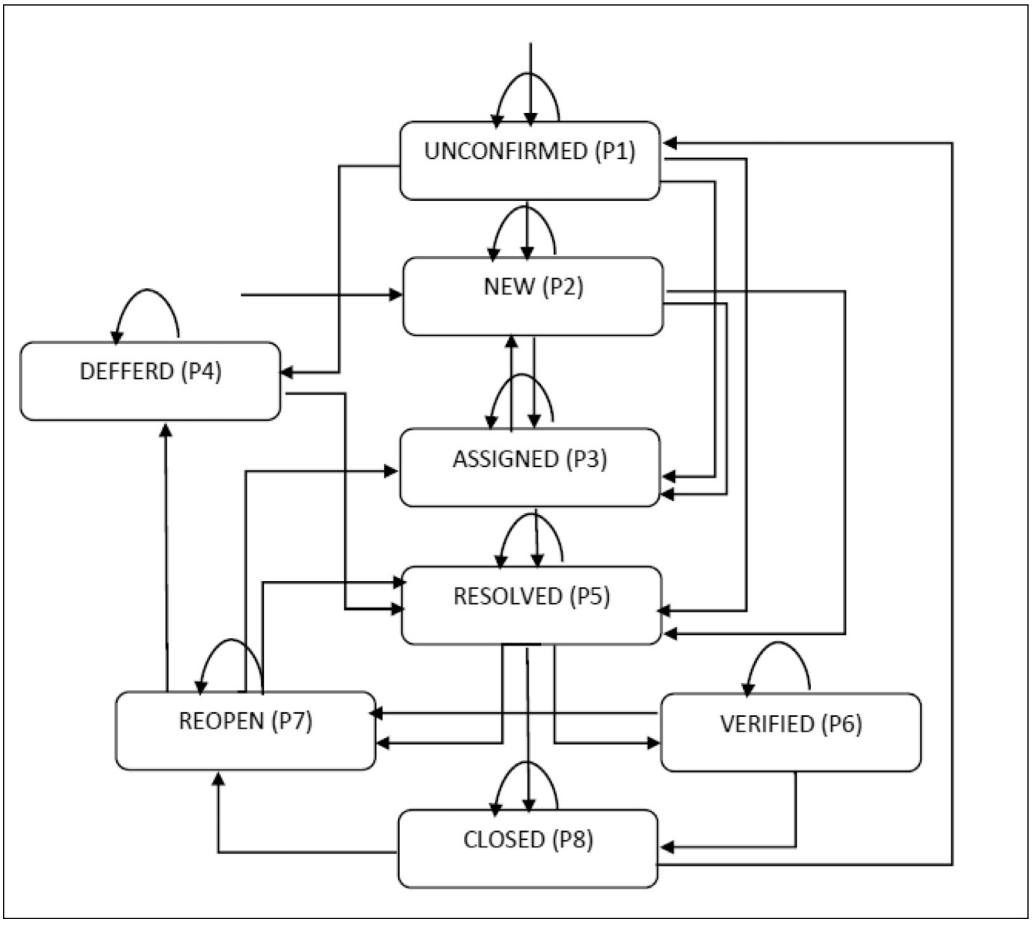


Fig. 4. Markov model for Bugzilla lifecycle.

$$+prob_{reop-res} * (R_t + \delta R_{t+1}) \quad (16)$$

Where U_t, Ne_t, and R_t denote the total numbers of Unconfirmed, New, and Reopen bugs at time t, respectively. $\delta U_{t+1}, \delta Ne_{t+1}$, *and* δR_{t+1} denote the numbers of newly Unconfirmed, New, and Reopen bugs in the Bugzilla tracker system between times t and $t+1$. Note that

$$\delta U_{t+1} = U_{t+1} - Ut \quad (17)$$

$$\delta Ne_{t+1} = Ne_{t+1} - Ne_t \quad (18)$$

$$\delta R_{t+1} = R_{t+1} - R_t \quad (19)$$

where U_{t+1}, Ne_{t+1}, and R_{t+1} are the total numbers of Unconfirmed, New, and Reopen bugs at time $t+1$. Therefore, we can simplify the above formula as below:

$$(U_t + \delta U_{t+1}) = U_{t+1} \quad (20)$$

$$(Ne_t + \delta Ne_{t+1}) = Ne_{t+1} \quad (21)$$

$$(R_t + \delta R_{t+1}) = R_{t+1} \quad (22)$$

Therefore, we can simplify the Eq. (16) as below:

$$N_{t+1} = prob_{uncof-res} * (U_{t+1}) + prob_{new-res} * (Ne_{t+1}) + prob_{reop-res} * (R_{t+1}) \quad (23)$$

Note that U_{t+1}, Ne_{t+1}, and R_{t+1} can be estimated using the following equations:

$$U_{t+1} = T_{t+1} * \alpha_{t+1}^{UNCONFIRMED} = T_{t+1} * \alpha_0^{UNCONFIRMED} * P^{t+1} \quad (24)$$

$$Ne_{t+1} = T_{t+1} * \alpha_{t+1}^{NEW} = T_{t+1} * \alpha_0^{NEW} * P^{t+1} \quad (25)$$

$$R_{t+1} = T_{t+1} * \alpha_{t+1}^{REOPEN} = T_{t+1} * \alpha_0^{REOPEN} * P^{t+1} \quad (26)$$

where T_{t+1} is the total number of bugs at time $t+1$, and α_{t+1} is the percentage of bugs of each state at time $t+1$. To estimate T_{t+1}, we applied regression analysis on historical data. We should mention that our approach is perfectly identical to that of Zhang et al. (2013). Because our Markov model deals with a greater number of states, we require different formulas, but they are in harmony with those in the primary study.

- Different features in KNN classification:
  We were able to extract all the features explained in the original study to construct a KNN classification scheme, with the exception of "ESC" and "Category." We substituted "Category" with "Component." In Bugzilla, Component represents the second-level category after product. For each product, there are predefined components such as developer tools, menus, and web application that describe bug category. However, regarding ESC, we were unable to find any equivalent in the Bugzilla dataset.
  - Measuring Distances of Priority and Severity
    There are seven levels of severity in the Bugzilla dataset, namely, "blocker," "critical," "major," "normal," "minor," "trivial," and "enhancement," and five levels of priorities, namely, "P1," "P2," "P3," "P4," and "P5.""P1" is the highest priority level and "P5" is the lowest priority level. The historical data showed that the mean of bug fixing time for

Table 1
Priorities distance measure.

	P0	P1	P2	P3	P4	P5
P0	0.00	0.17	0.33	0.50	0.67	0.83
P1	0.17	0.00	0.17	0.33	0.50	0.67
P2	0.33	0.17	0.00	0.17	0.33	0.5
P3	0.50	0.33	0.17	0.00	0.17	0.33
P4	0.67	0.50	0.33	0.17	0.00	0.17
P5	0.83	0.67	0.5	0.33	0.17	0.00

the bugs without priority is less than bugs with other priority. Therefore, we decided to make P0 closer to a bug with high priority level other than low priority level. In the same manner as Zhang et al., we defined the distances between severities and priorities, but because our study involves a greater number of levels compared to the primary study, we ended up with different metrics. Our severities and priorities distance measurements are presented in Tables 2 and 1, respectively.

- Measuring Distance in Bug Summary:
  In primary studies, the authors eliminated useless words such as "or," "that," "in," and "etc." by comparing the summary of bugs with the words generated from Category. Given that we replaced "Component" with "Category," this approach for eliminating useless words did not work as intended because the intersection of "Component" with bug summary was nil for most bug IDs in the Bugzilla dataset. Therefore, for each bug ID, we extracted a bag of words for bug summary as follows:
  1. Convert all sets of words in the bug summary to lower case
  2. Remove special characters and numbers
  3. Remove English stop words.
  4. Find the stemming of the text

  Thus, we filtered useless words by removing the stop word from bug summaries and made it easier to find the intersection to measure distance. Thereafter, we used the equation $d^s(\alpha, \beta)$ from the original study to calculate the distance between summaries. Note that W_C was removed from the equation because we were not able to extract any category label from "Component."

5. Comparison of replication results with original results

In this section, we report our findings in comparison to the original study. We evaluated the effectiveness of Zhang et al.'s model in answering the three research questions. Similar to the primary study, we started our analysis by performing exploratory analysis on the distribution of bug fixing times and reporting some statistics regarding bug fixing times according to their severity and priority. Fig. 5 shows the distributions of bug fixing time for the replicated study and the original study (project A). As can be seen in the plot, for both projects, bugs can not be fixed immediately, and some bugs may stay longer in the system. The distribution of bugs in the two studies is skewed, uneven, and long tail. In the replicated study, a day is considered as the time unit, but in the original study, the time unit was normalized for confidentiality.

Table 3
Comparison of bug fixing time (in units) based on severities and priorities–original study (Zhang et al., 2013).

Feature	Value	Max	Mean	Std.Dev
Severity	Blocking	6.86	0.58	1.07
	Functional	10.94	0.99	1.39
	Enhancement	10.32	1.22	1.59
	Cosmetic	10.06	1.24	1.73
Priority	Critical	7.82	0.56	0.90
	Serious	10.94	0.85	1.29
	Medium	10.06	1.40	1.69
	Minor	6.30	1.00	1.39

Table 4
Comparison of bug fixing time (in days) based on severities and priorities–replicated study.

Feature	Value	Max	Mean	Std. Dev
Severity	blocker	18	2.78	3.96
	Critical	116	6.21	16.96
	enhancement	100	4.15	13.01
	major	129	3.99	11.87
	minor	121	4.32	13.33
	normal	124	4.68	12.21
	trivial	121	5.15	15.32
Priority	P0	129	4.54	12.88
	P1	40	11.57	9.68
	P2	27	7.78	8.18
	P3	40	12.80	12.39
	P4	20	10.00	14.14
	P5	19	7.00	10.44

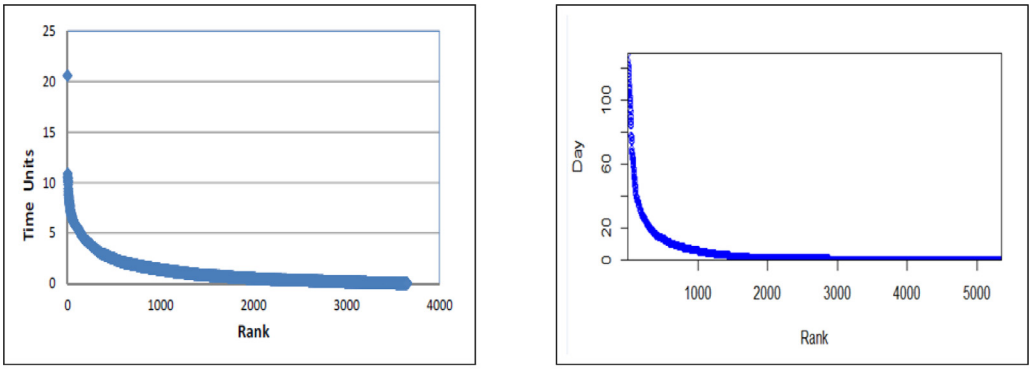
Tables 3 and 4 present the analysis of bug fixing time with respect to severity and priority in the primary and the replicated studies, respectively. We expected that the bugs with higher severity and higher priority would be fixed sooner, but this trend does not hold in all cases either for the original study or for the replicated one. In the cases where priority or severity has the higher mean, large variation may cause the trend to break. Therefore, it is necessary to use a machine learning tool because it is difficult to find a particular rule for predicting bug fixing time.

5.1. RQ1: How many bugs can be fixed in a given amount of time?

As mentioned in Section 3, the state transition probability is the conditional probability used to describe the Markov model, and it is defined as the probability of a bug being in state "i" at time $t+1$ if the bug is in state "j" at time t. The training period for calculating the transition matrix is 12 months (2014–2015), and it is used to predict the number of fixed bugs after 1 month, 2 months, and 3 months in 2015. For calculating the transition matrix, we chunked the data into 12 months. For each month, we counted the number of state transitions (from one state to the other) for the newly created defects. Thereafter, we counted the total number of state transitions during the year, created a table of defect transi-

Table 2
Severities distance measure.

	Blocker	Critical	Major	Normal	Minor	Trivial	Enhancement
Blocker	0.00	0.14	0.29	0.43	0.57	0.71	0.86
Critical	0.14	0.00	0.14	0.29	0.43	0.57	0.71
Major	0.29	0.14	0.00	0.14	0.29	0.43	0.57
Normal	0.43	0.29	0.14	0.00	0.14	0.29	0.43
Minor	0.57	0.43	0.29	0.14	0.00	0.14	0.29
Trivial	0.71	0.57	0.43	0.29	0.14	0.00	0.14
Enhancement	0.86	0.71	0.57	0.43	0.29	0.14	0.00



(a) CA technologies [7] (b) Firefox

Fig. 5. The cumulative distribution function for bug-fixing time.

tions, and computed the state transition matrix P. The transition matrix for Firefox is given below:

$$\begin{pmatrix} 0.2236 & 0.0003 & 0.0006 & 0.0701 & 0.7027 & 0.0000 & 0.0000 \\ 0.0000 & 0.8953 & 0.0606 & 0.0000 & 0.0387 & 0.0000 & 0.0000 \\ 0.0000 & 0.0096 & 0.0385 & 0.0000 & 0.9519 & 0.0000 & 0.0000 \\ 0.0028 & 0.0000 & 0.0000 & 0.2103 & 0.7869 & 0.0000 & 0.0000 \\ 0.1504 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.7258 & 0.1201 \\ 1.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 \\ 0.0000 & 0.0000 & 0.0303 & 0.1515 & 0.8182 & 0.0000 & 0.0000 \end{pmatrix}$$

Note that there are only seven states in the transition matrix because no "CLOSED" status appeared in the training set. We performed a quick exploratory analysis on the status of Firefox bugs and found out that QA never closed bugs. Therefore, we deleted "CLOSED" from our transition probability matrix.

Initially, the total number of Firefox bugs was 66,263 and the initial distribution of bugs α_0 for Firefox is as follows:

$$(0.0812 \quad 0.0479 \quad 0.0016 \quad 0.0820 \quad 0.7133 \quad 0.0699 \quad 0.0038)$$

To find total number of bugs at time $t+1$, we applied regression analysis to 12 months of historical data–$T_{t+1} = a * T_t + b$–and found a model with $R^2 = 0.995$. As in the original study, the regression model was used to predict the total number of bugs in three consecutive months.

With the transition matrix, initial state, and total number of bugs, we can compute the total number of resolved bugs for the next three months by using the following formula. Refer to Section 4.3 for further details on calculating U_{t+1}, Ne_{t+1}, and R_{t+1}.

$$N_{t+1} = prob_{uncof-res} * (U_{t+1}) + prob_{new-res} * (Ne_{t+1}) + prob_{reop-res} * (R_{t+1}) \quad (27)$$

Table 5 shows the results of both the original study on CA Technologies (projects A, B, and C) and our replication study (Firefox). As in the case of the original study, we showed the percentage of values of both predicted and actual number of fixed bugs with regard to the total number of bugs. Comparing the monthly MRE of CA Technologies' projects with those of the Firefox project shows that our result is consistent with those of the primary study. The highest MRE is 2.745% for the third month, and the lowest MRE is 0.766% for the first month. Firefox follows Project A with respect to the MRE trend (lowest to highest). Fig. 6 shows the performance of the average-based model in comparison with the Markov-based model. It can be inferred from the figure that the Markov model outperforms the average-based model in both project A and Firefox. However, the improvement achieved with the Markov model is more pronounced in the case of the Firefox project compared to

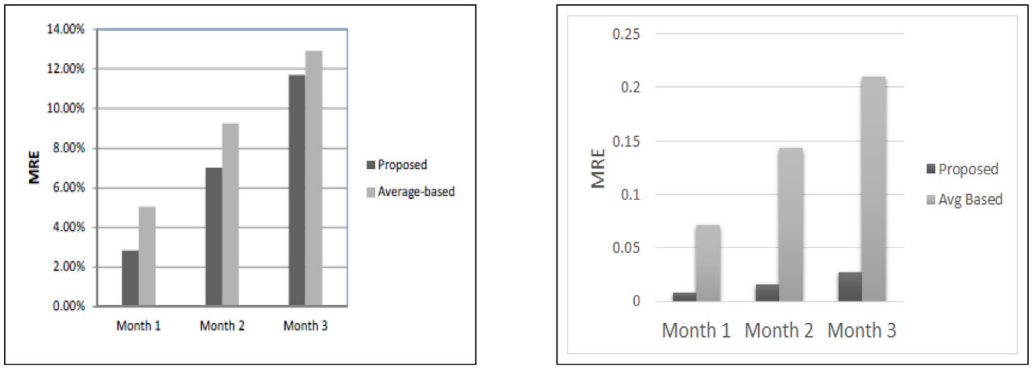
Table 5
Results of Markov-based prediction model.

	Period	Predicted	Actual	MRE
Project A	Month 1	18.38%	18.91%	2.831%
Original study	Month 2	19.81%	21.30%	6.997%
	Month 3	21.00%	23.78%	11.686%
Project B	Month 1	24.70%	25.90%	4.633%
Original study	Month 2	26.60%	27.10%	1.845%
	Month 3	28.20%	29.20%	3.425%
Project C	Month 1	18.84%	18.64%	1.070%
Original study	Month 2	20.14%	20.44%	0.015%
	Month 3	21.24%	21.04%	0.948%
Firefox	Month 1	70.50%	71.04%	0.766%
Replicated study	Month 2	69.61%	70.72%	1.579%
	Month 3	68.38%	67.27%	2.745%

that in the case of Project A, implying that the Markov model can predict more accurately in the case of the Firefox project. However, because the original study did not provide the difference values, we can not perform any statistical analysis.

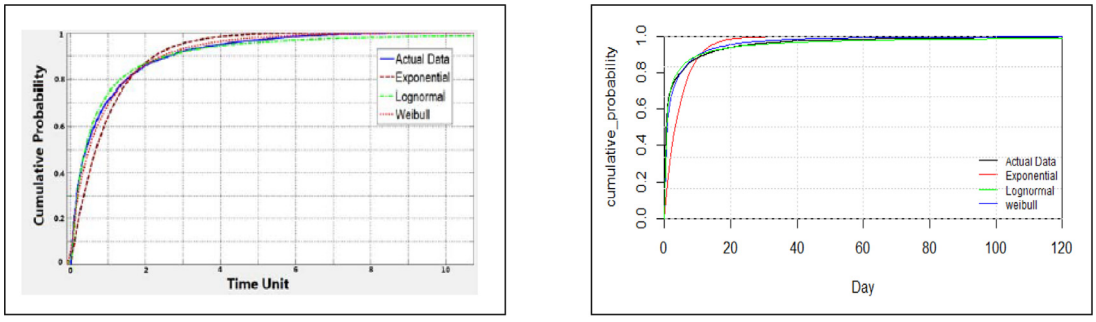
Although the Firefox lifecycle has a more complicated Markov model setting compared to the CA Technologies model, we believe that the number of states does not affect model performance. It also confirms the similarity between the bug handling systems in the two studies, although a simplified model is used in the primary study . The Firefox bug handling process can also be simplified by considering the three initial states of Unconfirmed, New, and Assigned as "Submitted," Resolved as "Fixed," and Verified and Closed as "Closed." The only state for which we can not find a perfect match in the CA Technologies process is Reopen. Furthermore, this proves that the Markov assumption holds in the bug handling systems of the two studies. In other words, the future state of a bug depends only on the current state and not on the sequence of past states. Owing to this memoryless property of the Markov model, which holds in both studies, we concluded that different states of the two studies do not have a significant effect on model accuracy.

In addition, we performed side analysis, which was not done in the original study. We are also interested in finding the percentage of Resolved bugs that were initially "Deferred" bugs. Deferred bugs are the bugs which are reported but not resolved in the current release. This analysis would provide us an insight about the distribution of monthly resolution efforts between non-deferred bugs and deferred bugs. The percentage can be computed by deeper investigation of state transition. According to the matrix, bugs might transfer either from ["Unconfirmed" to "Deferred" to "Resolved"] or ["Reopen" to "Deferred" to "Resolved"]. Therefore, by finding the probability of the aforementioned status changes and multiplying



(a) CA technologies, Project A [7] (b) Firefox

Fig. 6. The comparisons between Markov-based model with the Average-based method.



(a) CA technologies [7] (b) Firefox

Fig. 7. The cumulative distributions function for bug-fixing time.

them with the numbers of unconfirmed and reopen bugs, we can easily find the percentage of resolved bugs transferred from deferred bugs. The result shows that 9.1%, 12.72% , and 13.57% of resolved bugs were initially deferred bugs in the three consecutive months. Hence, only a small portion of the resolved bugs were originally deferred bugs. We may conclude if the bugs are deferred to the next release, the probability of them being resolved decreases because the development team shows a weak tendency to fix them. Consequently, the probability of being in danger of unsustainability increases.

5.2. RQ2: How long would it take to fix a given number of bugs?

Because we would like to manage deferred bugs in our future study, we initially need to investigate the effect of postponing bug fixing in the issue tracking system. However, that can not be achieved unless we gain intuition about the bugs that are reported and resolved in the same time without any postponement. Therefore, for the last two research questions, we filtered our data to exclude non-deferred bugs.

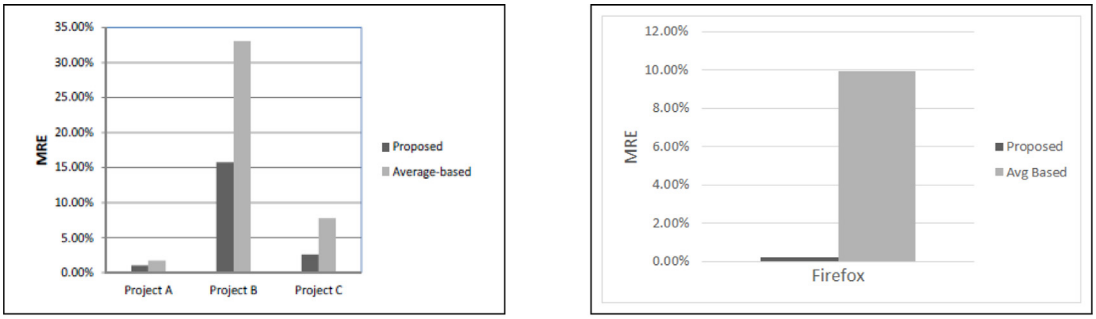
To perform Monte Carlo simulation, we initially found the best empirical fitting distribution of bug fixing time from three distributions: lognormal, Weibull, and exponential. The best distribution is the one with the highest R^2 and the lowest S_e. Table 6 shows the best-fitting distributions for three projects from CA Technologies and the Firefox project. We can see that the best-fitting distribution for Project A is Weibull, and for Projects B, C, and Firefox, it is lognormal. Note that the magnitude of S_e depends on the time unit used for time prediction. We consider "Day" as the time unit, but the original study does not disclose the time unit due to confidentiality. Fig. 7 shows the cumulative distribution function for

Table 6
Evaluation of bug fixing time distribution.

	Distribution	R^2	S_e
Project A	Exponential	0.9229	0.0438
Original study	Lognormal	0.9879	0.0159
	Weibull	0.9933	0.0128
Project B	Exponential	0.4972	0.0820
Original study	Lognormal	0.9800	0.0163
	Weibull	0.9578	0.0238
Project C	Exponential	0.8482	0.0639
Original study	Lognormal	0.9868	0.0188
	Weibull	0.9776	0.0246
Firefox	Exponential	0.4963	8.9330
Replicated study	Lognormal	0.9998	0.1777
	Weibull	0.8817	4.327

project A and Firefox. The higher R^2 value can be ascribed to the filtering of the training set to exclude non-deferred bugs. Lamkanfi and Demeyer (2012) also reported that filtering the bugs report to short life bugs would improve the performance of the prediction model.

Using the best-fitting distribution, we ran the Monte Carlo simulation and repeated the experiment 100 times. We took the average of the fixing times in the 100 runs as the fixing time for the testing set. The results of the Monte Carlo simulation were compared with those of the average-based model as a baseline. Fig. 8 presents the MRE comparison of the original study and the replicated study. The original study reported MRE ranges of 1.04%–15.76% for the three projects. The MRE of the replicated model is even lower than those projects and is equal to 0.2%. The relative



(a) CA technologies [7] (b) Firefox

Fig. 8. The Monte Carlo simulation results for predicting bug-fixing time.

Table 7
Comparison of F-measure for classifying the bugs as slow or quick.

	δ	0.1	0.2	0.4	1	2
Project A	Proposed	0.704	0.659	0.668	0.728	0.852
Original study	BayesNet	0.702	0.633	0.638	0.704	0.828
	NaiveBayes	0.701	0.636	0.638	0.698	0.832
	RBFNetwork	0.691	0.631	0.630	0.696	0.833
	Decision Tree	0.653	0.495	0.643	0.601	0.798
Project B	Proposed	0.679	0.685	0.650	0.672	0.703
Original study	BayesNet	0.669	0.613	0.597	0.637	0.643
	NaiveBayes	0.670	0.610	0.605	0.642	0.645
	RBFNetwork	0.679	0.614	0.610	0.621	0.628
	Decision Tree	0.618	0.520	0.627	0.670	0.632
Project C	Proposed	0.770	0.793	0.762	0.769	0.773
Original study	BayesNet	0.787	0.768	0.722	0.745	0.780
	NaiveBayes	0.790	0.773	0.739	0.745	0.775
	RBFNetwork	0.784	0.780	0.739	0.741	0.750
	Decision Tree	0.662	0.776	0.741	0.740	0.765
Firefox	Proposed	0.5616	0.6128	0.6102	0.6144	0.7357
Replicated study	BayesNet	0.4967	0.4912	0.5163	0.5495	0.5314
	NaiveBayes	0.4980	0.4993	0.5122	0.5505	0.5701
	RBFNetwork	0.3820	0.3521	0.4425	0.3864	0.5230
	Decision Tree	0.3820	0.3521	0.3542	0.3864	0.6599

improvement in the replicated study (97%) is also higher than that in the original study (38.5%–67.1%.).

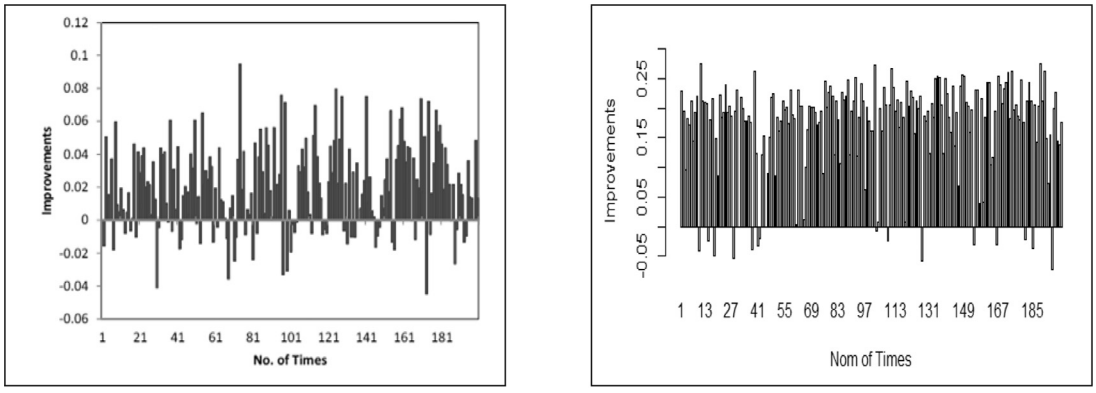
The results of the Monte Carlo simulation for predicting the fixing time are very promising. It satisfied the two main objectives of this study: first, it verified the robustness of the model, and second, it provided us with a model that can predict the fixing time for a given number of bugs upon their arrival before the end of the release cycle.

5.3. RQ3: How much time is required to fix a particular bug?

Table 7 presents a comparison of the proposed KNN classification scheme with four other classification schemes from Weka (BayesNet, NaiveBayes, RBFNetwork, and Decision tree). The models predict fixing times under different time thresholds δ (0.1, 0.2, 0.4, 1, and 2 time unit). The first three projects are from CA Technologies (original study), and the last one is Firefox from the replicated study. We can see that the model performance increases as the threshold increases. As can be inferred from the table for the original study, the F-measure of the proposed model ranges from 0.65 to 0.85, with an average of 0.7245. For the Firefox project, the performance is lower, ranging from 0.56 to 0.74 with an average of 0.6269. Nevertheless, the proposed scheme generally outperforms all the other classification techniques considered herein. Similar to the original study, we ran the pairwise Wilcoxon test, and the results confirm that the proposed method outperforms other classification methods significantly with a p value of 0.03125.

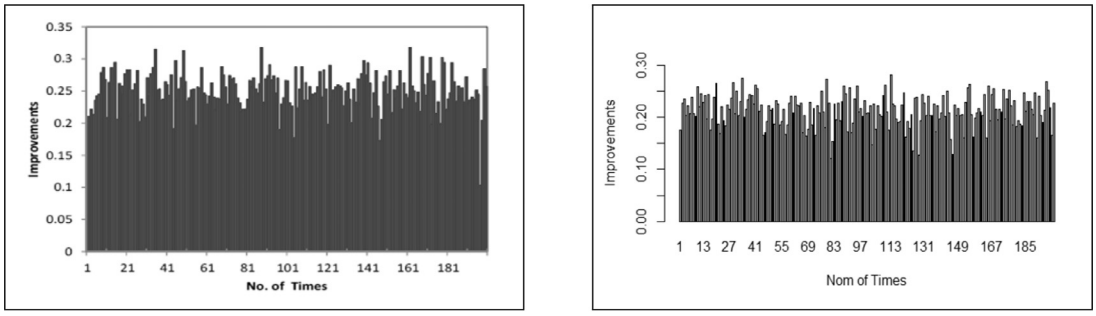
To confirm the extent to which the proposed method can improve performance, we measured the magnitude of improvement (difference between the F-measures of the proposed KNN scheme and the other classification schemes). Decision tree and RFB network were chosen as the benchmark models. We ran the prediction 200 times, and in each run, the same training and testing sets were used for the classification schemes compared. We used a threshold of 0.6 for classifying bugs into quick and slow. Fig. 9 shows the improvement of the proposed network in the cases of the original study and the replicated study (Firefox). The improvement ranges between 0.05% and 9.64% in the original study and between 0.27% and 27.84% in the replicated study (Firefox). Furthermore, the replicated study shows that 184 out 200 times, the proposed KNN scheme outperformed RFB network. Fig. 10 shows the improvement achieved with the proposed network compared to decision tree. For project A in the primary study, the improvement is between 10.35% and 31.78%. For Firefox, this value ranges between 12.11% and 28.17%.

The result of this section verifies that the KNN-based model is able to classify bugs into slow and quick for open source projects as well. The advantages of this model compared to the existing models in the literature is that the features are easy to extract and, hence, less expensive. Furthermore, the proposed model provides insight into the behaviour of non-deferred and deferred bugs. If the model predicts the fixing time of a bug as quick, it means that if the fix is implemented upon arrival of the bug, the fixing time would be less than the threshold. Later, we can examine how post-



(a) CA technologies [7] (b) Firefox

Fig. 9. Improvement of KNN classification over RBFNetwork.



(a) CA technologies [7] (b) Firefox

Fig. 10. Improvement of KNN classification over Decision Tree.

poning a bug might affect its fixing time. If the fixing time is still less than the threshold, we conclude that deferring the bug does not have a harmful effect on the system, but if its fixing time is greater than the threshold, the system is troubled because the bug was deferred.

5.3.1. Which is the most effective feature?

The original study also discussed the most effective feature in a side analysis. Fig. 11 shows the most effective features in both studies. Three feature-ranking methods were used in both studies, namely, chi square, information gain, and gain ratio (Witten and Frank, 2005). In the original study, the two most effective features were owner and submitter. In the replicated study, creator and submitter are the most important features. Several studies in the literature confirm that developer and submitter are highly correlated with bug fixing time (Bhattacharya and Neamtiu, 2011; Guo et al., 2010). We opine that the contribution of owner may be related to developers' skill, experience, and creativity. The contribution of submitter might be attributed to their reputation, communication skills, and quality and clearness of their bug reports.

6. Threats to validity

In this section, we discuss the threats to the validity of this study:

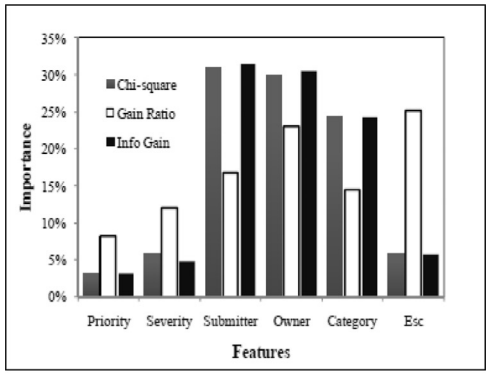
Internal Validity: As in the case of the original study, the first threat is suitability of the Markov model for a stable, large evolving system. The model only works for stochastic processes in which future events are dependent only on the current system status. Apparently, for short-lived non-systematic bug handling projects, the model does not work well (Zhang et al., 2013). The second threat is sampling bias in KNN classification. To avoid sampling bias in our experiment, as in the original study, we repeated the experiment 200 times and ensured that the behaviour of the proposed classification scheme was not related to specific sampled data. The third threat is statistical validity. In the case of statistical validity threats, we ensured that we did not violate any assumptions of the test. Similar to the primary study, we used the Wilcoxon Test (non parametric test) with less restrictive assumptions.

To mitigate the risk of researcher bias, we strictly followed the original study step-by-step to the extent possible. In the case of a misunderstanding, we communicated with the main author of the original study and made sure that we understood the methodology and the study approach well.

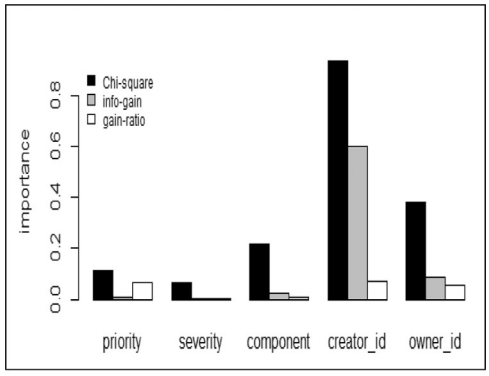
External Validity: Our results are based on the data collected from an open source bug repository, and they reflect the behaviour of bug fixing time in that specific organization. Although the original study was conducted on a commercial product and our study is based on an open source project, drawing general conclusions from this empirical study is not possible. Our effort is to extend the external validity of the original study from commercial products to open source artifacts. However, we are aware that this replication study only focused on one open source project and generalizing the results to all open source projects is not possible.

7. Conclusions across studies

In this paper, we replicated Zhang et al.'s work on predicting bug fixing time. Their ultimate objective was predicting bug fixing time to estimate maintenance effort and improve resource alloca-



(a) CA technologies [7]



(b) Firefox

Fig. 11. The importance of features.

tion in project management. Our goal is applying Carver's guidelines for replication and gaining a deeper understanding of managing deferred bugs in an issue tracking system. The original study focused on three research questions:

- How many bugs can be fixed in a given amount of time?
- How long would it take to fix a given number of bugs?
- How much time is required to fix a particular bug?

We can summarize the results of our replication study as follows:

- The Markov model is capable of predicting the number of fixed bugs in three consecutive months with a mean relative error of 1.70% in the Firefox project compared to a mean relative error of 3.72% for the CA Technologies projects.
- With the Monte Carlo simulation, we could predict the fixing time for a given number of bugs with mean relative errors of 0.2% and 6.45% for Firefox and CA Technologies projects, respectively.
- The KNN-based model classified the time for fixing bugs into slow and quick with an F-measure of 62.69% for Firefox and 72.45% for CA Technologies projects.

Original study was conducted on commercial project data from CA Technologies. To generalize the methodology, we replicated that study by using an open source bug tracking system. The results show that although the bug tracking system of the Firefox project may appear to be complex, in general, it is similar to the bug handling system of CA Technologies. Therefore, different states in the Markov model did not significantly affect the results. Additionally, the original study examined a simplified bug handling system and believed that appending a greater number of states such as "Assigned" and "Verified" may improve prediction. In the replicated study, we showed that the simplified process was a threat for their study and a finer Markov model would improve the accuracy slightly. Furthermore, compatible results indicate that their model is robust enough to be generalized, and we can rely on it in our future study on managing deferred bugs in an issue tracking system. Going forward, we can rely on this model as a baseline and determine whether deferring bugs would change their expected resolution time in management of defect debt.

Acknowledgement

This research is supported in part by NSERC Discovery Grant no. 402003-2012. We would like to thank Dr. Hongyu Zhang, the primary author of the original study, for helping us better understand the details of the original study. We also would like to thank Dr. Atakan Erdem for his insightful comments.

References

Akbarinasaji, S., 2015. Toward measuring defect debt and developing a recommender system for their prioritization. In: Proceedings of the 13th International Doctoral Symposium on Empirical Software Engineering, pp. 15–20.
Barnson, M. P., et al., 2001. The bugzilla guide.
Basili, V.R., Shull, F., Lanubile, F., 1999. Building knowledge through families of experiments. Software Eng., IEEE Trans. 25 (4), 456–473.
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.
Boehm, B., Basili, V.R., 2005. Software defect reduction top 10 list. Found. Empirical Software Eng. 426, 37.
Boehm, B.W., et al., 1981. Software engineering economics, 197. Prentice-hall Englewood Cliffs (NJ).
Brooks, A., Daly, J., Miller, J., Roper, M., Wood, M., 1996. Replication of experimental results in software engineering. International Software Engineering Research Network (ISERN) Technical Report ISERN-96-10, University of Strathclyde.
Caglayan, B., Turhan, B., Bener, A., Habayeb, M., Miransky, A., Cialini, E., 2015. Merits of organizational metrics in defect prediction: an industrial replication. In: Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on, 2. IEEE, pp. 89–98.
Carver, J.C., 2010. Towards reporting guidelines for experimental replications: a proposal. 1st International Workshop on Replication in Empirical Software Engineering.
Da Silva, F.Q., Suassuna, M., França, A.C.C., Grubb, A.M., Gouveia, T.B., Monteiro, C.V., dos Santos, I.E., 2014. Replication of empirical studies in software engineering research: a systematic mapping study. Empirical Software Eng. 19 (3), 501–557.
Daly, J., Brooks, A., Miller, J., Roper, M., Wood, M., 1994. Verification of results in software maintenance through external replication. In: Software Maintenance, 1994. Proceedings., International Conference on. IEEE, pp. 50–57.
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.
Guo, P.J., Zimmermann, T., Nagappan, N., Murphy, B., 2010. Characterizing and predicting which bugs get fixed: an empirical study of microsoft windows. In: Software Engineering, 2010 ACM/IEEE 32nd International Conference on, 1. IEEE, pp. 495–504.
Habayeb, M., 2015. On the Use of Hidden Markov Model to Predict the Time to Fix Bugs. Ryerson University Master's thesis.
Habayeb, M., Miranskyy, A., Murtaza, S.S., Buchanan, L., Bener, A., 2015. The firefox temporal defect dataset. In: Proceedings of the 12th Working Conference on Mining Software Repositories. IEEE Press, pp. 498–501.
Hooimeijer, P., Weimer, W., 2007. Modeling bug report quality. In: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering. ACM, pp. 34–43.
Juristo, N., Gómez, O.S., 2012. Replication of software engineering experiments. In: Empirical software engineering and verification. Springer, pp. 60–88.
Kaner, C., Falk, J., Nguyen, H.Q., 1999. Testing computer software. Dreamtech Press.
Kim, S., Whitehead Jr, E.J., 2006. How long did it take to fix bugs? In: Proceedings of the 2006 international workshop on Mining software repositories. ACM, pp. 173–174.
Ko, Y., 2012. A study of term weighting schemes using class information for text classification. In: Proceedings of the 35th international ACM SIGIR conference on Research and development in information retrieval. ACM, pp. 1029–1030.
La Sorte, M.A., 1972. Replication as a verification technique in survey research: a paradigm. Sociol. Q. 13 (2), 218–227.
Lamkanfi, A., Demeyer, S., 2012. Filtering bug reports for fix-time analysis. In: Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on. IEEE, pp. 379–384.
de Magalhães, C.V., da Silva, F.Q., Santos, R.E., Suassuna, M., 2015. Investigations about replication of empirical studies in software engineering: a systematic mapping study. Inf. Softw. Technol. 64, 76–101.

Panjer, L.D., 2007. Predicting eclipse bug lifetimes. In: Proceedings of the Fourth International Workshop on mining software repositories. IEEE Computer Society, p. 29.

Shull, F., Basili, V., Carver, J., Maldonado, J.C., Travassos, G.H., Mendonça, M., Fabbri, S., 2002. Replicating software engineering experiments: addressing the tacit knowledge problem. In: Empirical Software Engineering, 2002. Proceedings. 2002 International Symposium n. IEEE, pp. 7–16.

Shull, F.J., Carver, J.C., Vegas, S., Juristo, N., 2008. The role of replications in empirical software engineering. Empirical Software Eng. 13 (2), 211–218.

Sjoberg, D.I., Dyba, T., Jorgensen, M., 2007. The future of empirical methods in software engineering research. In: 2007 Future of Software Engineering. IEEE Computer Society, pp. 358–378.

Snipes, W., Robinson, B., Guo, Y., Seaman, C., 2012. Defining the decision factors for managing defects: a technical debt perspective. In: Managing Technical Debt (MTD), 2012 Third International Workshop on. IEEE, pp. 54–60.

Tate, K., 2005. Sustainable software development: an agile perspective. Addison-Wesley Professional.

Tom, E., Aurum, A., Vidgen, R., 2013. An exploration of technical debt. J. Syst. Software 86 (6), 1498–1516.

Wikipedia, 2015a. Firefox — Wikipedia, the free encyclopedia. [Online; accessed 22-December-2015], URL https://en.wikipedia.org/wiki/Firefox.

Wikipedia, 2015b. Software bug — Wikipedia, the free encyclopedia. [Online; accessed 22-December-2015], URL https://en.wikipedia.org/wiki/Software_bug.

Witten, I.H., Frank, E., 2005. Data mining: practical machine learning tools and techniques. Morgan Kaufmann.

Zhang, F., Khomh, F., Zou, Y., Hassan, A.E., 2012. An empirical study on factors impacting bug fixing time. In: Reverse Engineering (WCRE), 2012 19th Working Conference on. IEEE, pp. 225–234.

Zhang, H., Gong, L., Versteeg, S., 2013. Predicting bug-fixing time: an empirical study of commercial software projects. In: Proceedings of the 2013 International Conference on Software Engineering. IEEE Press, pp. 1042–1051.