On the Prediction of Long-lived Bugs: An Analysis and Comparative Study using FLOSS Projects

Luiz Alberto Ferreira Gomes

Institute of Computing (IC), University of Campinas (UNICAMP), Campinas, Brazil. Institute of Exact Sciences and Informatics (ICEI), Pontifical Catholic University of Minas Gerais (PUC-MG), Minas Gerais, Brazil.

Ricardo da Silva Torres

Department of ICT and Natural Sciences, Norwegian University of Science and Technology (NTNU)

Mario Lúcio Côrtes

Institute of Computing (IC), University of Campinas (UNICAMP), Campinas, Brazil

Abstract

Context: Software evolution and maintenance activities in today's Free/Libre Open Source Software (FLOSS) rely primarily on information extracted from bug reports registered in bug tracking systems. Many studies point out that most bugs that adversely affect the user's experience across versions of FLOSS projects are long-lived bugs. However, proposed approaches that support bug fixing procedures do not consider the real-world lifecycle of a bug, in which bugs are often fixed very fast. This may lead to useless efforts to automate the bug management process.
Objective: This study aims to confirm whether the number of long-lived bugs is significantly high in popular open-source projects and to characterize the population of long-lived bugs by considering the attributes of bug reports. We also aim to conduct a comparative study evaluating the prediction accuracy of five well-known machine learning algorithms and text mining techniques in the task of predicting long-lived bugs.
Method: We collected bug reports from six popular open-source projects repositories (Eclipse, Freedesktop, Gnome, GCC, Mozilla, and WineHQ) and used the following machine learning algorithms to predict long-lived bugs: K-Nearest Neighbor, Naïve Bayes, Neural Networks, Random Forest, and Support Vector Machines.
Results: Our results show that long-lived bugs are relatively frequent (varying

Email addresses: `luizgomes@pucpcaldas.br` (Luiz Alberto Ferreira Gomes), `ricardo.torres@ntnu.no` (Ricardo da Silva Torres), `cortes@ic.unicamp.br` (Mario Lúcio Côrtes)

from 7.2% to 40.7%) and have unique characteristics, confirming the need to study solutions to support bug fixing management. We found that the Neural Network classifier yielded the best results in comparison to the other algorithms evaluated.
Conclusion: Research efforts regarding long-lived bugs are needed and our results demonstrate that it is possible to predict long-lived bugs with a high accuracy (around 70.7%) despite the use of simple prediction algorithms and text mining methods.

Keywords: Software maintenance, Bug Tracking System, Long-lived Bugs, Machine learning, Text mining.

1. Introduction

A Bug Tracking System (BTS) is an application that keeps track of reported software bugs in software projects [1] and plays a key role in Free/Libre Open Source Software (FLOSS) and Closed Source Software (CSS) as they are a useful communication and collaboration tool. In both environments, the activities of software evolution planning and maintenance rely primarily on the information of opened bug reports. This is particularly true in FLOSS, which is well-known for the existence of thousands of users and developers with distinct levels of expertise spread out around the world, which might create or be responsible for dealing with several bug reports [2].

Frequently, users communicate with a BTS using bug reports, which enables them to communicate with those in charge of maintaining the software system [3]. To do that, users should inform a short description, a long description, and an associated severity level (e.g., blocker, critical, major, minor, or trivial) by filling out a bug report form. After which, a software team member will review the bug report and either approve or reject it (rejections can be due to bug report duplication). If the bug report is approved, the team member will then provide more information; for example, indicating its priority and assigning a person responsible for fixing it.

The number of bug reports in large- and medium-sized software FLOSS projects is frequently high [4]. For instance, the Eclipse project had 84,245 bug reports opened from 2013 to 2015, whereas the Android project had over 107,456, and the JBoss project had over 81,920 in the same period. Therefore, manual handling of bug reports (such as attributing a severity level) may be entirely subjective, tiresome, and error-prone, where a wrong decision within the bug report lifecycle may strongly affect the planning of maintenance activities.

Another critical activity from maintenance planning is allocating resources to handle bugs associated with each opened bug report, which could be significant [5]. Estimating the “bug fix time” is essential for many stakeholders [6, 7]. For software managers, it is one of the main factors that helps them to perform such allocation more effectively [8, 9, 10]. In large system projects with tight schedules and limited resources, it may not be possible to close all known bugs before the upcoming release. Many bugs can be reported over a long period, incurring in costs, such as wasted time in release planning and defect analysis, and inspection [11, 12]. Thus, software managers have to decide which bugs to fix in the current release and which to defer to upcoming versions. This may accumulate an ever-expanding backlog of unfixed defects [12]. For these reasons, timely identification of bugs with long fixing time, right after opening the related report, may help managers to prioritize and allocate resources more effectively when faced with a large number of bug reports [9, 13].

Estimating or predicting a bug as long-lived is not only very important for software managers in their daily activities, but also it is equally essential for the quality assurance team. Software bugs, mainly those with long fixing time, may have continuous adverse effects on software quality, including user inconvenience, inappropriate functionality, and security issues [13]. For instance, in complex software systems, if bugs are not quickly fixed, structural problems could emerge, compromising even the reproduction of a given bug [14]. Approximately 90% of long-lived bugs adversely affect users' experience [15], which may disturb end-users for a long time, even when in small number. In such scenarios, users may switch software and favor the competition [16]. In summary, the proper classification of long-lived bugs may be essential to increase customer satisfaction.

These facts suggest that many long-lived bugs could be fixed more quickly through careful selection and prioritization if developers could predict a long-lived bug. Machine Learning (ML) algorithms have been successfully applied in solving many real-world prediction problems, including those related to automating bug report handling, such as bug severity prediction [17].[1] In addition, since bug reports typically come with textual descriptions, text mining techniques are likely candidates in providing appropriate input for these algorithms.

In this context, the general purpose of our research is to compare the accuracy of traditional ML algorithms in predicting long-lived bugs in FLOSS projects. Based on this purpose our specific goals are:

G_1. Confirm whether the population of long-lived bugs in popular FLOSS projects is significant, as well as investigate their main characteristics.

G_2. Conduct a comparative study considering the evaluation of five well-known machine learning algorithms to predict long-lived bugs.

- **$G_{2.1}$**. Evaluate how the number of terms from unstructured text fields in bug reports affect long-lived bug prediction;
- **$G_{2.2}$**. Evaluate and identify the best bug fixing time threshold in order to classify a bug as long-lived.

[1] In our previous work [17], we provided a comprehensive review of recent research efforts on automating bug report severity prediction, where we showed that many strategies to predict bug severity were based on machine learning methods. In this study, however, we focus on the long-lived/short-lived classification problem, comparing different classifiers and bug-related characteristics.

G$_3$. Understand the characteristics of long-lived bugs predicted with high and low accuracy in the testing phase.

To meet these goals, this study addresses the following research questions:

RQ$_1$. *Is there a common understanding of what long-lived bugs are in the literature?* This research question aims to investigate if there is a shared understanding of what a long-lived bug is in the bug-related literature. Such understanding is essential to determine a bug fixing time threshold that could be used to label a bug as long-lived and thus address the other proposed research questions.

RQ$_2$. *How frequent are long-lived bugs in FLOSS projects?* Assessing the proportion of long-lived bugs in FLOSS projects could indicate if investing in long-lived bug prediction models is beneficial.

RQ$_3$. *What are the main characteristics of long-lived bugs compared to short-lived bugs in FLOSS projects?* Based on information provided in bug reports, we aim to determine the characteristics of long-lived bugs compared to short-lived bugs in FLOSS projects and thus allow developers to clearly distinguish long-lived from short-lived bugs.

RQ$_4$. *What is the comparative accuracy of machine learning algorithms when predicting long-lived bugs?* We aim to compare the accuracy of long-lived bug prediction made by different ML algorithms, where the algorithm with the best prediction capability is best fit in terms of balanced accuracy. This understanding is essential to select models that could be used to automate the prediction of long-lived bugs with high accuracy.

RQ$_5$. *What are the main characteristics of bugs correctly predicted as long-lived and incorrectly predicted as short-lived?* Even if our long-lived prediction models do not correctly predict many long-lived bugs, investigating the characteristics of those with high accuracy and the characteristics of those with low accuracy could be very useful to improve the prediction models.

In summary, the contributions of our research are:

- Characterization of long-lived bugs in relevant FLOSS projects;
- An evaluation of the performance of different models in predicting long-lived bugs;
- Characterization of how the number of terms will affect the prediction of long-lived bugs;
- Identification of suitable values for the bug fixing time threshold and number of terms in the feature vector used to predict long-lived bugs.

The article is organized as follows. Section 2 provides the background terminology and concepts about bug tracking systems, text mining, and machine learning techniques. Section 3 presents related work. Section 4 describes the methodology used. Section 5 reports our results. Section 6 describes the significance of our findings and how they can be interpreted. Section 7 describes the main threats to the validity of our research. Finally, Section 8 states our conclusions and highlights future research addressing this topic.

2. Terminology and Concepts

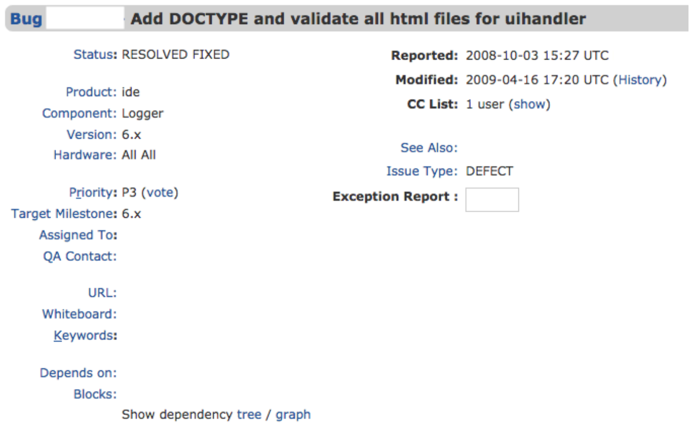
This section provides an overview of basic concepts related to bug tracking systems, text mining, machine learning, and evaluation metrics needed to better understand our paper.

2.1. Bug Tracking Systems

Bug Tracking System (BTS) [1] is a software application that records and tracks information regarding change requests, bug fixes, and technical support that could occur during the life cycle of any given software.

2.1.1. Bug Report

Usually, while reporting a bug in a BTS, a user is asked to fill out a form with the information required to reproduce and fix a bug. Normally, this form is called bug report. Figure 1 shows an example of a bug report form as supplied by Bugzilla containing data attributes that describe a bug (e.g., summary and description). Although there is no agreement on the terminology or on the amount of information that users and maintenance teams must provide to fill a bug report, they often describe their needs in popular BTS (e.g., Bugzilla, Jira, and Redmine) [18], providing at least information about the attributes shown in Table 1.



Bug Add DOCTYPE and validate all html files for uihandler

Status:	RESOLVED FIXED	Reported:	2008-10-03 15:27 UTC
		Modified:	2009-04-16 17:20 UTC (History)
Product:	ide	CC List:	1 user (show)
Component:	Logger		
Version:	6.x		
Hardware:	All All	See Also:	
		Issue Type:	DEFECT
Priority:	P3 (vote)	Exception Report :	
Target Milestone:	6.x		
Assigned To:			
QA Contact:			
URL:			
Whiteboard:			
Keywords:			
Depends on:			
Blocks:			
	Show dependency tree / graph		

Figure 1: A bug report example.

Table 1: Common attributes of a bug report.

Bug Reporter	Account of the user who created the bug report.
Summary	Short description of the report in one line.
Description	Long and detailed description of report in many lines of text. It could include source code snippets and stack tracing reports.
Component	Each product is divided into different components (e.g., Core, Editor, and UI).
Severity	Report severity level (e.g., blocker, critical, major, minor, and trivial).
Type	Type of report (e.g., bug, improvement, and new feature)
Assignee	Account of the user in charge of fixing the bug.

After the user has reported a bug, the development team is in charge of its assessment, which consists of approving or rejecting the bug. In case of approval, the team may provide complementary information, for example, assigning a person responsible for the request or defining the severity level for the bug. Typically, the steps a bug report goes through are modeled as a state machine (Figure 2). At first, the bug report is said to be *Unconfirmed*. The developer team can change the status to *Resolved* if the bug is not confirmed or, otherwise, to *New*. When someone becomes in charge of fixing the bug, the bug report state is changed to *Assigned* by the developer team. Therefore, in the standard flow, the bug report status is assigned to *Resolved* (bug fixed), then *Verified* (bug checked), and finally *Closed*.

As shown in Figure 2, other state transitions may occur throughout the bug report life cycle. All changes that occurred in a bug report are often stored in a repository, keeping valuable historical information about a particular software.

2.2. Machine Learning (Classification) Algorithms

Machine Learning (ML) [20] is an application of Artificial Intelligence (AI) that provides systems the ability to learn and improve from experience without being explicitly programmed. There are two types of ML algorithms: predictive (or supervised) and descriptive (or unsupervised). Long-lived bug prediction is considered a supervised problem.

A predictive algorithm builds a model based on historical training data and uses this model to predict, from the values of input attributes, an output label (class attribute) for a new sample. A predictive task is called *classification* when the label value is discrete, or *regression* when the label value is continuous.

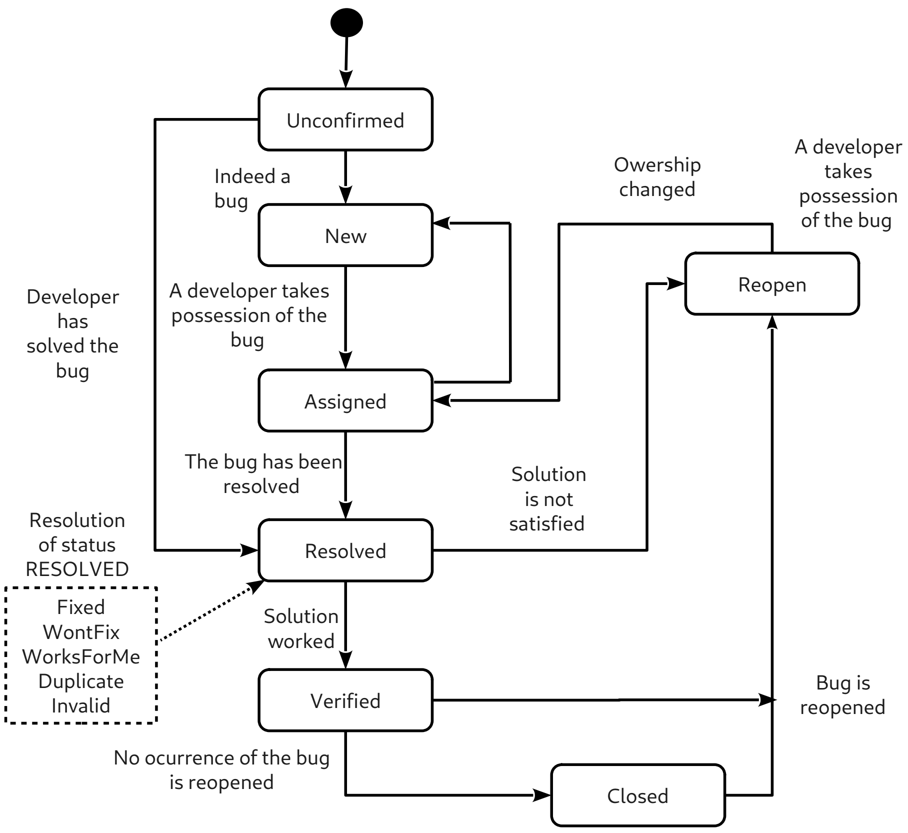

Unconfirmed
Indeed a bug
New
A developer takes possession of the bug
Assigned
The bug has been resolved
Resolved
Solution worked
Verified
No ocurrence of the bug is reopened
Closed
Developer has solved the bug
Resolution of status RESOLVED
Fixed
WontFix
WorksForMe
Duplicate
Invalid
Owership changed
A developer takes possession of the bug
Reopen
Solution is not satisfied
Bug is reopened

Figure 2: The bug report life cycle according to Zhang et al. [19].

2.2.1. Classifiers

An ML algorithm works over a dataset, which contains many samples or instances x_i, where $i = \{1..n\}$. Each instance is composed of $\{x_{i1}, x_{i2}, ..., x_{id}\}$ input attributes or independent variables, where $d = \{1..m\}$, and one output attribute or dependent variable, $x_{i(m+1)}$. Input attributes are commonly named features, or feature vector, and output attribute are commonly named class or category. Traditional ML classification algorithms are k-Nearest Neighbors, Naïve Bayes, Neural Networks, Random Forests, and Support Vector Machines. In practice, they can be applied for both classification and regression tasks. However, this paper regards them only in the classification scenario. A brief description of each algorithm is presented below [21]:

- **k-Nearest Neighbors (KNN)** classifies a new instance based on its similarity measure to the k-nearest labeled neighbors. Usually, the KNN classifier uses the Euclidean distance to quantify the proximity between neighbors. Each instance in a dataset should be represented as a point of an n-dimensional space (feature vector) in order to calculate this distance.

- **Naïve Bayes (NB)** decides to which class an instance belongs based on the Bayesian Theorem of conditional probability. The probabilities of an instance belonging to each of the C_k classes given the instance x is $P(C_k|x)$. Naïve Bayes classifiers assume that, given the class variable, the value of a particular feature is independent of the value of any other feature.
- **Neural Network (NN)** is a learning algorithm that is inspired by the structure and functional aspects of biological neural networks [22]. It is structured as a network of units called neurons, with weighted, directed connections. Neural network models have been demonstrated to be capable of achieving remarkable performance in document classification [23].
- **Random Forest (RF)** [24] relies on two core principles: (i) the creation of hundreds of decision trees and their combination into a single model; and (ii) the final decision based on the majority of the considered trees.
- **Support Vector Machine (SVM)** is a learning algorithm in which each feature vector of each instance is a point in an n-dimensional space. In this space, SVM learns an optimal way to separate the training instances according to their class labels. In the case of linear separability, the output of this algorithm is a hyperplane, which maximizes the separation among feature vectors of different classes. Given a new instance, SVM assigns a label based on which subspace its feature vector belongs to [25].

2.2.2. Evaluation Metrics

The specific metrics we used to assess prediction performance are listed below [21, 26, 27]:

- **Accuracy** is the percentage of correctly classified observations among all observations:
$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{1}$$
where TP is the number of true positives, TN is the number of true negatives, FP is the number of false positives, and FN is the number of false negatives.
- **Balanced Accuracy** is calculated as the average of the proportion of the accuracy of positive class and the accuracy of negative class.. The formula of Balanced Accuracy is:
$$BalancedAccuracy = \frac{\frac{TP}{TP+FN} + \frac{TN}{FP+TN}}{2} \tag{2}$$
- **Sensitivity** corresponds to the hit ratio in the positive class (true positive rate). The formula of Sensitivity is:
$$Sensitivity = \frac{TP}{TP + FN} \tag{3}$$

- **Specificity** corresponds to the hit ratio in the negative class (true negative rate). The formula of Specificity is:

$$Specificity = \frac{TN}{TN + FP} \tag{4}$$

- **Area Under ROC Curve (AUC)**, which ranges between 0 and 1, is used to assess the performance of ML algorithms. An algorithm outperforms another if its AUC value is closer to 1.

- **Kappa** takes into account the accuracy that would be generated by chance in order to determine the agreement among categorical variables. The formula of Kappa is:

$$Kappa = \frac{O - E}{1 - E} \tag{5}$$

  where O is the observed accuracy and E is the expected accuracy based on the marginal totals of the confusion matrix.

2.2.3. Hyper-parameters Tuning

Adjusting the hyper-parameters is key in machine learning as each algorithm has its own set of parameters and these values can significantly impact algorithm performance. The role of this activity is to select the values which will lead to the best algorithm performance in a specific context. In some cases, the values selected can change the resulting model's accuracy from 1 to 95% [28].

Such adjustment is considered an iterative activity that occurs during the training phase of an ML building process. In a typical protocol, researchers will manually pick one or more ML algorithms, after which they will set values for the hyper-parameters of each algorithm (e.g., the value of k in the KNN algorithm).

Researchers commonly use three procedures to set these hyper-parameters [29]: (i) default values specified in software packages, (ii) manual configuration based on the literature, experience, or trial-and-error procedures, or (iii) configuring them for optimal predictive performance by using tuning strategies (e.g., grid search or random search). In summary, researchers train the ML model with the hyper-parameters to get the optimal model accuracy.

After adjusting the hyper-parameters or choosing other ML algorithms, researchers should train the model again until the predicting model achieves a satisfactory prediction accuracy. When this goal is reached, the predictive modeling process can be considered complete. Table 2 describes the hyper-parameters for each ML algorithm used in our study.

2.3. Text Mining

The common ML algorithms cannot directly process unstructured text (e.g., *summary* and *description* fields from the bug report form). Therefore, during the preprocessing stage, these unstructured text fields are converted into more manageable representations. Typically, the content of these fields is represented

Table 2: Description of the hyper-parameters for each ML algorithm investigated.

ML Algorithm	**Hyper-parameters**
KNN	**k**: Number of neighbors
Naïve Bayes	**fL**: Laplace correction or smoother **usekernel**: Allow to use a kernel density estimate for continuous variable versus Gaussian density estimate **ajust**: Bandwidth of kernel density adjustment
Neural Network	**size**: Hidden units **decay**: Weight decay
Random Forest	**mtry**: Randomly Selected Predictors
SVM (Radial Kernel)	**C**: Cost **sigma**: The width for Gaussian distribution

by feature vectors, points of an n-dimensional space. Text mining is the process of converting unstructured text into a structure suited for analysis [30]. It is composed of three primary activities [31]:

- **Tokenization** is the action to parsing a character stream into a sequence of tokens by splitting the stream at delimiters. A token is a block of text or a string of characters (without delimiters such as spaces and punctuation), which is a useful portion of the unstructured data.
- **Stop words removal** eliminates commonly used words that do not provide relevant information to a particular context, including prepositions, conjunctions, articles, verbs, nouns, pronouns, adverbs, and adjectives.
- **Stemming** is the process of reducing or normalizing inflected (or sometimes derived) words into their word stem or base form (e.g., "working" and "worked" into "work").

Two of the most traditional ways of representing a document relies on the use of a bag of words (unigrams) or a bag of bigrams (when two terms appear consecutively, one after the other) [30]. In this approach, all terms represent features, and thus the dimension of the feature space is equal to the number of different terms in all documents (in our context, bug reports).

Methods for assigning weights to features may vary. The simplest one is to assign binary values representing the presence or absence of the term in each text. Term Frequency (TF), another type of encoding scheme, considers the number of times in which the term appears in each document. Term Frequency-Inverse Document Frequency (TD-IDF) is a more complex weighting scheme that takes into account the frequency of the term in each document and in the whole collection. The importance of a term in this scheme is proportional to its frequency in the document and inversely proportional to the frequency of the term in the collection [32].

We highlight that all encoding methods mentioned above often produce sparse matrices in which most of the values are zero [30]. One can quantify the sparsity of a matrix by dividing the number of zeroes in the matrix by the total number of elements in the matrix.

3. Related Work

Giger et al. [33] conducted experimental studies on bug reports from six FLOSS projects hosted by Eclipse, Mozilla, and Gnome, and proposed a classifier based on decision tree algorithm to classify bugs into "fast" or "slow." Furthermore, they showed that the addition of post-submission bug report data of up to one month in the feature vector might improve the model performance.

Lamkanfi et al. [34] observed that, for both Eclipse and Mozilla, a fraction of the bug reports indicated conspicuous fix-times, where the bugs are often fixed within few minutes. They proposed to filter out these conspicuous bug reports when using data mining techniques to predict the fix-times of reported bugs.

Zhang et al. [5] performed an empirical study on bug fixing-times in three projects of CA Technologies company. They proposed a model based on a Markov chain to predict the number of bugs that could be fixed in the future. Also, they employed a Monte Carlo simulation to predict the total fixing-time for a given amount of bugs. Moreover, they classified bugs as "fast" and "slow" regarding different threshold times.

Saha et al. [15] extracted the bug repositories from seven well-known FLOSS projects and analyzed long-lived bugs from five different perspectives: proportion, severity, assignment, reasons, and the nature of fixes. Their study showed that there is a fair number of long-lived bugs in FLOSS projects (although less frequent than short-lived bugs), and more than 90% of them negatively affected user experience. The reasons for these long-lived bugs are many, including, for example, longer assignment time and the lack of understanding of their priority. However, many bugs resulted in long-lived bugs without a specific reason.

Rocha et al. [35] characterized the workflow followed by Mozilla Firefox developers when resolving bugs. They proposed the concept of bug flow graphs (BFG) to help understand the workflow. They concluded that (a) when a bug is not formally assigned to a developer, it takes ten more days to be resolved; (b) approximately 94% of duplicate bugs are resolved within two days or less after they appear in the tracking system; (c) incomplete bugs, which are never assigned to developers, usually take 70 days to be closed; (d) more skilled developers resolve bugs faster in comparison to less skilled ones; (e) for less skilled developers, assigning a person responsible for the bug usually takes more time in comparison to the time taken to actually fix the bug.

Habayeb et al. [36] proposed a novel approach using Hidden Markov models and temporal sequences to predict when a bug report will be closed. The approach is empirically demonstrated using eight years of bug reports collected from the Firefox project. The results indicate around 10% higher accuracy than the frequency based classification approaches.

Akbarinasaji et al. [13] replicated the study performed by Zhang et al. [5] using an open source software project and confirmed their results.

Although the aforementioned studies present relevant results for researchers in this area, we can observe two major shortcomings: First, only two of the existing papers ([33] and [5]) specifically focused on "long-lived bug prediction." However, both investigated just one ML algorithm each: Decision Tree [33] and Markov Chain [5]. In our paper, we have compared the accuracy of five well-known ML algorithms in long-lived bug prediction. Furthermore, we extend this investigation by detailing the cases of success associated with the best predictor.

Second, studies investigating bug related life cycle issues considered few FLOSS projects. In our paper, we have considered six relevant FLOSS projects, from projects funded and maintained by big organizations, such as Eclipse and Mozilla, to voluntary projects, such as WineHQ. In addition, we have characterized long-lived bugs taking into account six different dimensions using bug reports extracted from these projects, which is much more than those observed in related studies.

4. Methodology

This section describes the methodology used to address the research questions. As in traditional methodologies used in ML experiments, it comprises the following steps: data collection, data preprocessing, data description, and training and testing procedures.

4.1. Data Collection

This step includes selecting FLOSS datasets, studying and interpreting their data structure, and finally extracting relevant data from their repositories (feature extraction). In our research, we used the following FLOSS projects: Eclipse, Freedesktop, GCC, Gnome, Mozilla, and WineHQ. All projects are open source, well-established, have a considerable number of registered bug reports, use standard repositories, and were being studied by other researchers [1, 34, 37, 18, 38]. These projects are described as follows:

- **Eclipse**[2]: Eclipse is an integrated development environment used in computer programming and is the most widely used Java IDE.
- **Freedesktop**[3]: Freedesktop hosts the development of free and open source software focused on interoperability and shared technology for open source graphical and desktop systems.

[2]`www.eclipse.org` (As of Nov 2020).
[3]`www.freedesktop.org` (As of Nov. 2020).

- **GCC**[4]: GCC is a compiler system developed by GNU Project supporting C, C++, Objective-C, Objective-C++, Fortran, ADA, Java, Go, among other programming languages.
- **Gnome**[5]: Gnome is a free and open source desktop environment for Unix-like operating systems.
- **Mozilla**[6]: Mozilla encompasses several open source projects, such as Firefox, Thunderbird, and Bugzilla.
- **WineHQ**[7]: WineHQ is a compatibility layer capable of running Windows applications on several POSIX-compliant operating systems, such as Linux, macOS, and BSD.

First, we extracted the datasets from bug reports in XML format of the previous projects hosted in the Bugzilla site. The sample consisted of randomly selected bugs that occurred in the time interval considered. We then stored bug reports in a CSV file format for each project. Table 3 shows additional information on the six datasets considered in our study.

Table 3: FLOSS projects used in our research.

Project	**Number of Bugs**	**Observation Period**	
		From	**To**
Eclipse	9724	2001-10-10	2018-01-30
Freedesktop	7644	2003-02-05	2019-08-15
GCC	9946	1999-02-26	2018-01-31
Gnome	7772	1999-01-02	2018-01-24
Mozilla	9945	1998-04-15	2014-04-22
WineHQ	6037	2000-09-27	2018-04-17

4.2. Data Preprocessing

Raw data previously collected from the Eclipse, Freedesktop, GCC, Gnome, Mozilla, and WineHQ bug report repositories were not correctly structured to serve as input to ML algorithms [39]. The traditional way to address this problem is to run procedures to extract, organize, and structure relevant features out of the raw data. Specific Java applications and R scripts were written to perform feature extraction. The following preprocessing tasks were executed:

[4]`https://gcc.gnu.org/` (As of Nov. 2020).
[5]`https://www.gnome.org` (As of Nov. 2020).
[6]`https://www.mozilla.org` (As of Nov. 2020).
[7]`https://www.winehq.org` (As of Nov. 2020).

- Extraction of relevant features: *bug id*, *opened date*, *component name*, *summary*, *description*, *reporter name*, *assignee name*, *resolution status*, *resolution date*, *severity level*, and *status of bug report*;
- Selection of bug reports with a *Closed* or *Resolved* status and a *Fixed* resolution status. This type of bug report was effectively implemented by the development team and can no longer have their resolution date altered.
- Use of resolution date in bug reports as the ground truth to compute the bug fix time in days (resolution date – open date).

Finally, we stored the preprocessed datasets in CSV format files.[8]

4.3. Training and Testing

We chose five traditional ML algorithms: *KNN*, *Naïve Bayes*, *Neural Network*, *Random Forest*, and *SVM* which were implemented using the Caret[9] R library to build the model for long-lived bug prediction. We performed a grid search procedure to select the best hyper-parameter for each algorithm in training among the hyper-parameters available (Table 4). The Caret library was set up to evaluate each model automatically using the Accuracy metric, and to report the resulting values. Furthermore, we applied the above ML algorithms in an unbalanced and in a balanced dataset. The balanced dataset was created employing the SMOTE [40] method. To yield more stable models in our experiments, we trained and tested each model using the repeated 10×5 fold cross-validation technique [41].[10] Finally, we performed the Wilcoxon signed-rank statistical test [42] (with a significance level of 95%) to evaluate the statistical significance among the ML algorithm performances.

In the testing phase, each long-lived prediction model was validated with 25% of each bug report dataset to measure its accuracy in an unknown dataset.

5. Results

This section reports our results regarding six FLOSS projects and addresses the research questions proposed in our study. The section is organized according to the research question as follows. First, we investigated if there is a common understanding on what is a long-lived bug. We then evaluated the proportion of long-lived bugs in the FLOSS projects studied; in addition, we characterized the population of long-lived bugs based on the bug report attributes. Next, we build different ML models to predict long-lived bugs based on the description field of bug report forms. Finally, we analyzed the properties of true positives and false negatives yielded by our prediction models.

[8]The created dataset will be made publicly available subjected to the acceptance of the paper.

[9]`https://caret.r-forge.r-project.org` (As of Sep 2020).

[10]Repeated Cross-Validation $n \times k$: divides a dataset into k folds in n iterations. In each iteration, it saves a different fold for testing and uses all the others for training [27].

Table 4: Candidate hyperparameters values for ML algorithms.

ML Algorithm	**Hyperparameters**
k-NN	k = {5, 11, 15, 21, 25, 33}
Naïve Bayes	fL = {0..5}, usekernel={TRUE, FALSE}, ajust = {0..5}
Neural Network	size= {10, 20, 30, 40, 50}, decay=0.5
Random Forest	mtry = {25, 50, 75, 100} e ntree = 200.
SVM	C=2^{-5}, 2^{0}, 2^{5}, 2^{10} e sigma= 2^{-15}, 2^{-10}, 2^{-5}, 2^{0}, 2^{5}.

5.1. *RQ1: Is there a common understanding of what long-lived bugs are in the literature?*

Definitions for long-lived bugs found in the literature retrieved from the ACM Digital Library,[11] the IEEE Xplore,[12] ScienceDirect,[13] and Springer.[14] are shown in Table 5.

To build such a table, we constructed a search string using six main terms: "software maintenance," "bug report," "bug fixing," "bug resolution," and "long-lived." The syntax of the string was adapted according to the source (e.g., wildcards, connectors, apostrophes, and quotation marks) and was then applied on the abstract of the manuscripts being searched. We searched databases for manuscripts published between 2010 to 2020. This procedure led to a total of 41 manuscripts. After applying the inclusion and exclusion criteria (Does the paper categorize bugs according to fixing bug time?) over the full text, we reached a set of ten publications (summarized in Table 5).

Different terms were employed by authors to convey the idea of long-lived bugs (e.g., slowly-fixed bugs and languishing bugs; Table 5). In addition, two different approaches were used to separate software bugs with short and long life cycles. The first approach consisted of transforming the bug-fixing times into a discrete value based on quartiles using 25%(Q1), 50% (Q2 or median), or 75% (Q3) [33, 6, 8, 9, 10]. However, these authors used different names to refer to each quartile (e.g., slowly-fixed bugs or not very fast refers to Q1). The second approach consisted of employing a single threshold over a bug-fixing time to label a bug as a short- or a long-lived bug [43, 44, 12, 45, 15].

[11]`https://dl.acm.org/` (As of Nov. 2020).
[12]`https://ieeexplore.ieee.org/` (As of Nov. 2020)
[13]`https://www.sciencedirect.com/` (As of Nov. 2020).
[14]`https://www.springer.com/`(As of Nov. 2020)

Table 5: Definitions of *long-lived bugs* in the literature.

Reference	**Year**	**Definition**
Giger et al. [33]	2010	The authors defined slowly-fixed bugs as bugs that have a bug-fixing time (time between it was first opened and the date the bug status changed to *fixed*) higher than the median.
Canfora et al. [43]	2011	The authors distinguished a short-lived from a long-lived bug based on a survival function ($S(t) = Pr(T > t)$), which determines the probability that a bug survives longer than some specified time t.
Marks et al. [44]	2011	The authors classified bug-fixing times in one of three high-level classes: bugs fixed within 3 months, bugs fixed within one year, and bugs fixed within 3 years.
Abdelmoez et al. [6]	2012	The authors used the first quartile Q1 to classify bug-fixing times into very fast and not very fast, and the third quartile Q3 to classify bug-fixing times into very slow and not very slow.
Saha et al. [45, 15]	2014, 2015	The authors defined long-lived bugs as those that are not fixed within one year after they are reported.
Francis and Williams [12]	2013	The authors used the term languishing bug, which refers to defects that remain unfixed for a long period. This period, or bug age, might be an arbitrary number or it might be determined based on the historical likelihood of a bug being closed after that age; the authors used one, two, and three years. In addition to bug age, the authors also used release cycle and user interest to classify a languishing bug.
Ardimento [8]	2016	The authors classified bug reports as *fast* and *slow* based on the third quartile, or 75% of the empirical distribution of bug resolution times.
Ardimento and Dinapoli [9]	2017	The authors classified bug reports as *fast* and *slow* baed on the third quartile or 75% of the empirical distribution of bug resolution times.
Sepahvand et al. [10]	2020	The authors classified each bug report as having *short fixing times*, for bugs with fixing time less than the threshold, or *long fixing time* for all other times. The threshold used was the median of all bug fixing times.

5.2. RQ2: How frequent are long-lived bugs in FLOSS projects?

Answering the second research question is essential since there are few reasons to investigate if long-lived bug populations are small. To measure the population of long-lived bugs in the FLOSS projects, we defined a long-lived bug as one that is not fixed within one year (365 days), as proposed by Saha et al. [45, 15].

Although the threshold used could be considered subjective, it is relevant to measure populations. First, a one-year threshold is conservative and covers at least one cycle of most projects [45, 15]. Hence, we can safely classify a bug as long-lived if it survived for more than one year (threshold value). Second, such threshold (one year) enables us to compare the population of long-lived bugs independently and uniformly considering different projects. However, in our paper, we have also investigated quartile based thresholds in predicting long-lived bugs - research question RQ4.3.

The percentage value of long-lived bugs in FLOSS projects used in our study ranged from **7.2%** (in Eclipse) to **40.7%** (in WineHQ) (Figure 3).

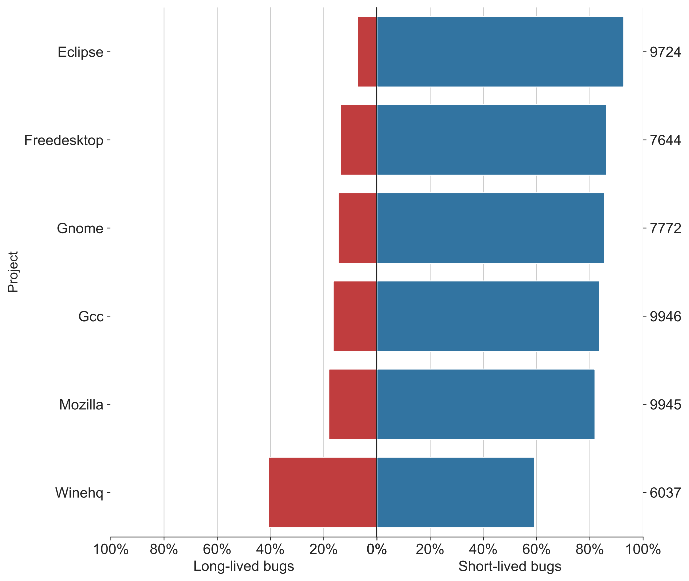


Figure 3: Proportion of long- and short-lived bugs in each FLOSS project studied considering the bug-fixing threshold of 365 days. The values in the secondary y-axis correspond to the total number of bug reports in each dataset.

Distribution of bug-fixing times retrieved from the datasets investigated is shown in Figure 4. The first quartile (25%) Q1 ranged from 1 day in Eclipse to 22 days in WineHQ. The second quartile Q2 (50%), or median, ranged from 8 days in Eclipse to 220 days in WineHQ, and the third quartile Q3 (75%) ranged from 63 days in Eclipse to 709 days in WineHQ. The percentage of outliers was ≈ 15.2% in Eclipse, ≈ 12.2% in Freedesktop, ≈ 13.3% in Gnome, ≈ 15.5%

in Mozilla, ≈ 6.1% in Gcc and WineHQ. Table 6 illustrates a few bug report outliers. The bug report can be viewed by clicking on the associated bug id.

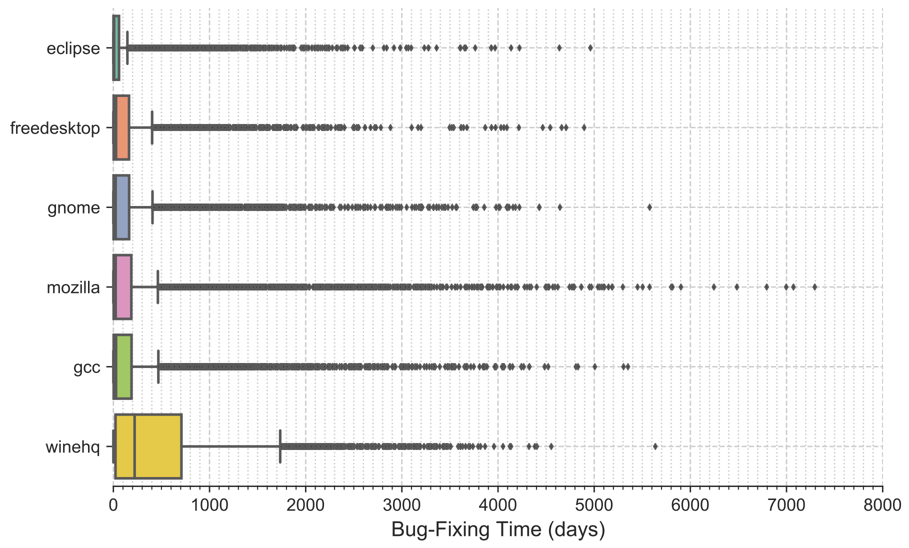


Figure 4: Distribution of bug-fixing times retrieved from the studied datasets.

Table 6: Samples of outliers found in the investigated projects.

Project	Bug Id	Created at	Closed at	Bug-fixing time (in years)
Eclipse	34970	2003-03-13	2016-10-11	13.59
Freedesktop	3952	2005-08-02	2018-12-28	13.41
Gcc	7721	2002-07-06	2017-02-28	14.66
Gnome	241477	2003-04-16	2018-07-23	15.28
Mozilla	791	1998-09-11	2018-08-31	19.98
WineHQ	1719	2003-09-17	2019-02-21	15.44

5.3. RQ3: What are the main characteristics of long-lived bugs compared to short-lived bugs in FLOSS projects?

After investigating the definition of long-lived bugs and its prevalence in FLOSS projects, we determined the bug characteristics that distinguish long- from short-lived bugs by examining the following bug report fields: bug reporters, assignees, components, severity levels, summaries, and descriptions.

5.3.1. Bug Reporter

The Pareto Chart in Figure 5a shows that the percentage of bug reports created by the top 20% bug reporters in the FLOSS projects investigated varied from **58.9%** (in Freedesktop) to **72.9%** (in Mozilla), indicating that bugs are not reported evenly by people.

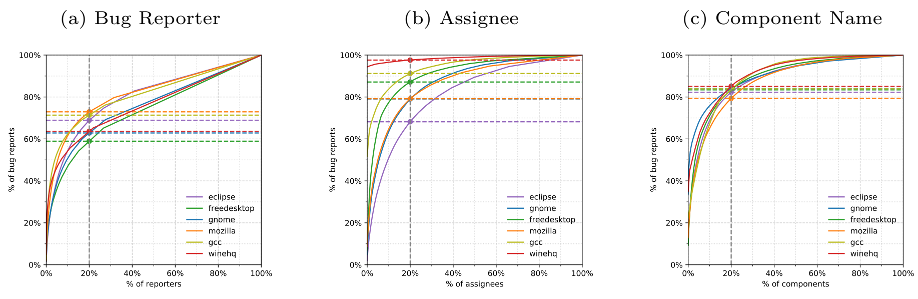


Figure 5: Percentage of bugs by (a) bug reporters, (b) assignees or (c) component names.

Figure 6 shows that the percentage of long-lived bugs for many individual bug reporters is high. For example, **ed** in Eclipse (≈ 15.4%), **freedesktop** in Freedesktop (≈ 23.5%), **dawn** in Gnome (≈ 22.6%), **timeless** in Mozilla (≈ 23.9%), **burnus** in GCC (≈ 24.2%), and **fgouget** in WineHQ (≈ 49.0%). In addition, 7 out of the 10 top bug reporters in WineHQ opened a substantial number of long-lived bugs, exceeding 30% of them.

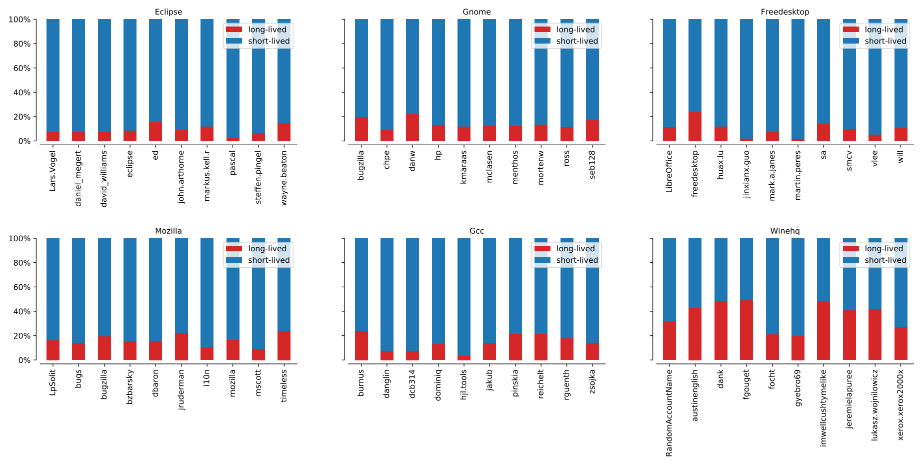


Figure 6: Percentage of long-lived bugs reported by the top 10 bug reporters considering a threshold of 365 days.

5.3.2. Assignee

Figure 5b shows that the percentage of bugs opened by the top 20% of assignees in the FLOSS projects investigated varied from **68.1%** (in Eclipse) to **97.5%** (in WineHQ), indicating that few actors are responsible for most of the bug reports.

As observed for bug reporters, many of top-10 assignees were responsible for fixing a high percentage of long-lived bugs (Figure 7): **mdt-papyrus-inbox** in Eclipse (≈ 31.8%), **nobody** in Mozilla (≈ 28.1%), **nautilus-maint** in Gnome (≈ 37.9%), **xorg-team** in Freedesktop (≈ 32.6%), **tromey** in GCC (≈ 56.0%), and **dpaun** in WineHQ (≈ 63.6%). Additionally, most assignees (8 out of 10) in WineHQ fixed more than 30% of long-lived bugs in relation to the total number bugs assigned to them. However, although some bug reports are considered to be closed, the assignee field is not updated correctly. We identified these situations differently in our datasets (Figure 7): labels ending with "inbox" in Eclipse, labels ending with "bugs" in Freedesktop, in Gnome and in WineHQ, and unassigned labels in GGC.

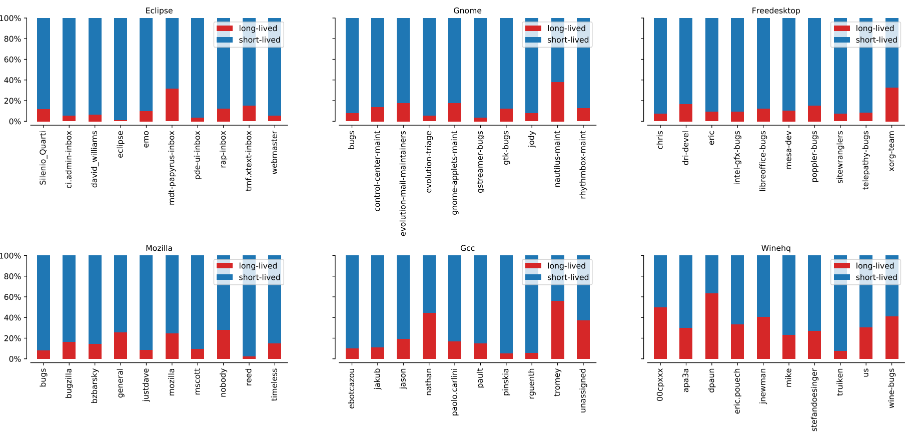


Figure 7: Percentage of long-lived bugs reported by the top-10 assignees considering a threshold of 365 days.

5.3.3. Component

Figure 5c shows that the percentage of bugs opened by the top 20% of components in the investigated FLOSS varied from **79.3%** (in Mozilla) to **85.0%** (in WineHQ), which also can be considered significant proportions.

In addition, considering all bugs reported, the percentage of long-lived bugs in projects with many components was high: **Xtext** in Eclipse (≈ 15.3%), **Contacts** in Gnome (≈ 26.1%), **Server/General** in Freedesktop (≈ 29.0%), **English US** in Mozilla (≈ 62.3%), **java** in GCC (≈ 61.0%), and **-unknown** in WineHQ (≈ 48.4%) (Figure 8). As noted for bug reporters and assignees, for WineHQ, over 30% of components (7 out of 10) had long-lived bugs.

Furthermore, we observed two important properties: (i) no long-lived bugs were reported in Eclipse, Hyades[15] and (ii) the high proportion of long-lived

[15]Hyades is an open-source platform for Automated Software Quality (ASQ) tools and a range of open-source reference implementations of ASQ tools for testing, tracking,

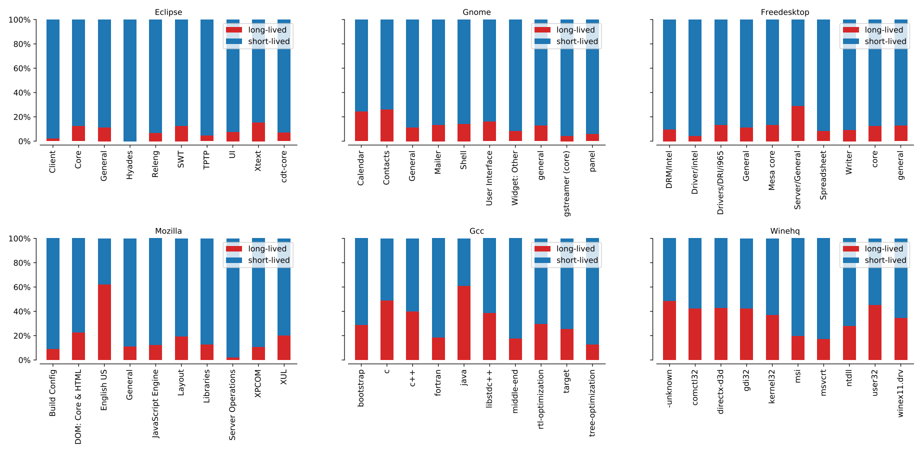


Figure 8: Percentage of long-lived bugs reported by the top-10 components considering a threshold of 365 days.

bugs in **unknown** modules of WineHQ (Figure 8).

5.3.4. Severity Level

For most projects, the minor severity level had the highest proportion of long-lived bugs, varying from ≈ 12% to ≈ 44%, except in GCC where the critical severity level had the highest proportion of long-lived bugs (≈ 35%), followed by the minor severity level with ≈ 33% (Figure 9).

5.3.5. Summary

Word clouds generated considering the summary field for all projects investigated indicate there is a group of words (such as error, fail, file, and regression) that repeatedly appear in the summary field of both short and long-lived bugs (Figure 10).

5.3.6. Description

Word clouds generated considering the description field for all projects investigated indicate there is a group of words (such as *eclipse*, *java*, *lib*, and *usr*) that repeatedly appear in the description field of both short or long-lived bugs (Figure 11). Additionally, reserved words (such as *const*, *char*, and *v_float*) and constant labels (e.g., *0x0000000000000000*) probably originated from code snippets written in a programming language (e.g., *C/C++* or *Java*) or stack traces yielded by the reported bug, which was included in the description field of the bug report.

and monitoring software systems. Online: `https://www.eclipse.org/org/press-release/apr152003uml2pr.html` (As of Jan. 2020).

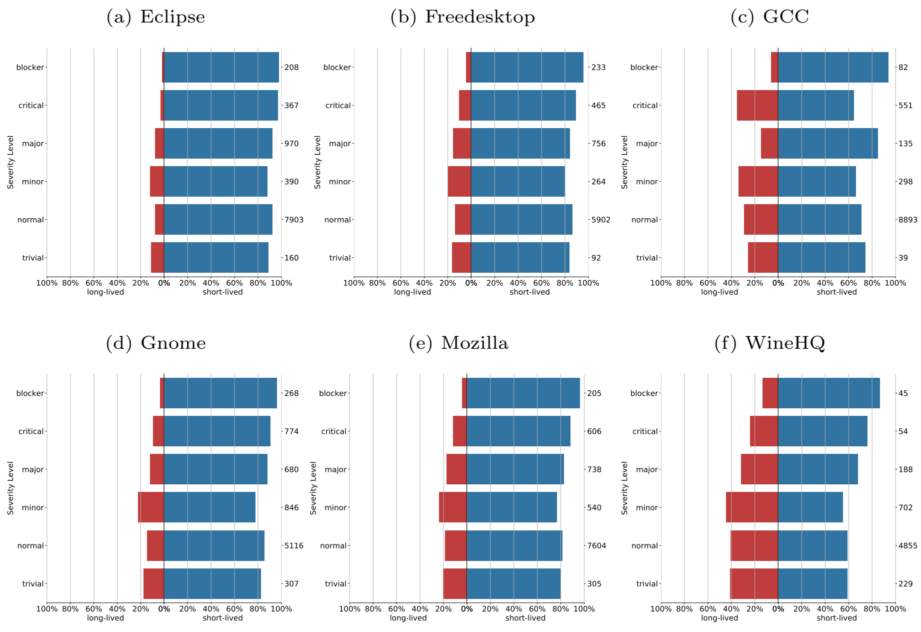


Figure 9: Percentage of long-lived and short-lived bugs by severity levels.

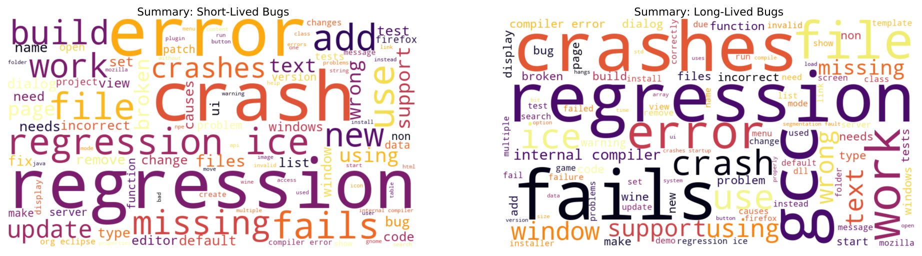


Figure 10: Word clouds based on the summary field of bug reports for all projects investigated

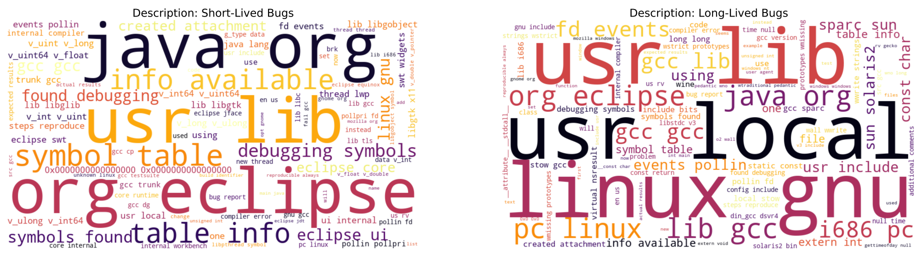


Figure 11: World clouds based on the description field of bug reports for all investigated projects.

5.4. RQ4: What is the comparative accuracy of machine learning algorithms when predicting long-lived bugs?

This question focused on comparing the performance accuracy of traditional ML algorithms when predicting long-lived bugs, namely *KNN*, *Naïve Bayes*, *Neural Networks*, *Random Forest*, and *SVM*. We conducted four experiments, changing specific parameters before executing each one.

5.4.1. RQ4.1. Evaluating the performance of ML algorithms when predicting long-lived bugs

In the first experiment, we evaluated the performance of selected ML algorithms when predicting long-lived bugs using the Eclipse dataset and the parameters shown in Table 7. For each combination of these parameters, every algorithm builds a prediction model based on features extracted from summaries or descriptions of bug reports. Moreover, to select the best model throughout the repeated cross-validation process, each algorithm computed metrics based on Accuracy.

Table 7: Parameters for RQ4.1 experiment.

Parameter	**Value**
Dataset	Eclipse
Bug report fields	Summary and Description
Number of observations	9724
Predictive algorithms	KNN, NB, NN, SVM, and RF
Number of terms	100
Metrics to select model	Accuracy, Kappa and AUC
Balancing methods	Unbalanced and SMOTE
Sampling method	Repeated CV 5×10
Bug fixing time threshold	365 days

The best balanced accuracy scores of ML algorithms used to predict long-lived bugs for the Eclipse project dataset are shown in Table 8. The evaluated methods were sorted in descending order according to their balanced accuracy. The **SVM** algorithm yielded the best performance accuracy ($\approx 98\%$ of balanced accuracy). This model used the **description** field to generate the text feature vector, the **SMOTE** method to balance the dataset, and the **accuracy** metric to select the best model (Table 8).

Considering the bug fixing time threshold of 365 days, the Eclipse dataset is notably unbalanced (Figure 3). Therefore, we ran each algorithm over the same unbalanced dataset or over a balanced dataset built applying the SMOTE method onto the original dataset. The frequency of long-lived bugs increased from 7.85% (601/8252) to 48.89% (8352/17081) (Table 8). Furthermore, we can clearly observe that the balancing of datasets yielded better predictions of long-lived bugs.

Table 8: The best balanced accuracy scores of ML algorithms used to predict long-lived bugs for the Eclipse project dataset.

Algorithm	Feature	Balancing Method	Sensitivity	Specificity	Balanced Accuracy	Total Bugs	Short-lived Bugs	Long-lived Bugs
svm	description	smote	0.997	0.966	0.981	17081	8729	8352
rf	description	smote	0.955	0.948	0.951	17081	8729	8352
rf	summary	smote	0.910	0.904	0.907	14863	7651	7212
svm	summary	smote	0.941	0.834	0.888	14863	7651	7212
nn	summary	smote	0.907	0.767	0.837	14863	7651	7212
knn	summary	smote	0.853	0.751	0.802	14863	7651	7212
knn	description	smote	0.999	0.564	0.782	17081	8729	8352
nn	description	smote	0.814	0.675	0.744	17081	8729	8352
nb	description	smote	0.832	0.234	0.533	17081	8729	8352
nb	summary	smote	0.814	0.234	0.524	14863	7651	7212
rf	summary	unbalanced	0.011	0.994	0.503	8252	7651	601
rf	description	unbalanced	0.004	0.998	0.501	9425	8729	696
nn	summary	unbalanced	0.002	0.999	0.501	8252	7651	601
knn	description	unbalanced	0.000	1.000	0.500	9425	8729	696
svm	description	unbalanced	0.000	1.000	0.500	9425	8729	696
knn	summary	unbalanced	0.000	1.000	0.500	8252	7651	601
svm	summary	unbalanced	0.000	1.000	0.500	8252	7651	601
nn	description	unbalanced	0.000	1.000	0.500	9425	8729	696
nb	description	unbalanced	0.326	0.628	0.477	9425	8729	696
nb	summary	unbalanced	0.273	0.665	0.469	8252	7651	601

We used a Wilcoxon signed-rank test to check if the classifiers trained with the Eclipse dataset were significantly different (Table 9). Each cell shows the results of a single classifier paired with the other. Left ('←') and up ('↑') arrows indicate the most accurate, while an empty cell refers to "no statistical difference between the pairs of classifiers" in that row and column.

Table 9: Statistical test for the balanced accuracy of classifiers.

classifier	knn	nb	nn	rf	svm
knn	–	←	–	↑	↑
nb	↑	–	↑	↑	↑
nn	–	←	–	↑	↑
rf	←	←	←	–	↑
svm	←	←	←	←	–

5.4.2. RQ4.2. Evaluating the impact of the number of terms used in the feature vector on ML algorithm performance

To answer this question, we used the same parameters as before, but changed the number of terms (we used 100, 150, 200, 250, and 300; Table 10).

Table 10: Parameters for the RQ4.2 experiment.

Parameter	**Value**
Dataset	Eclipse
Bug report field	Description
Number of observations	9724
Predictive algorithm	SVM
Number of terms	100, 150, 200, 250, 300
Metric to select model	Accuracy
Balancing method	SMOTE
Sampling method	Repeated CV 5 $\times$ 10
Bug fixing time threshold	365 days

The sensitivity, specificity, and balanced accuracy scores for the SVM with different numbers of terms in the text feature vector varied slightly (Figure 12).

The SVM algorithm yielded the best-balanced accuracy ($\approx$ 98.9%) when the text mining process extracted **150 highest TD-IDF scored terms** of the description field to build the text feature vector. Furthermore, considering that we need a model that predicts the positives better (long-lived bugs), we might want to take into account the vector with 150 features, given its better sensitivity, as shown in Figure 12.

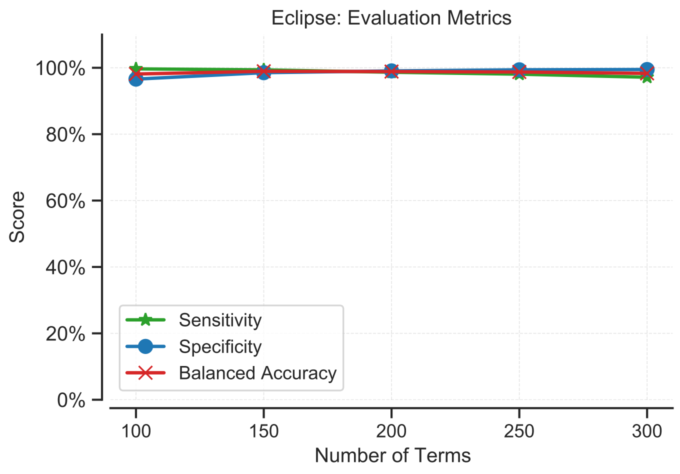


Figure 12: Sensitivity, specificity, and balanced accuracy scores using the SVM to predict long-lived bugs with the Eclipse dataset in relation to the number of terms used in the feature vector.

5.4.3. RQ4.3. Evaluating the impact of bug-fixing time threshold on ML algorithm performance

Determining the bug-fixing time threshold, which separates a short-lived from a long-lived bug, is necessary to evaluate the performance of SVM in predicting long-lived bugs. Thus, we changed the bug fixing time thresholds to 8, 63, 108, and 365 days, which refer to the median (as suggested by Giger et al. [33]), the third quartile, the mean bug-fixing time from the Eclipse dataset, and the threshold value (as suggested by Saha et al. [15], respectively; Table 11).

Table 11: Parameters for RQ4.3 experiment.

Parameter	**Value**
Dataset	Eclipse
Bug report fields	Description
Number of observations	9724
Predictive algorithms	SVM
Number of terms	150
Metric to select model	Accuracy
Balancing method	SMOTE
Resampling method	Repeated CV 5×10
Bug fix time threshold	8, 63, 108, and 365 days

The balanced accuracy, sensitivity, and specificity varied sharply for SVM with different bug-fixing time thresholds (Figure 13), suggesting that the SVM algorithm yielded the best balanced accuracy ($\approx$ 98.9%) when the bug-fixing time threshold is 365 days.

5.4.4. RQ4.4. Evaluating the performance of ML algorithms using other FLOSS datasets

We partitioned the preprocessed datasets into two disjoint subsets: a subset for training, with 75% of the bug reports, and a subset for testing, with the remaining 25% of the bug reports. We did so in order to investigate the characteristics of bugs with high and low accuracy. For such, we evaluated the balanced accuracy performance of all algorithms studied, with the experimental parameters in Table 12, over Eclipse, Freedesktop, GCC, Gnome, Mozilla, and WineHQ datasets.

Figure 14 shows the balanced accuracy performance of ML algorithms over all projects considering its balanced datasets with SMOTE. We can observe that the Neural Network (NN) was the best in three datasets: **55.4%** in Freedesktop, **70.7%** in GCC, and **61.8%** in Mozilla. Also, we can see that the Neural Network was slightly worse than KNN in Eclipse (**54.2**% versus **54.9**%) and Gnome (**59.4**% versus **59.9**%), and than Random Forest (RF) (**56.4**% versus **57.4**) in WineHQ.

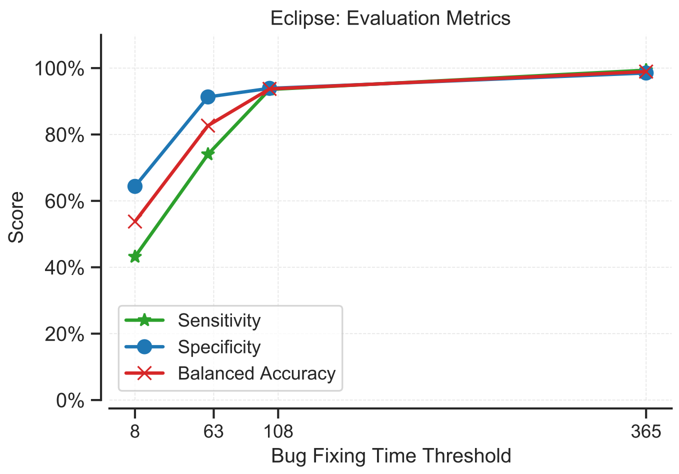


Figure 13: Sensitivity, specificity, and balanced accuracy scores from SVM predicting long-lived bugs using the Eclipse dataset for different bug-fixing time thresholds.

Table 12: Parameters for RQ4.4 experiment.

Parameter	**Value**
datasets	Eclipse, Freedesktop, GCC, Gnome, Mozilla, and WineHQ
Bug report fields	Description
Number of observations	9724
Predictive algorithms	KNN, NB, NN, RF, SVM
Number of terms	100
Metric to select model	Accuracy
Balancing methods	Smote
Resampling method	Repeated CV 5×10
Bug fix time threshold	365

In addition, Figure 15 presents the evolution of balanced accuracy according to percentage of long-lived bugs in each dataset. There seems to be a linear relationship between these two variables in all datasets, except for the WineHQ dataset. WineHQ shows unique characteristics that might affect the performance of ML algorithms, such as it has the lowest number of assignees, the highest median bug-fixing time, the highest Q1 and Q3 interquartile distance (Figure 4), and the lowest quantity of bug reports.

5.5. RQ5: What are the main characteristics of bugs correctly predicted as long-lived and incorrectly predicted as short-lived?

Even if our classifier did not identify many long-lived bugs, investigating the characteristics of algorithms with high and low accuracy could be very useful

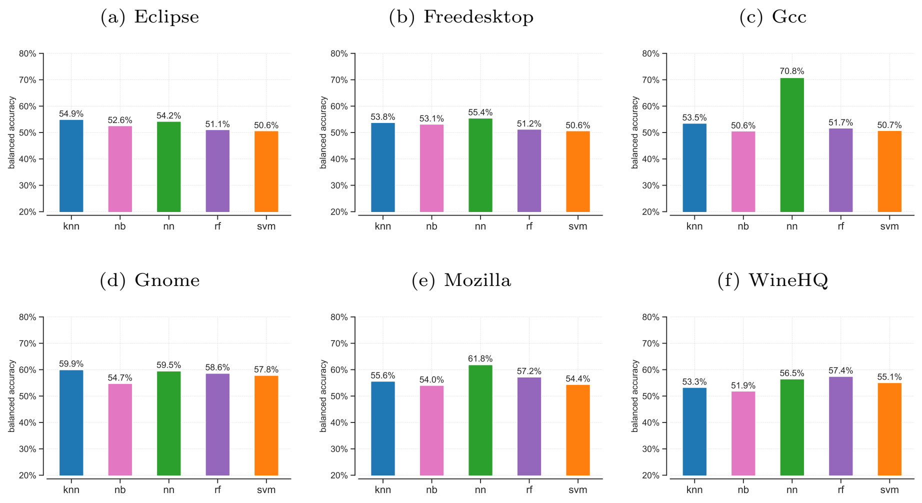


Figure 14: Balanced accuracy performance for all classifiers over all datasets projects.

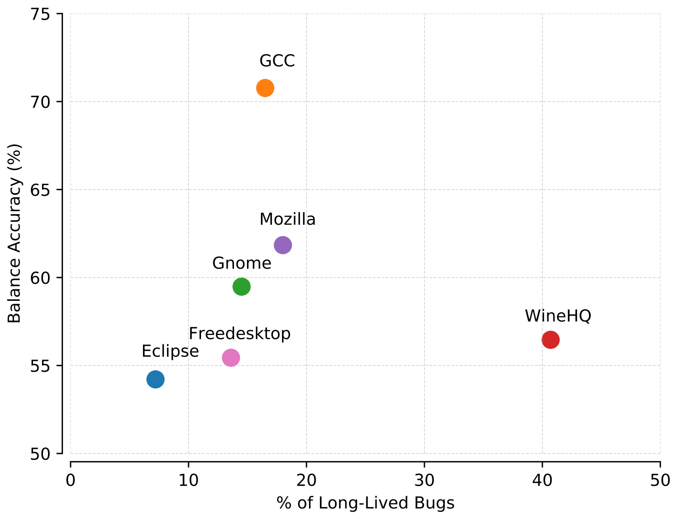
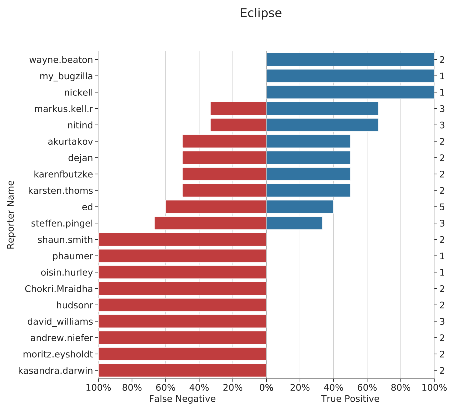


Figure 15: Balanced Accuracy versus Percentage of Long-Lived Bugs.

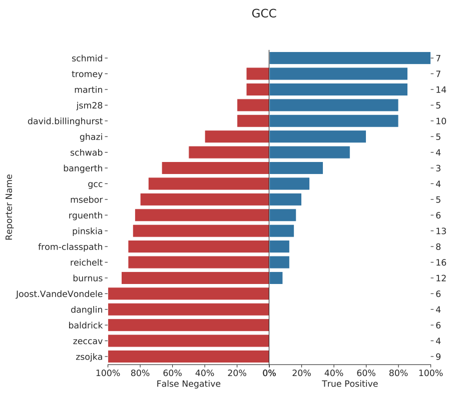
for developing teams. Thus, we investigated the characteristics of true positives and false negatives from some perspectives: bug reporter, assignee, software component, and description attributes. We limited our investigation to the Eclipse and GCC dataset because the Eclipse dataset yielded the worst accuracy and the GCC dataset yielded the best accuracy.

5.5.1. *Bug Reporter*

Our results showed that for **3 out of 20** bug reporters in Eclipse and **1 out of 20** in GCC, the prediction yielded in 100% true positives, which was the case for *wayne.beaton* in Eclipse and *schimd* in GGC. On the other hand, for a fair amount of bug reporters (**9 out of 20** in Eclipse, and **5 out of 20** in GCC), the prediction resulted in 100% false negative - the case of *shaun.smith* in Eclipse and *Joost.VandeVondele* in GCC (Figure 16).



(a)



(b)

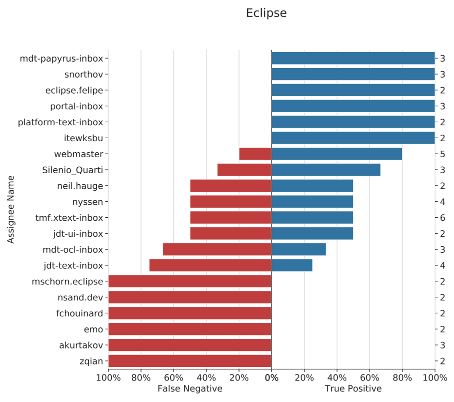
Figure 16: True positives/false negatives in (a) Eclipse and (b) GCC datasets for different bug reporters.

5.5.2. *Assignee*

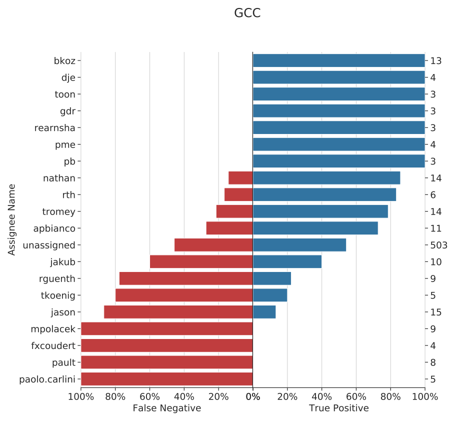
Our results showed that for **6 out of 20** assignees in Eclipse and **7 out of 20** in GCC, the long-lived prediction yielded in 100% true positives for all bugs (*snorthov* in Eclipse and *bkoz* in GGC). Similarly, for **6 out of 20** assigneees in Eclipse and **1 out of 20** in GCC, the prediction yielded in 100% false negatives for all bugs (*akurtakov* in Eclipse and *pault* in GCC). In addition, we found a significant number of unassigned bugs - **27 out of 57 (≈ 47%)** in Eclipse and **503 out of 648 in GCC (≈ 77%)** (Figure 17).

5.5.3. *Component*

Our results showed that for **3 out of 20** components in the Eclipse dataset and **1 out of 20** in the GCC dataset, the long-lived prediction yielded in 100% true positives — for example, *Diagram* in Eclipse and *pending* in GGC. Similarly, for **3 out of 20** in Eclipse and **2 out of 20** in GCC, the prediction yielded in 100% false negatives — for example, *cdt-core* in Eclipse and *testsuite* in GCC. Furthermore, we found that true positives and false negatives were balanced for most components in both datasets (Figure 18).

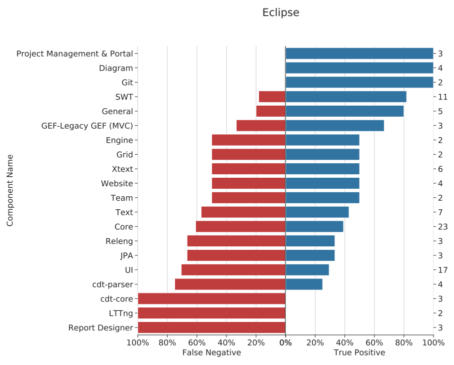


(a)

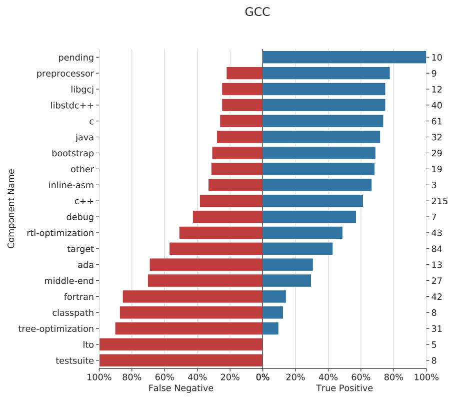


(b)

Figure 17: True positives/false negatives in the (a) Eclipse and (b) GCC dataset for different assignees.



(a)



(b)

Figure 18: True positives/false negatives in the (a) Eclipse and (b) GCC dataset for different components.

5.5.4. Severity Level

Our results showed that the long-lived predictor yielded 100% true positives only for the critical severity level in Eclipse. Conversely, in both datasets, our predictor yielded 100% false negatives for the blocker, major, and trivial severity levels. Furthermore, we can notice a high number of bugs classified with a normal severity level, **156 out of 187 (≈ 83%)** in Eclipse and **629 out of 724 in the GCC (≈ 86%)** datasets (Figure 19).

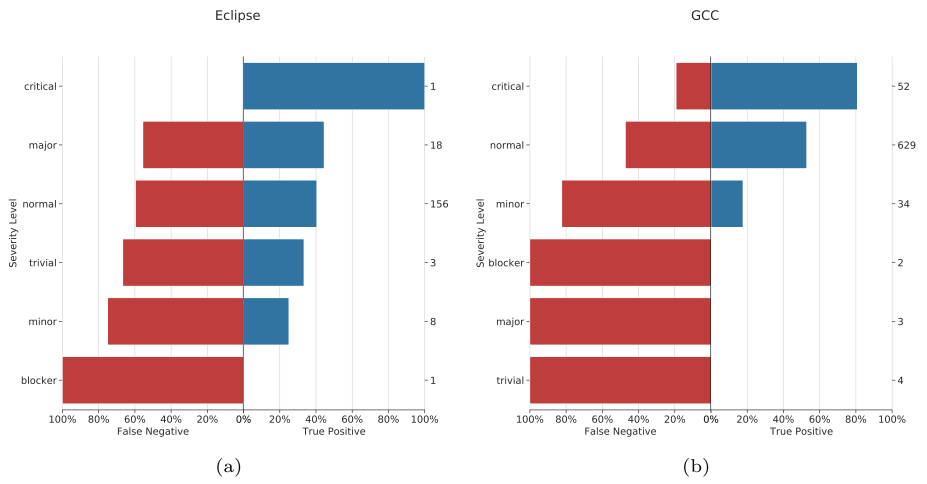


Figure 19: True positives/false negatives in the (a) Eclipse and (b) GCC dataset for different severity levels.

5.5.5. Description

The words used to create the feature vector according to the description field is shown in Figure 20. The words in panels (a) and (c) yielded true positives and in panels b and d, false positives for both the Eclipse and GCC datasets. Similar word colors indicate similar values of TF-IDF, while word size indicates the amplitude of the TD-IDF value (for instance, larger word size indicates higher values of TD-IDF).

Figure 21 refers to the lollipop chart for the feature vector created according to the description field of bug reports. The words in panels (a) and (c) yielded true positives and in b and d yielded false negatives using the Neural Network algorithm to predict long-lived bugs in the Eclipse and GCC datasets. The y-axis refers to TF-IDF, while the x-axis refers to the words in the feature vectors.TF-IDF values of a few terms varied greatly for words that yielded false negatives (only three terms for the Eclipse dataset and one for the GCC dataset), while the variation seemed smaller for terms that yielded true positives.

In both datasets, the sparsity ratio of the feature vector matrices, which was based on the 100 most TD-IDF weighted terms, is very high for both *true positive* bugs and *false negative* bugs (Figure 22). Nevertheless, the sparsity ratio for *true positives* and *false negatives* in the GCC dataset is slightly smaller in relation to the Eclipse dataset.

6. Discussions

This section discusses the significance of our findings and it is divided by research question as the results section.

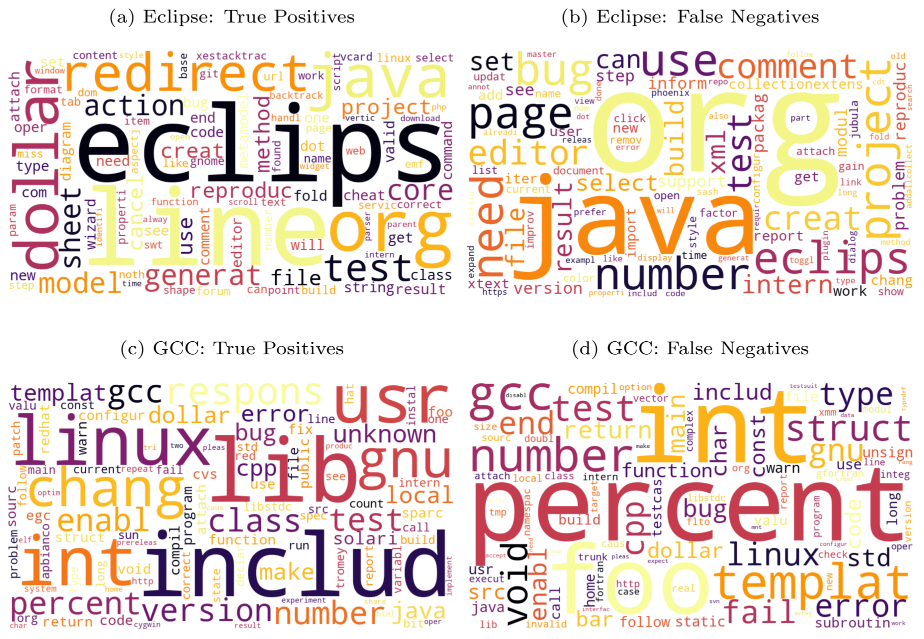


Figure 20: Word clouds used to create the feature vectors used in prediction testing according to the description field of bugs reports that yielded (a) true positives in Eclipse, (b) false negatives in Eclipse, (c) true positives in GCC, and (d) false negatives in GCC.

6.1. *RQ1: Is there a common understanding of what long-lived bugs are in the literature?*

Table 5 indicates two distinct points of view about long-lived bugs. While some authors [43, 44, 45, 15] consider absolute values as thresholds (based on the release cycle), others [6, 33, 12, 8, 9, 10] consider threshold values based on the statistical distribution of bug-fixing times for each FLOSS project. These different visions suggest there is no shared understanding of what long-lived bugs represent. Furthermore, we believe the definition of the long-live threshold is related to particular characteristics of each project (e.g., number of people in the team) and, therefore, each development team should choose the more suitable threshold for their project.

6.2. *RQ2: How frequent are long-lived bugs in FLOSS projects?*

Saha et al. [15] found that long-lived bugs accounted for 5% to 9% bugs in Java Projects (Eclipse) and for 14% to 37% in C projects (GCC and WineHQ). We investigated three other FLOSS projects (Freedesktop, Gnome, and Mozilla) and our results corroborate the findings obtained by these authors (Figure 3).

The percentage of long-lived bugs is higher in projects with fewer assignees (e.g., WineHQ). This can be observed by examining the bug-fixing time quartile (Figure 4) and the assignee demographic (Figure 5b). Our results also showed

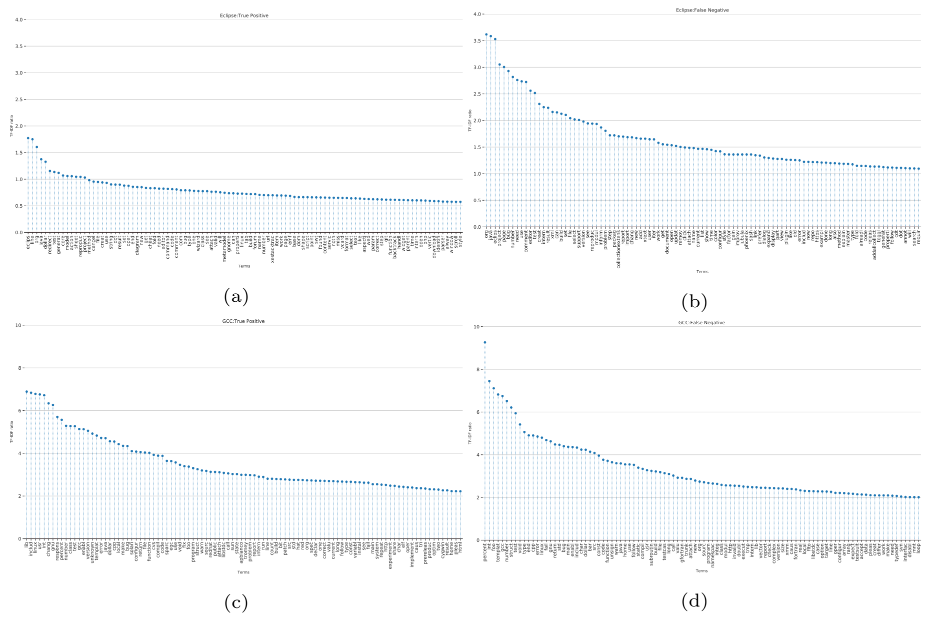


Figure 21: Lollipop chart for the feature vectors created based on the description field of bug reports: (a) true positives in Eclipse, (b) false negatives in Eclipse, (c) true positives in GCC, and (d) false negatives in GCC.

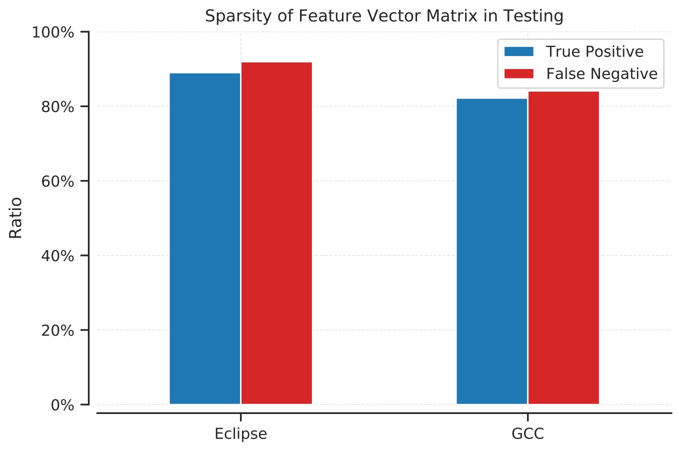


Figure 22: Sparsity of true positive and false negative bugs for feature vectors based on the description field in the Eclipse and GCC datasets.

a high number of outliers in most projects (Eclipse, Freedesktop, Gnome, and Mozilla), which may indicate either a significant number of bugs with very long bug-fixing time or a delay in updating the status resolution of bugs.

6.3. RQ3: What are the main characteristics of long-lived bugs compared to short-lived bugs in FLOSS projects?

6.3.1. Bug reporters

The top 20% bug reporters opened more than half of the bugs reports in the projects we investigated (Figure 5a) and, in some cases, they opened more than 70% of bug reports. We expected that more experienced reporters would have provided more accurate information in their bug reports, which would have enabled the development team to better evaluate and schedule these bugs. However, Figure 6 shows otherwise, as a significant percentage of long-lived bugs was associated with the top-10 bug reporters.

6.3.2. Assignees

Similarly, the top 20% assignees were in charge of fixing most of the reported bugs (68% – 97%), which may be related to the availability of human resources. Indeed, the Eclipse project, which is well-known and supported by big companies, had the lowest rate of bugs assigned to the top 20% assignees, while WineHQ, a voluntary project, had the highest rate. Thus, the lack of human resources seems to affect the number of long-lived bugs in certain projects (Figure 7).

6.3.3. Components

We expected that large projects with more components would have a smaller concentration of bug reports than small projects with fewer components. However, most bugs were reported for specific components (top 20% of the total number of components) regardless of the project size (nearly 80% in Mozilla – a large project – and 85% in WineHQ – relatively smaller), indicating that bugs were not evenly distributed but rather concentrated in certain components (Figure 5c). This suggests the existence of a set of complex components that holds a considerable number of long-lived bugs (Figure 8).

6.3.4. Severity Level

Bug reports labeled as having minor severity levels prevailed in long-lived bugs in all projects, except for the GCC project (see Figure 9). This suggests that the development team considered the severity level when deciding which bug would be fixed first, and bugs labeled as more severe were given priority. However, the percentage of long-lived bugs with trivial severity level (considered less severe than minor severity level) was smaller than long-lived bugs with minor severity level, contrasting with our initial analysis. Furthermore, the percentage of long-lived bugs that had been labeled with critical and major severity levels was relatively high, suggesting that information on bug severity level was not used by the development team to distinguish a short-lived bug from a long-lived.

6.3.5. Summary

All bug report attributes previously analyzed were structured, while the summary attribute is textual. Figure 10(a)-(b) presents a preliminary analysis of this attribute using word clouds with 100 terms (one for short-lived and other for long-lived bugs), each generated from the bug report summary of all projects. From the word clouds we created for short- and long-lived bugs, we were able to determine words in common for both of them, indicating the feature vectors would be very similar. Thus, building a model that accurately predicts long-lived bugs based on this attribute would be quite challenging.

6.3.6. Description

As the summary attribute mentioned above, the description attribute is also textual. Figure 11(a)-(b) presents a preliminary analysis of the description attribute using word clouds with 100 terms (one for short-lived and other for long-lived bugs), each generated from all projects. Similarly to the summary attribute, the word cloud for the description attribute also shows many words in common for both short- and long-lived bugs. We can observe in word clouds for the description attribute many reserved words of programming languages like int, extern, using, and lib. Thus, being able to handle these words properly is also essential to build models that can accurately predict long-lived bugs.

6.4. RQ4: What is the comparative accuracy of machine learning algorithms when predicting long-lived bugs?

6.4.1. RQ4.1: Evaluating the performance of ML algorithms when predicting long-lived bugs.

When comparing the traditional ML algorithms in predicting long-lived bugs using a textual feature vector, we observe that the performance of algorithms is very similar, but the SVM algorithm is slightly better (see Table 8).

6.4.2. RQ4.2: Evaluating the impact of the number of terms used in the feature vector on ML algorithm performance.

Figure 12 indicates that increasing the number of terms used in the feature vector did not significantly affect the balanced accuracy of the SVM algorithm when predicting a long-lived bug. The best-balanced accuracy (using terms with the 150 highest weight values) and the worst-balanced accuracy (using terms with 150 highest weight values) differed only by 0.8%. Therefore, the terms that obtained the best accuracy could be considered the most relevant terms needed to distinguish a short- from a long-live bug in our current scenario. Nonetheless, we recommend the feature vector to contain the lowest number of terms with the best performance (in our case 150 terms), given the accuracy is not affected by other factors, such as time processing and memory footprint.

6.4.3. RQ4.3: Evaluating the impact of bug-fixing time threshold on ML algorithm performance.

The median bug-fixing time equally binned bug reports into short- and long-lived, i.e., bugs with a bug-fixing time less than or equal to the median were considered short-lived bugs, all others were considered long-lived bugs. We expected that the median would be the most suitable threshold value because the binning process produces a balanced dataset. However, Figure 13 shows the threshold based on the median yielded the worst balanced accuracy and sensibility. On the other hand, the highest balanced accuracy was obtained using the bug-fixing time threshold of 365 days, which was suggested by Saha et el. [15] based on the release cycle of the Eclipse, GCC, and WineHQ projects. Thus, we believe there are patterns among the features vectors (boosted by the SMOTE method) that better enabled the predictor to distinguish a long- from a short-lived bug.

6.4.4. RQ4.4: Evaluating the performance of ML algorithms using other FLOSS datasets

Figure 14 clearly shows that the predictive model performed better compared to a random prediction (50% probability of predicting a long-lived bug) [33]. The figure shows that the predictive model yielded greater balanced accuracy for the GCC dataset using Neural Network in comparison to other algorithms in other datasets.

Curiously, Figure 15 shows that the balanced accuracy performance increased as the number of long-lived bugs increased, except for WineHQ. These results suggest a relationship between the number of long-lived bugs and the performance of ML algorithms. Such correlation may indicate that there is a specific proportion of long-lived bugs needed to train a predictive model with good performance. The performance of our automated model is even more surprising when considering that our prediction efforts rely only on two attributes: summary and description.

6.5. RQ5: What are the main characteristics of bugs correctly predicted as long-lived and incorrectly predicted as short-lived?

6.5.1. Bug Reporter

Our long-lived bug predictor based on the Neural Network algorithm had a high rate of true positives and false negatives for specific bug reporters. Although the feature vector did not include the description field, our results suggest that certain bug reporters provided descriptions with characteristic patterns that enabled a more accurate classification of long-lived bugs. However, for other bug reporters, the descriptions given did not seem to affect the classification of long-lived bugs, leading to a high rate of false negatives.

6.5.2. Assignee

Similarly, for certain assignees, the long-lived bug predictor yielded a high rate of true positives or false negatives. On one hand, this suggests that assignees correctly evaluated the terms in the description field (high rate of true positives); on the other hand, it also suggests that the terms in the description field might not be related to the time an assignee takes to fix a bug. Another possible reason for the number of false negatives could be the delay in allocating assignees for any given bug or even not assigning a person at all, as occurred in the Eclipse and GCC projects.

6.5.3. Component

The test results using the Eclipse and GCC datasets yielded 100% true positives for certain components. These components may be those that often take a long time to improve or correct for unknown reason. However, the description given in bug reports (with specific patterns of terms) could be used in alerting the maintenance team about possible long-lived cycles. In addition, our results also showed components with 100% false negatives, indicating that the description given in these bug reports failed to characterize long-lived bugs.

6.5.4. Severity Level

The description of bugs with the blocker severity level should contain terms that allow them to be classified as long-lived bugs. As the highest level of severity, the development team most likely wishes to quickly identify blocker long-lived bugs. However, bug reporters failed to provide patterns of terms that characterize blocker long-lived bugs for both the Eclipse and GCC projects.

6.5.5. Description

The word clouds extracted from long-lived bug descriptions that yielded *true positives* and *false negatives* exhibited a great number of words in common, many of which are words of programming languages, suggesting the presence of many pieces of code into bug report descriptions. However, the TF-IDF weight distributions of words in each cloud (true positives and false negatives) are somewhat different: while the distribution of TF-IDF values for the *true positive* feature vectors was balanced, the distribution for the *false negatives* feature vectors was unbalanced, which could be related to the sparsity of the feature vector matrix.

7. Threats to Validity

The main threats to validity in this study are summarized below:

- We have assumed that bug fix times extracted from repositories are correct and that there is a close relationship between the time to fix a bug and the short and long descriptions typically found in bug reports (this correlation was investigated elsewhere [5]);

- We have considered six repositories, which lead to a collection composed of 51,770 bug reports. Although we cannot generalize the results to other datasets, the characteristics presented by Eclipse, Freedesktop, GCC, Gnome, Mozilla, and WineHQ repositories, particularly regarding the balance of the data, are similar to those shown in other studied repositories [33, 34, 15];

- Another limitation refers to the lack of investigation regarding the impact of different text mining techniques on the balanced accuracy of the ML algorithms studied here.

8. Conclusion

We did not find evidence that there is a shared understanding of what a long-lived bug is (RQ_1). Using a conservative bug fixing time threshold (of 365 days), we confirmed that a significant percentage of bugs in the FLOSS projects studied here were long-lived bugs (RQ_2). An analysis of bug report fields, namely bug reporter, assignee, component, severity level, summary, and description, showed that there were some differences between the population of long-lived and short-lived bugs. For example, when the assignee field was analyzed, we found an apparent shortage of resources in projects with the greatest number of long-lived bugs (RQ_3). We also compared the performance of popular ML algorithms to predict long-lived bugs and found that Support Vector Machines were better than the others in accurately predicting long-lived bugs for the Eclipse dataset ($RQ_{4.1}$).

Furthermore, our results have shown that the number of terms in the feature vector had no significant impact in the accuracy of classifiers when predicting long-lived bugs ($RQ_{4.2}$). We confirmed that 365 days is a bug fixing time threshold that enables the classifier algorithms to yield a good balanced accuracy. This threshold is longer than the bug fix time for most bugs with a short lifecycle ($RQ_{4.3}$).

Finally, we showed that the differences between long-lived bugs correctly predicted as long-lived bugs (true positives) and long-lived bugs incorrectly predicted as short-lived bugs (false negative) are related to the terms used in the feature vectors and to the sparsity of the feature vector matrix (RQ_5).

Given the neural network yielded good results ($RQ_{4.4}$), we believe that investigating other learning algorithms (such as deep learning) could be useful in predicting long-lived bugs [46]. In addition, analyzing different bug report fields (e.g., bug reporter, assignee, and component) – $RQ_{4.4}$ – and including other data based on software engineering metrics (for example, the complexity of feature vectors and the temporal evolution of attributes in the feature matrix [3]) would also improve our understanding of the topic. Moreover, we also intend to investigate other repositories and other BTS and thus develop an approach that represents BTS data generally and uniformly as this would facilitate the development of a general-purpose ML-based long-lived bug prediction assistant.

Acknowledgments

The authors thank the National Council for Scientific and Technological Development - CNPq (grant #307560/2016-3), the São Paulo Research Foundation – FAPESP (grants #2014/12236-1, #2015/24494-8, #2016/50250-1, and #2017/20945-0), the FAPESP-Microsoft Virtual Institute (grants #2013/50155-0 and #2014/50715-9), and The Pontifical Catholic University of Minas Gerais (PUC-MG) that supports the first author through the Permanent Program For

Professor Qualification (PPCD). This study was partially funded by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

References

[1] A. Lamkanfi, S. Demeyer, E. Giger, B. Goethals, Predicting the severity of a reported bug, in: 2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010), 2010, pp. 1–10. doi:10.1109/MSR.2010.5463284.

[2] Y. C. Cavalcanti, P. A. da Mota Silveira Neto, I. d. C. Machado, T. F. Vale, E. S. de Almeida, S. R. d. L. Meira, Challenges and opportunities for software change request repositories: a systematic mapping study, Journal of Software: Evolution and Process 26 (7) (2014) 620–653. doi:10.1002/smr.1639.

[3] I. Sommerville, Software Engineering, Pearson, 2010. doi:10.1111/j.1365-2362.2005.01463.x.

[4] G. Yang, S. Baek, J.-W. Lee, B. Lee, Analyzing emotion words to predict severity of software bugs: A case study of open source projects, in: Proceedings of the Symposium on Applied Computing, SAC '17, ACM, New York, NY, USA, 2017, pp. 1280–1287. doi:10.1145/3019612.3019788.

[5] H. Zhang, L. Gong, S. Versteeg, Predicting bug-fixing time: An empirical study of commercial software projects, in: 2013 35th International Conference on Software Engineering (ICSE), 2013, pp. 1042–1051. doi:10.1109/ICSE.2013.6606654.

[6] W. Abdelmoez, M. Kholief, F. M. Elsalmy, Bug fix-time prediction model using naïve bayes classifier, in: 2012 22nd International Conference on Computer Theory and Applications (ICCTA), 2012, pp. 167–172.

[7] W. H. A. Al-Zubaidi, H. K. Dam, A. Ghose, X. Li, Multi-objective search-based approach to estimate issue resolution time, in: Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering, PROMISE, Association for Computing Machinery, New York, NY, USA, 2017, p. 53–62. doi:10.1145/3127005.3127011. URL https://doi.org/10.1145/3127005.3127011

[8] P. Ardimento, M. Bilancia, S. Monopoli, Predicting bug-fix time: Using standard versus topic-based text categorization techniques, 2016, pp. 167–182. doi:10.1007/978-3-319-46307-0_11.

[9] P. Ardimento, A. Dinapoli, Knowledge extraction from on-line open source bug tracking systems to predict bug-fixing time, in: Proceedings of the 7th International Conference on Web Intelligence, Mining and Semantics,

WIMS '17, Association for Computing Machinery, New York, NY, USA, 2017. doi:10.1145/3102254.3102275.
URL https://doi.org/10.1145/3102254.3102275

[10] R. Sepahvand, R. Akbari, S. Hashemi, Predicting the bug fixing time using word embedding and deep long short term memories, IET Software 14 (3) (2020) 203–212.

[11] C. Liu, J. Yang, L. Tan, M. Hafiz, R2fix: Automatically generating bug fixes from bug reports, in: 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, 2013, pp. 282–291.

[12] P. Francis, L. Williams, Determining "grim reaper" policies to prevent languishing bugs, in: 2013 IEEE International Conference on Software Maintenance, 2013, pp. 436–439.

[13] S. Akbarinasaji, B. Caglayan, A. Bener, Predicting bug-fixing time: A replication study using an open source software project, Journal of Systems and Software 136 (2018) 173 – 186. doi:https://doi.org/10.1016/j.jss.2017.02.021.

[14] B. S. Rawal, A. K. Tsetse, Analysis of bugs in google security research project database, in: 2015 IEEE Recent Advances in Intelligent Computational Systems (RAICS), 2015, pp. 116–121.

[15] R. K. Saha, S. Khurshid, D. E. Perry, Understanding the triaging and fixing processes of long lived bugs, Information and Software Technology 65 (2015) 114 – 128. doi:https://doi.org/10.1016/j.infsof.2015.03.002.

[16] M. E. Mezouar, F. Zhang, Y. Zou, Are tweets useful in the bug fixing process? an empirical study on firefox and chrome, Empirical Softw. Engg. 23 (3) (2018) 1704–1742. doi:10.1007/s10664-017-9559-4.
URL https://doi.org/10.1007/s10664-017-9559-4

[17] L. A. F. Gomes, R. da Silva Torres, M. L. Côrtes, Bug report severity level prediction in open source software: A survey and research opportunities, Information and Software Technology 115 (2019) 58–78. doi:https://doi.org/10.1016/j.infsof.2019.07.009.

[18] Y. Tian, D. Lo, C. Sun, Information retrieval based nearest neighbor classification for fine-grained bug severity prediction, in: 2012 19th Working Conference on Reverse Engineering, 2012, pp. 215–224. doi:https://doi.org/10.1109/WCRE.2012.31.

[19] T. Zhang, G. Yang, B. Lee, A. T. S. Chan, Predicting severity of bug report by mining bug repository with concept profile, in: Proceedings of the 30th Annual ACM Symposium on Applied Computing, SAC '15, ACM, New York, NY, USA, 2015, pp. 1553–1558. doi:10.1145/2695664.2695872.

[20] P. Flach, Machine Learning: The Art and Science of Algorithms That Make Sense of Data, Cambridge University Press, New York, NY, USA, 2012.

[21] S. Marsland, Machine Learning: An Algorithmic Perspective, Second Edition, 2nd Edition, Chapman & Hall/CRC, 2014.

[22] S. Haykin, Neural Networks: A Comprehensive Foundation, 2nd Edition, Prentice Hall PTR, Upper Saddle River, NJ, USA, 1998.

[23] J. Zhou, H. Zhang, D. Lo, Where should the bugs be fixed? - more accurate information retrieval-based bug localization based on bug reports, in: Proceedings of the 34th International Conference on Software Engineering, ICSE '12, IEEE Press, Piscataway, NJ, USA, 2012, pp. 14–24.

[24] L. Breiman, Random Forests, Machine Learning 45 (1) (2001) 5–32. `doi:10.1023/A:1010933404324`.

[25] Y. Tian, N. Ali, D. Lo, A. E. Hassan, On the unreliability of bug severity data, Empirical Softw. Engg. 21 (6) (2016) 2298–2323. `doi:10.1007/s10664-015-9409-1`.

[26] Y. Zhao, Y. Cen, Data Mining Applications with R, 1st Edition, Academic Press, 2013.

[27] M. Kuhn, K. Johnson, Applied Predictive Modeling, SpringerLink : Bücher, Springer New York, 2013.

[28] G. Luo, A review of automatic selection methods for machine learning algorithms and hyperparameter values, Network Modeling Analysis in Health Informatics and Bioinformatics 5 (1) (2016) 18. `doi:10.1007/s13721-016-0125-6`.

[29] P. Probst, B. Bischl, A.-L. Boulesteix, Tunability: Importance of Hyperparameters of Machine Learning Algorithms, arXiv e-prints (2018) arXiv:1802.09596`arXiv:1802.09596`.

[30] R. Feldman, J. Sanger, Text Mining Handbook: Advanced Approaches in Analyzing Unstructured Data, Cambridge University Press, New York, NY, USA, 2006.

[31] G. Williams, Data Mining with Rattle and R: The Art of Excavating Data for Knowledge Discovery, Springer, 2011. `arXiv:arXiv:1011.1669v3`, `doi:10.1007/978-1-4419-9890-3`.

[32] A. Srivastava, M. Sahami, Text Mining: Classification, Clustering, and Applications, 1st Edition, Chapman and Hall/CRC, 2009.

[33] E. Giger, M. Pinzger, H. Gall, Predicting the fix time of bugs, in: Proceedings of the 2Nd International Workshop on Recommendation Systems for Software Engineering, RSSE '10, ACM, New York, NY, USA, 2010, pp. 52–56. `doi:10.1145/1808920.1808933`.

[34] A. Lamkanfi, S. Demeyer, Filtering bug reports for fix-time analysis, in: 2012 16th European Conference on Software Maintenance and Reengineering, 2012, pp. 379–384. doi:10.1109/CSMR.2012.47.

[35] H. Rocha, G. de Oliveira, M. T. Valente, H. Marques-Neto, Characterizing bug workflows in mozilla firefox, in: Proceedings of the 30th Brazilian Symposium on Software Engineering, SBES 2016, Maringá, Brazil, September 19 - 23, 2016, 2016, pp. 43–52. doi:10.1145/2973839.2973844.

[36] M. Habayeb, S. S. Murtaza, A. Miranskyy, A. B. Bener, On the use of hidden markov model to predict the time to fix bugs, in: Proceedings of the 40th International Conference on Software Engineering, ICSE '18, ACM, New York, NY, USA, 2018, pp. 700–700. doi:10.1145/3180155.3182522.

[37] A. Lamkanfi, S. Demeyer, Q. D. Soetens, T. Verdonck, Comparing mining algorithms for predicting the severity of a reported bug, in: 2011 15th European Conference on Software Maintenance and Reengineering, 2011, pp. 249–258. doi:10.1109/CSMR.2011.31.

[38] H. Valdivia Garcia, E. Shihab, Characterizing and predicting blocking bugs in open source projects, in: Proceedings of the 11th Working Conference on Mining Software Repositories, MSR 2014, ACM, New York, NY, USA, 2014, pp. 72–81. doi:10.1145/2597073.2597099.

[39] E. de Jonge, M. van der Loo, An introduction to data cleaning with R, Statistics Netherlands (2013) 53.

[40] A. Fernandez, S. Garcia, F. Herrera, N. V. Chawla, Smote for learning from imbalanced data: Progress and challenges, marking the 15-year anniversary, Journal of Artificial Intelligence Research 61 (2018) 863–905.

[41] N. Japkowicz, M. Shah, Evaluating Learning Algorithms: A Classification Perspective, Cambridge University Press, New York, NY, USA, 2011.

[42] F. Wilcoxon, Individual Comparisons by Ranking Methods, Springer New York, New York, NY, 1992. doi:10.1007/978-1-4612-4380-9_16.

[43] G. Canfora, M. Ceccarelli, L. Cerulo, M. Di Penta, How long does a bug survive? an empirical study, in: 2011 18th Working Conference on Reverse Engineering, 2011, pp. 191–200.

[44] L. Marks, Y. Zou, A. E. Hassan, Studying the fix-time for bugs in large open source projects, in: Proceedings of the 7th International Conference on Predictive Models in Software Engineering, Promise '11, Association for Computing Machinery, New York, NY, USA, 2011. doi:10.1145/2020390.2020401.
URL https://doi.org/10.1145/2020390.2020401

[45] R. K. Saha, S. Khurshid, D. E. Perry, An empirical study of long lived bugs, in: 2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE), 2014, pp. 144–153. `doi:10.1109/CSMR-WCRE.2014.6747164`.

[46] I. Goodfellow, Y. Bengio, A. Courville, Deep Learning, MIT Press, 2016.