New Article

Luiz Alberto Ferreira Gomes, Ricardo da Silva Torres, and Mario Lúcio Côrtes,

***Abstract*—Software evolution and maintenance activities in today Free/Libre Open Source Software (FLOSS) rely primarily on information extracted from reports registered in bug tracking systems. Many studies point out that most bugs that adversely affect the user's experience are the long-lived ones, even though they are fairly infrequent in open source projects. However, proposed approaches for supporting bug fixing procedures have not considered the real-world lifecycle of a bug, which is often fixed very fast. That may lead to useless efforts to automate the bug management process.** *One of the goals of this paper is to confirm that, for popular open source projects, the amount of short-lived bugs is very high.* **This paper also conducts a comparative study considering the evaluation four well-known machine learning algorithms with respect to prediction accuracy, to build a model to predict long-lived bugs. In our experiments, we have collected bug reports from six popular open source projects repositories (Eclipse, Freedesktop, Gnome, Gcc, Mozilla, and WineHQ) and used the following algorithms: K-Nearest Neighbor, Naïve Bayes, Neural Networks, Random Forest, and Support Vector Machines.** *The experiments show that short-lived bugs are much more frequent that long-lived ones, which may impact the choice of appropriate algorithms for supporting bug fixing management. We also observed that the Naïve Bayes yields superior results when compared to the other evaluated approaches. In addition, our results demonstrated that it is possible to predict long-live bugs with reasonable balanced accuracy (up to 68% in average, and up to 86% individually for the different datasets), despite the use of simple classifiers*

***Index Terms*—Software maintenance; Bug tracking systems; Machine learning; Text mining.**

I. Introduction

Bug tracking systems (BTS) have been playing a key role as a communication and collaboration tool in Closed Source Software (CSS) and Free/Libre Open Source Software (FLOSS). In both development environments, planning of software evolution and maintenance activities relies primarily on information of bug reports registered in this kind of system. This is particularly true in FLOSS, which is characterized by the existence of many users and developers with different levels of expertise spread out around the world, who might create or be responsible for dealing with several bug reports [1].

A user interacts with a BTS often through a simple mechanism called bug report form, which enables to communicate a bug to those in charge of developing or maintaining the software system [2]. Initially, he or she should inform a short description, a long description, and an associated severity level (e.g., blocker, critical, major, minor, and trivial). Subsequently, a software team member reviews this bug report and confirms or declines it (e.g., due to bug report duplication). If the bug report is confirmed, the team member should provide more information to complement the bug report form, for example, by indicating its priority and by assigning a person who will be responsible for fixing the bug. The number of bug reports in large and medium software FLOSS projects is frequently very large [3]. For example, the Eclipse project had 84,245 bug reports opened from 2013 to 2015 alone, whereas the Android project had over 107,456, and the JBoss project had over 81,920. Therefore, manual handling of bug reports (e.g., assignment of severity level) may be a quite subjective, cumbersome, and error-prone process, and a wrong decision throughout the bug report lifecycle may strongly disturb the planning of maintenance activities. For instance, an important maintenance team resource could be allocated to address less significant bug reports before the most important ones.

Due to its evident importance, both the business and academic community have made an extensive investigation towards the proposal of methods to automate different aspects of both software development and bug report management (e.g., bug report duplication detection). *However, the proposed approaches have not considered the fixing time of a bug, which may result in useless automation efforts, in the case, for example, of short-lived bugs. In fact, Saha et al. [4] points out that the amount of short-lived bugs in open source projects is very high.*

The definition of long-lived is subjective since the time threshold for deciding whether a bug is long-lived or short-lived could vary across projects, practitioners, or studies. *Saha et al. [4], for instance, consider that a long-lived bug should survive over at least two release cycles of a product which corresponds at a minimum one year in the FLOSS investigated in the paper. Even in smaller numbers, approximately 90% of long-lived bugs adversely affect the user's experience [4], which may disturb the users for a long time. These facts may suggest that many long-lived bugs could be fixed more quickly through careful selection and prioritization if developers could be able to predict a long-lived bug.*

Therefore, it may be useful investigating techniques to implement long-lived bugs predictors. 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 [1]. Furthermore, since bug reports typically come with textual descriptions, text mining techniques are likely candidates for providing appropriate input to these algorithms.

In this context, the general purpose of our research is to conduct a comparative study considering the evaluation of four well-known machine learning algorithms for prediction accuracy to build a model to predict long-lived bugs. Our

L. Gomes is with the Department of Software Engineering, Pontifical Catholic University of Minas Gerais, Belo Horizonte, MG, Brazil; e-mail: luizgomes@pucpcaldas.br

L. Gomes and M. Côrtes are with the Institute of Computing, University of Campinas, Campinas, Brazil.

Ricardo S. Torres is with the Department of ICT and Natural Sciences, Norwegian University of Science and Technology (NTNU), Ålesund, Norway; e-mail: ricardo.torres@ntnu.no

Manuscript received April 19, 2005; revised August 26, 2015.

specific goals are:

G$_1$. *Confirm that, for popular FLOSS projects, the amount of short-lived bugs is very high.*

G$_2$. Conduct a comparative study considering the evaluation four well-known machine learning algorithms to predict long-lived bugs.

G$_{2.1}$. Evaluate the effects of number of terms from unstructured text fields of bug report to perform long-lived prediction;

G$_{2.2}$. Evaluate and identify the best bug fixing time threshold to perform long-lived prediction.

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

RQ$_1$. *What time threshold has been used by researchers to decide if a bug is short or long-lived?* This research question aims to discover a common time threshold used by researchers to classify a bug as a short or long-lived bug. This threshold could be used to label a bug as short or long lived in training phase of model classification building.

RQ$_2$. *How frequent are the long-lived bugs in FLOSS projects?* This research question aims to assess the proportion of long-lived bugs in FLOSS projects. The answer to this question might indicate if it is worth investing in efforts to build long-lived bug prediction models.

RQ$_3$. *What are the main characteristics of long-lived compared to short-lived bugs in FLOSS projects?* This research question aims to determine the characteristics of long-lived bugs compared to short-lived bugs in FLOSS projects, including its severity.

RQ$_4$. *What is the accuracy of machine learning algorithms when predicting long-lived bugs?* This The research question aims to compare the accuracy of the long-lived bug prediction made by different classification algorithms. Here the algorithm with the best prediction capabilities is best fit in terms of balanced accuracy;

RQ$_5$. *What is the effect of number of terms on the performance of different classification algorithms?* This research question aims to investigate if there is a correlation between the number of terms in the feature vector and the accuracy observed for the different algorithms.

RQ$_6$. *Which value for the bug fixing time threshold is the most effective to predict a long lived bug?* This research question aims to identify the more suitable time threshold to perform the long lived bug prediction.

In summary, the contributions of our research are:

- Characterization of the bug fixing time in relevant FLOSS projects;
- Development of a machine learning and text mining framework able to perform long-lived bug prediction;
- Characterization of the effects of the number of terms in the long-lived bug prediction problem;
- Identification of suitable values for the bug fix time threshold and the number of terms to predict long-lived bugs.

The article is organized as follows. Section II provides the background terminology and concepts about bug tracking systems, text mining, and machine learning techniques. Section III presents related work. Section IV describes the methodology used in our work. Section V presents achieved results and discusses our findings. Finally, Section VII presents conclusions and points out possible future work.

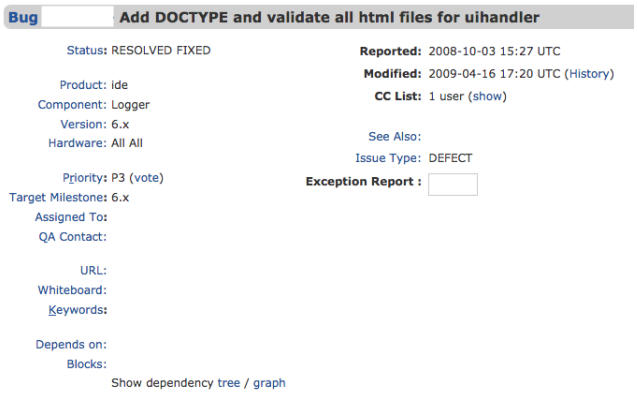


Fig. 1: A bug report example.

TABLE I: Common Attributes in a Bug Report Form.

Type	Type of report (e.g., bug, improvement, and new feature)
Short description	short description of report in one line.
Long Description	Long and detailed description of report in many lines of text. It could include source code snippets and stack tracing reports.
Severity	Report severity level (e.g., blocker, critical, major, minor, and trivial).

II. Terminology and Concepts

This section provides an overview about basic concepts related to bug tracking systems, text mining, machine learning, and evaluation metrics.

A. Bug Tracking Systems

Bug Tracking System (BTS) [5] is a software application that keeps the record and tracks information about change requests, bug fixes, and technical support that could occur during the software lifecycle. Usually, while reporting a bug in a BTS, a user is asked to provide information about the bug by filling out a form, typically called bug report form (Figure 1).

1) Bug Report: Although there is no agreement on the terminology or the amount of information that users must provide to fill a bug report, they often describe their needs in popular BTS (e.g., Bugzilla, Jira, and Redmine) [6], providing at least information about the attributes shown in Table I.

After the user has reported a bug, the development team is in charge of its assessment. The assessment consists of approving or, for some reason (e.g., duplication), not approving the bug. In case of approval, this team may provide complementary information by, for example, assigning a person to be responsible for handling this request or defining a new severity level for the report. Typically, the sequence of steps a bug report goes through is modeled as a state machine. Figure 2 shows an example of such a state machine, with a typical set of states

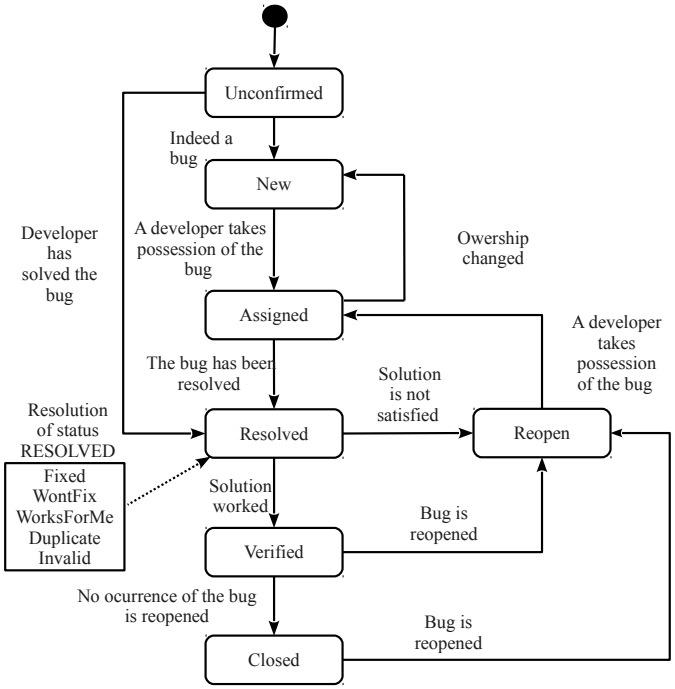


Fig. 2: The bug report lifecycle according to Zhang et al. [7].

a bug report can hold during its lifecycle in a BTS. Initially, a bug report is in the *Unconfirmed* state. The developer team will change the bug report state to *Resolved*, if the bug were not confirmed, or, otherwise, to *New*. When someone was in charge of fixing the bug, the bug report state will be changed to *Assigned* by the developer team. Therefore, in the standard flow, the bug report status will be 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 lifecycle. All changes occurred in a bug report are often stored in a repository, keeping valuable historical information about a particular software.

B. Machine Learning

Machine Learning (ML) [8] 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.

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 as commonly named class or category. Traditional ML classification algorithms are k-Nearest Neighbors, Naïve Bayes, Random Forest, 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. Next, a brief description of each one is presented [9]:

- **k-Nearest Neighbors** (k-NN) classifies a new instance based on its similarity measure to the k-nearest labeled neighbors. Usually, the k-NN classifier utilizes the Euclidean distance to quantify the proximity of neighbors. To calculate this distance, each instance in a dataset should be represented as point of an n-dimensional space (feature vector).
- **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.
- **Support Vector Machine** (SVM) is a learning algorithm in which each feature vector of each instance is a point in an n-dimensional space. SVM learns in this space an optimal way to separate the training instances according to their class labels. In case of linear separability, the output of this algorithm is a hyperplane, which maximizes the separation among feature vectors of instances of different classes. Given a new instance, SVM assigns a label based on which subspace its feature vector belongs to [10].
- **Random Forest** [11] 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.

2) Evaluation Metrics: Finally, it is worth mentioning the specific metrics we use for assessing prediction performance. The most common performance measures for evaluating the accuracy of classification algorithms are described as follows [9], [12]:

- **Accuracy**. Accuracy is the percentage of correctly classified observations among all observations. Accuracy is the number of True Positives (TP) plus the number of True Negatives (TN), all of this divided by the total of positive class instances (P) plus the total of negative class instances (N).
- **Balanced Accuracy**. Balanced Accuracy is calculated as the average of proportion of TP/(P+N) and TN/(P+N) of each class individually.
- **Recall**. Recall is TP divided by TP plus the number of False Negatives (FN). A low recall indicates many false negatives.
- **Precision**. Precision is TP divided by TP plus the number of False Positives (FP). A low precision can also indicate many false positives.
- **F-measure**. F-measure conveys the balance between precision and recall, and can be calculated as their harmonic mean.

C. Text Mining

The common ML algorithms cannot directly process unstructured text (e.g., *short description* and *long description* fields from the bug report form). Therefore, during a preprocessing step, these unstructured text fields are converted into more manageable representations. Typically, the content of these fields is represented by feature vectors, points of an n-dimensional space. Text mining is the process of converting unstructured text into a structure suited to analysis [13]. It is composed of three primary activities [14]:

- **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, common verbs, nouns, pronouns, adverbs, and adjectives.
- **Stemming** is the process of reducing or normalizing inflected (or sometimes derived) words to their word stem, base form—generally a written word 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) [13]. 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 (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 quantification scheme, considers the number of times in which the term appears in each document. Term Frequency-Inverse Document Frequency (TD-IDF), a more complex weighting scheme takes into account the frequencies 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.

III. Related Works

Giger et al. [15] 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 features might improve the model performance. Lamkanfi et al. [16] have observed that, for the cases of both Eclipse and Mozilla, a fraction of the bug reports indicate conspicuous fix-times where the bugs are often fixed within few minutes. They proposed to filter out these conspicuous bug reports when the researchers utilize data mining techniques to predict the fix-times of reported bugs. Zhang et al. [17] did 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 on different threshold times. Saha et al. [4] 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, which although not so frequent as short-lived bugs, there are a fair number of long-lived bugs in each FLOSS investigated and over 90% of them adversely affect the user's experience. The reasons for these long-lived bugs are many including, for example, long assignment time and the lack of understanding of their importance. However, many bugs became long-lived without any specific reason.

Rocha et al. [18] reported a characterization study focused on the workflow followed by Mozilla Firefox developers when resolving bugs. They proposed the concept of Bug Flow Graphs (BFG) to help understand this characterization. They concluded that (a) when a bug is not formally assigned to a developer, it requires ten more days to be resolved; (b) approximately 94% of duplicate bugs are closed within two days or less after they appear in the tracking system; (c) incomplete bugs, which are never assigned to developers, usually require 70 days to be closed; (d) more skilled developers show a faster resolution time than less skilled ones; (e) for less skilled developers a bug usually spends more time waiting to be assigned than being fixed. Habayeb et al. [19] proposed a novel approach using Hidden Markov models and temporal sequences of developer activities 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. [20] replicated the study performed by Zhang et al. [17] using an open source software project and confirmed their results.

IV. Methodology

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

A. Data Collection

This step in the experimental research encompasses selecting FLOSS datasets to serve as data source, studying and interpreting their data structure, and finally extracting relevant data from their repository (feature extraction). In our research, we used the following FLOSS projects: Eclipse, Freedesktop, Gnome, Mozilla, Netbeans, and WineHQ. That was a natural choice, since they are open, well established, have a considerable number of bug reports already registered, use standard repositories, and were under study by other researchers [5], [6], [16], [21], [22]. These projects are described in the following:

TABLE II: FLOSS projects used in our research.

Project	Number of Bugs	Observation Period	
		From	To
Eclipse	7614	2001-10-10	2018-01-31
Freedesktop	7644	2003-01-09	2018-01-22
Gnome	4904	1999-02-18	2018-01-04
Mozilla	2163	1998-04-21	2014-04-22
Netbeans	6640	1999-02-11	2017-09-30
WineHQ	6185	2000-08-30	2019-02-26

- **Eclipse**[1]: Eclipse is an integrated development environment used in computer programming, and is the most widely used Java IDE.
- **Freedesktop**[2]: Freedesktop hosts the development of free and open source, focused on interoperability and shared technology for open-source graphical and desktop system.
- **GNOME**[3]: GNOME is a free and open-source desktop environment for Unix-like operating systems.
- **Mozilla**[4]: Mozilla encompasses several open-source projects, such as Firefox, Thunderbird, and Bugzilla.
- **Netbeans**[5]: Netbeans is an integrated development environment used in computer programming, and is a popular Java IDE.
- **WineHQ**[6]: WineHQ is a compatibility lavel capable of running Windows applications on several POSIX-compliant operating systems, such as Linux, macOS, and BSD.

We have downloaded randomly the bug reports of the previous open source projects which are hosted in a Bugzilla BTS. All bug reports were stored in a cvs file format for each project. We extracted the datasets by collecting reports submitted within a period of time. Each dataset only contains resolved or closed bug reports, whereas feature requests and maintenance tasks are filtered out. Table II shows additional information on the six datasets considered in our study.

We have used the final bug fix date in the bug reports as the ground truth to compute the bug fix time (fixed date – opened date). Only bug reports with fix time less than equal to 730 days (two years) were considered in our research.

B. Data Preprocessing

Raw data previously collected from the Eclipse, Freedesktop, Gnome, Mozilla, Netbeans, and WineHQ bug reports repositories were not properly structured to serve as input to ML algorithms [23]. The classical way to address this problem is to run preprocessing 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:

- Extraction of relevant features: key, creation date, resolution data, short description, long description, and bug fix time of bug reports;
- Selection of only bug reports with status equals to *Closed* or *Resolved*, and resolution status equals to *Fixed*. This type of bug report was effectively implemented by the development team and they can no longer have their resolution date changed.
- Performing text mining in the concatenation of short long description and long description fields to identify the 100, 200, 300, 400, and 500 most frequent terms using the TF-IDF weighting scheme. This information is then converted into features for each bug report.

C. Data Description

We have used histograms, medians, and quartiles to statistically describe and visualize the bug fix time distribution over the investigated FLOSS projects.

D. Training and Testing

Training and testing steps start with partitioning the already preprocessed dataset in 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. No approach (e.g., SMOTE [24]) was employed to balance the training set. Yet, in the training phase, we have used the Leave-Group-Out CV (LGOCV) technique [25] to obtain more stable estimates of each algorithm's performance and enhance replicability of the results [26]. In the testing phase, each ML algorithm was validated with 25% of each bug report dataset to measure its accuracy.

We have chosen four traditional ML algorithms: *k-NN*, *Naïve Bayes*, *Random Forest*, and *SVM* which were implemented using the Caret[7] R library. It is worth to mention that: (i) we have used the R Caret Library algorithms default setting parameters; and (ii) we have not used any class data balancing strategy.

V. Results

This section presents the experimental results and describes the significance of our findings. The discussion presented here was organized by the research question, just like the previous section.

A. RQ1: How the literature has defined a long-lived bug?

As mentioned before, RQ1 aims to find out if there is in the bug-related literature a shared understanding of what is a long-lived bug. Table III shows papers related to long-lived bugs and, also, references related to bug fixing effort estimation and bug report life cycle, which will support addressing the current research question. The table has five columns: the bibliographic reference (first column), the year of publication (second column), the definition of long-lived bugs (third column), the experimental data sets (fourth column), and the

[1]www.eclipse.org (As of May 2019).
[2]www.freedesktop.org (As of May 2019).
[3]www.gnome.org (As of May 2019).
[4]www.mozilla.org (As of May 2019).
[5]www.netbeans.org (As of May 2019).
[6]www.winehq.org (As of May 2019).
[7]https://caret.r-forge.r-project.org (As of May 2019).

author's comments (last column). It is important to emphasize that the results presented in Table III are not expected to be a full systematic literature review.

From Table III, we can observe that only **three publications** defined a long-lived bug formally. Two of them, which are from the same authors, consider a bug to be long-lived if it **survives more than one year**, and another paper, if it **survives more than the median of bug fixing time**.

The FLOSS column shows that most papers used bug data from the Eclipse project, and the column comment provided some highlights. For example, 90% of long-lived bugs affect a user's normal working experience, and the reasons for long-lived bugs are diverse. RICARDO: include the year of each publication - DONE
RICARDO: try to make the quotes shorter MARIO: fix correcoes editoriais. Verificar

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

In its turn, RQ2 aims to investigate the prevalence of long-lived bugs in FLOSS projects. Figure 3(a)-(f) shows the percentage of long-lived bugs in selected FLOSS. MARIO: Nao explicitou qual threshold foi usado aqui. Um ano? Não sei se seria bom colocar tbem esta info no caption da figura. The percentage values of long-lived bugs in this chart range widely from 7.7% (in Eclipse) to 40.7% (in WineHQ).

In addition, Figure 4 shows the bug fixing time distribution for each investigated project. As shown in the figure, the project's teams have fixed most bugs within 365 days. Moreover, they have fixed around 50% of the total number of bugs within from 8 (median for Eclipse) to 220 days (median for WineHQ). MARIO: Colocar a unidade - dias - no eixo horizontal dos histogramas.

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

Besides investigating the shared understanding of long-lived bugs and its prevalence in FLOSS projects, we investigate, in RQ3, the characteristics of long-lived bugs based on the following bug reports fields: bug reporters, assignee, component, severity level, summary, and description. MARIO: Nao seria melhor explicar a diferenca entre bug reporter e assignee? Eh obvia pra nos, mas para o leitor/revisor?

1) Bug Reporter: Figure 5 shows the percentage of bugs reported by the top-ten reporters. As shown in the figure, these reporters have registered a meaningful percentage (8.3% to 22.7%) of total bugs in our data sets.

Figure 6 splits up the bugs reported by each of the top-ten reporters as either short-lived or long-lived bugs. Notice that the percentage of long-lived bugs in this total is relatively high for some of them: **ed** in Eclipse (≈ 15.4%), **freedesktop** in Freedesktop (≈ 23.5%), **dawn** in Gnome (≈ 22.6%), **timeless** in Mozilla (≈ 23.97%), **burnus** in GCC (≈ 24.2%), and **fgouget** in WineHQ (≈ 49%). Furthermore, most reporters (7 out of 10) in WineHQ registered a percentage of long-lived bugs which are superior than 30%.

2) Assignee: Figure 7 presents the percentage of bugs assigned to the top-ten assignees for each data set. The top-ten assignees, as shown in this figure, have fixed a very significant proportion (9.4% - 96.8%) of total bugs in each data set. Notice that the top-ten assignees from WineHQ are asosciated with most of the bugs (96.8%).

Figure 8 splits up the total of bugs reported by each top-ten assignee in short-lived and long-lived bugs. As observed for bug reporters, the percentage of long-lived bugs in this total was relatively high for some assignees: **mdt-papyrus-inbox** in Eclipse (≈ 31.8%), **nobody** MARIO: Luiz, onde vc explica o que eh o nobody, ou o nao atribuido? 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%). Furthermore, most reporters MARIO: Acho que aqui seria assignees, e nao reporters. Certo? (8 out of 10) in WineHQ registered a percentage of long-lived bugs in the total of bugs reported for each of them superior to 30%.

3) Component: Figure 9 shows the percentage of bugs associated with the top-ten components. As shown in the figure, these components raised a significant proportion (32.5% - 80.2%) of total bugs in each of the data sets. MARIO: redacao alternativa: a significant fraction (32.5% - 80.2%) of total bugs in each of the data sets occurred in these components.

Figure 9 MARIO: refere-se a figura errada. Seria a fig 10. categorizes the total of bugs reported by each top-ten component as short-lived and long-lived bugs. As noted for previous bug-report fields, the percentage of long-lived bugs in this total was relatively high for some components: **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%). Furthermore, most components (7 out of 10) in WineHQ registered a percentage of long-lived bugs in the total of bugs reported for each of them superior to 30%.

We can notice two facts in MARIO: referencia errada. Deveria ser fig. 10 Figure 9: (i) Hyades[8] had no long-lived bugs reported; and (ii) the high proportion of unknown modules in bug reports of WineHQ.

4) Summary: Figure 11 presents the word cloud generated over the summary field for all investigated projects. We can observe that there is a group of repeated words in the summary field of either short or long lived bugs (e.g., error, fail, file, and regression). MARIO: A figura ainda se refere a short description. Atualizar para summary

5) Description: Figure 12 shows the word cloud generated over the description field for all investigated projects. Also, as in word clouds for summary bug report field, we can notice that there is a group of repeated words in the summary field of either short or long lived bugs (e.g., eclipse, java, lib, and usr). MARIO: A figura ainda se refere a long description. Atualizar para description

[8]Hyades is an open-source platform for Automated Software Quality (ASQ) tools and a range of open-source reference implementations of ASQ tooling for testing, tracking and monitoring software systems. Online: https://www.eclipse.org/org/press-release/apr152003uml2pr.html

TABLE III: Publications related to long-lived bugs.

Reference	Year	Definition	FLOSS	Author's Comment
Chiara et al. [27]	2008		Authors have used mnemonics to refer to projects instead of their real names.	• "When a release date approaches, developers hasten to correct bugs with the aim of including corrections in the forthcoming release. In fact, if changes are verified and committed before the scheduled deadline, they can be included in the release; otherwise, they would shift to the next release."
Giger et al. [15]	2010	A slowly-fixed bug (in our context, a long-lived bug) has a bug fixing time (the time between the opening date and the date of the last change of the bug resolution to FIXED) higher than the median.	Eclipse, Gnome, and Mozilla	• "Assignee, reporter, and month opened are the attributes that have the strongest influence on the fix-time of bugs."
Saha et al. [28]	2014	Bug that are not fixed within one year after they are reported	Eclipse	• "More than 90% of long lived bugs affect users' normal working experiences and thus are important to fix. However, it took a long time to fix these bugs even after realizing their severity. Moreover, there are multiple bug reports for these long lived bugs,which indicate the users' demand for fixing them."
Saha et al. [4]	2015	Bug that are not fixed within one year after they are reported	Eclipse, GCC, Linux Kernel, and WineHQ	• "Reasons for long lived bugs are diverse. While problem complexity, problems in reproducing errors, and not understanding the importance of some of the bugs in advance are the common reasons, we observed there are many bug-fixes that were delayed without any specific reason."
Rocha et al. [18]	2016		Firefox	• "When a bug is not formally assigned to a developer it requires ten more days to be re- solved;"
Wang et al. [29]	2018		CXF, Flink, Flume, Groovy, Hadoop, NiFi, OpenJPA, PDFBox, Tuscany, Wicket	• "The experimental results show that the text feature based method perform better than other baseline on 10 projects when predicting the version-length of bug lifecycle."

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

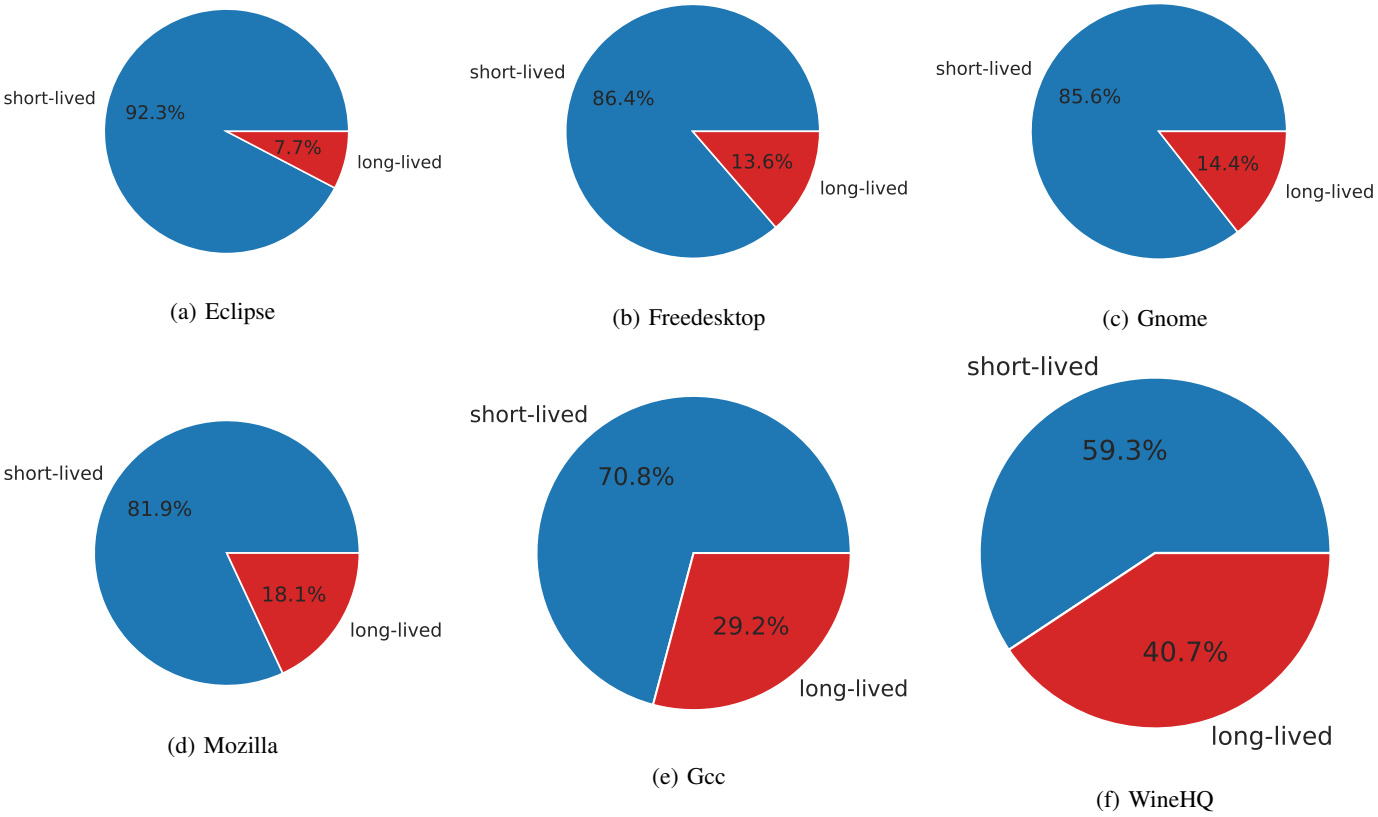
The purpose of RQ4 is to compare the performance of different ML algorithms in predicting a long-lived bug, namely *KNN*, *Naïve Bayes*, *Neural Networks*, *Random Forest*, and *Support Vector Machines*. The investigated algorithms rely on distinct principles, which might lead to varying effectiveness performances. To answer that research question, we conducted four experiments, changing specific parameters before the execution of each one. The following sections present achieved results for each experiment.

1) RQ4.1. Evaluating performance of ML algorithm when predicting long-lived bug: In the first experiment, we evaluated the performance of selected ML algorithms when predicting long-lived bugs on the Eclipse data set using parameters presented in Table IV. The number of parameter combinations derived from this table is equal to 60. For each combination, every algorithm has built a prediction model based on a feature matrix 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, Kappa, or AUC. As shown in MARIO: referencia errada. Nao seria table V? Figure **??**, considering the bug fixing time threshold equals 365 days, the Eclipse data set is notably unbalanced. Therefore, we ran each algorithm either over the same unbalanced data set or over a balanced data set built applying the SMOTE method on the original data set.

TABLE IV: First experimental parameters.

Parameter	Value
Data set	Eclipse
Bug report fields	Summary and Description
Number of observations	9998
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 5x2
Bug fixing time threshold	365 days

Table V shows the performance of investigated algorithms when predicting long-lived bugs on the Eclipse data set. To facilitate the visualization and analysis of results, we have



(a) Eclipse (b) Freedesktop (c) Gnome

(d) Mozilla (e) Gcc (f) WineHQ

Fig. 3: The pie charts for percentages of long-lived and short-lived bugs in investigated FLOSS projects.

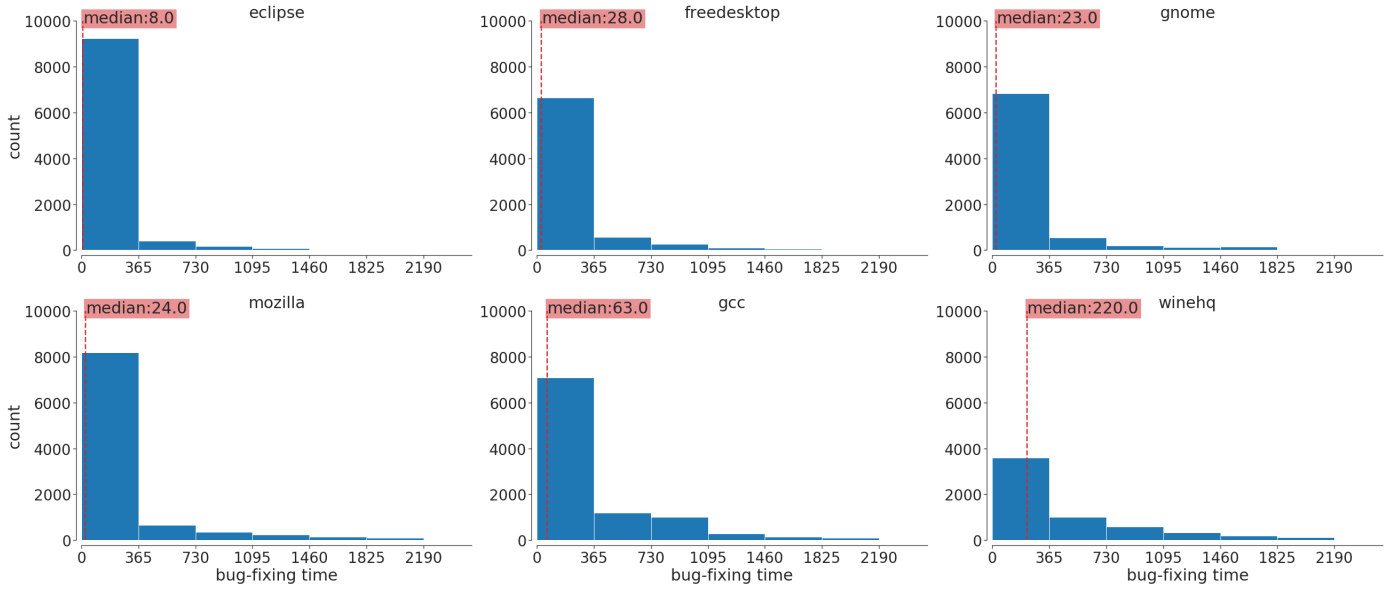


Fig. 4: Distribution histograms of bug fixing time in FLOSS projects.

displayed only the ten first rows sorted in descending order by balanced accuracy scores.

As one can observe, the **Neural Network** algorithm produced the best predicting model (≈ 54% of Balanced Accuracy) using the **Description** field to generate the text feature vector, the **SMOTE** method to balance data set, and the **Accuracy** metric to choose the best model during the training phase. Notice, the Neural Network had yielded the same Balanced Accuracy score using either Accuracy or Kappa, as shown in Table V, we use Accuracy in the next experiments as this metric is easy to understand and to interpret, being widely used in practice [26]. RICARDO: melhor usar balanced accuracy ou normalized accuracy? nas bibliotecas do R e do Python o termo utilizado é balanced accuracy

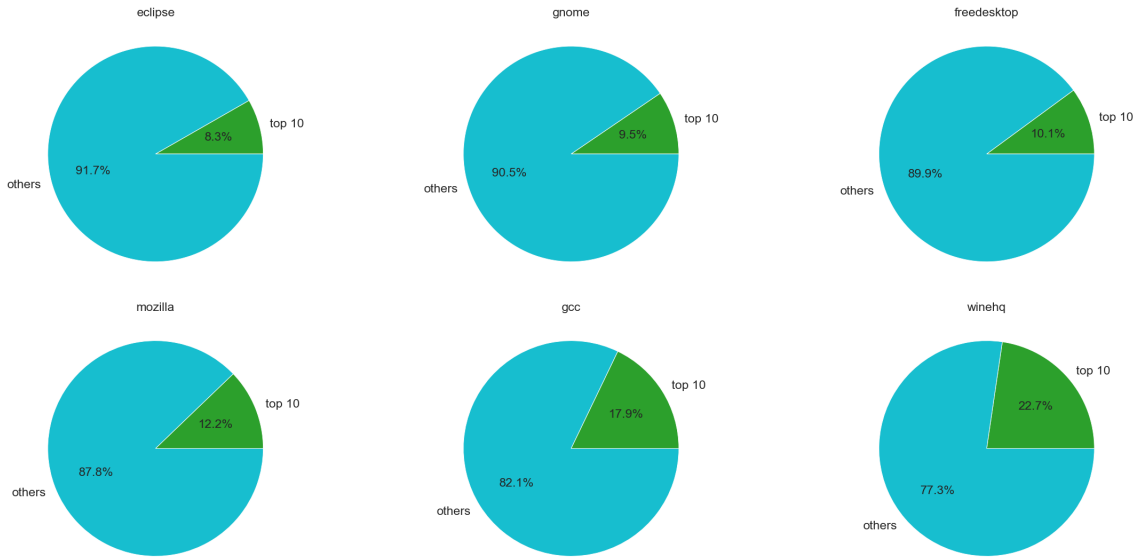


Fig. 5: Percentage of bugs reported by top ten reporters.

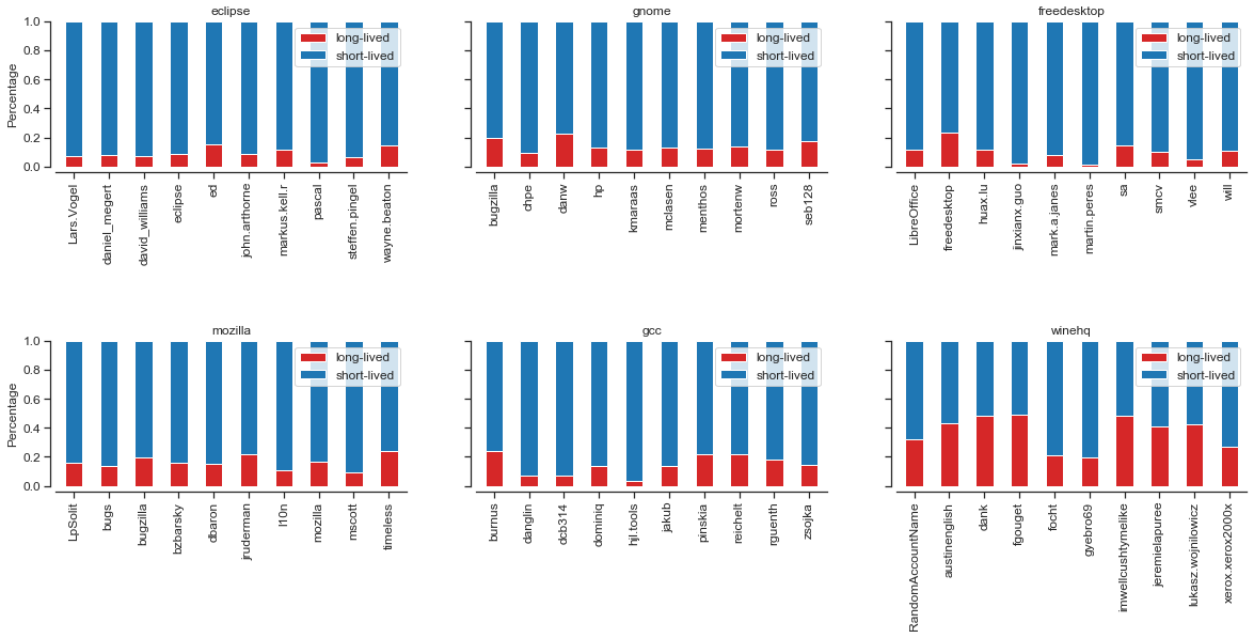


Fig. 6: Percentage of long-lived bugs by bug reporter.

TABLE V: Performance of predictive algorithms for Eclipse project.

#	Bug Report Field	Algorithm	Balancing Method	Metric	Sensitivity (%)	Specificity (%)	Balanced Accuracy (%)
1	Description	Neural Network	Smote	Accuracy	37.96	70.46	54.21
2	Description	Neural Network	Smote	Kappa	37.96	70.46	54.21
3	Summary	Näive Bayes	None	Kappa	62.96	45.17	54.06
4	Summary	Näive Bayes	None	AUC	62.96	45.17	54.06
5	Description	Random Forest	Smote	Accuracy	10.69	95.40	53.05
6	Description	Random Forest	Smote	Kappa	10.69	95.40	53.05
7	Description	Neural Network	Smote	AUC	37.43	68.66	53.04
8	Description	Näive Bayes	None	Kappa	64.17	40.88	52.52
9	Description	Näive Bayes	None	AUC	64.17	40.88	52.52
10	Description	KNN	Smote	AUC	59.89	44.75	52.32

2) RQ4.2. Evaluating the impact on ML algorithm performance when varying the number of terms: The number of

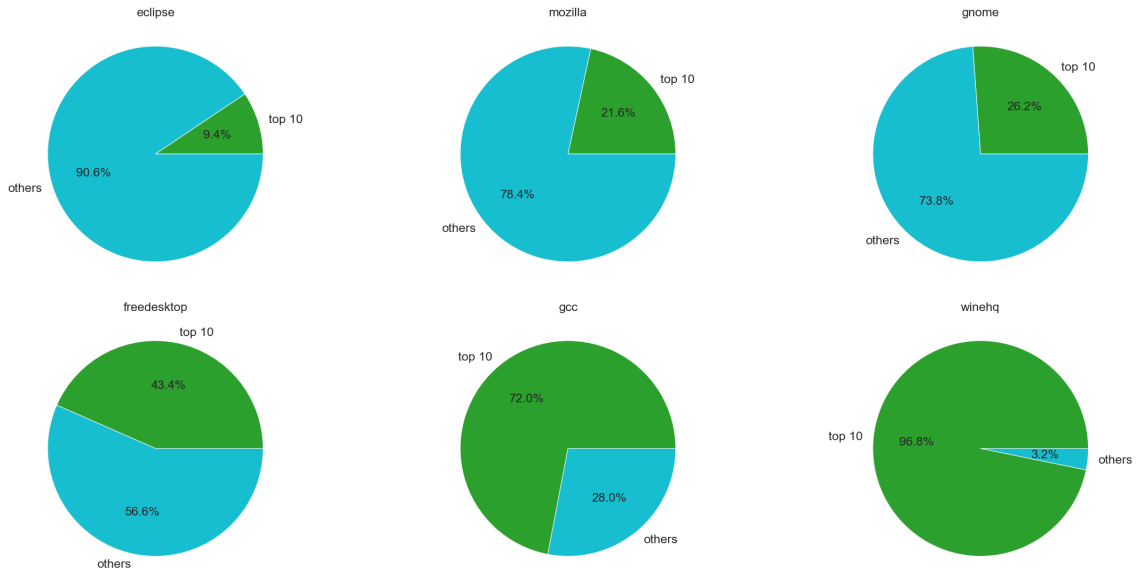


Fig. 7: Percentage of bugs reported by top-ten assignees.

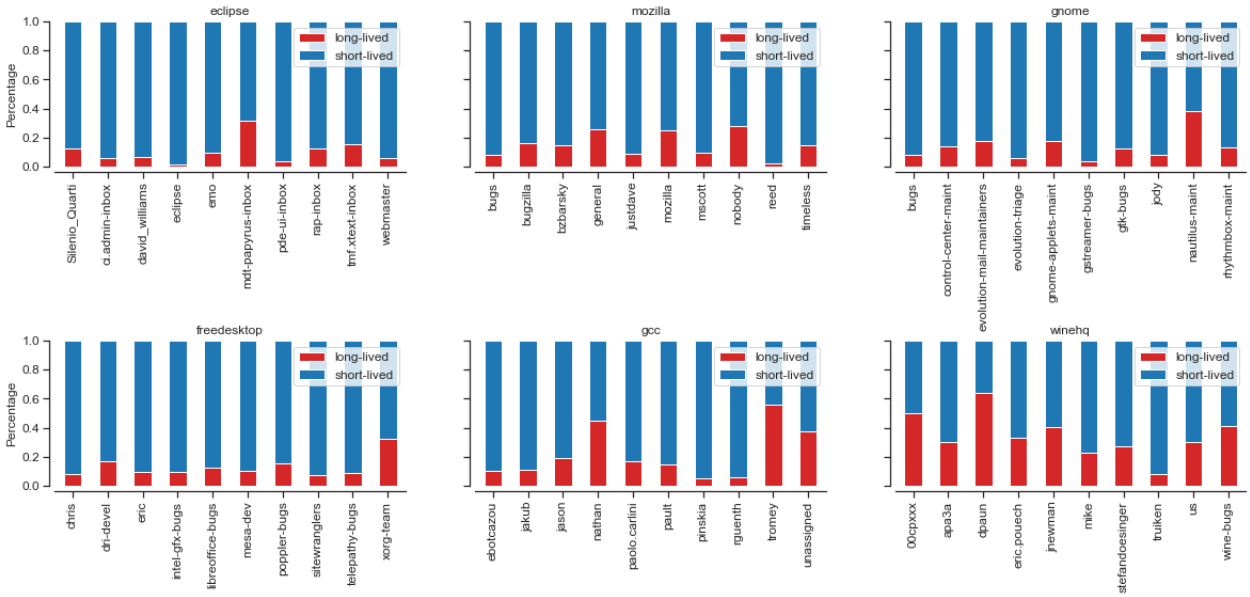


Fig. 8: Percentage of long-lived bugs by assignee.

terms and, hence, the size of feature vectors in the training set also plays an essential role in evaluating the effectiveness of the ML algorithms performance when predicting long lived bugs. Therefore, we investigated the influence of the number of terms in the feature vectors on the balanced accuracy performance of the best predicting algorithm choose in RQ4.1, which was the Neural Network.

Table VI presents the parameters used in this second experiment of RQ4. In our evaluation, we kept most parameters used in the previous research question and changed the number of terms to a set of values comprised of 100, 150, 200, 250, and 300 to run this experiment.

RICARDO: nao entendi o que voce quis dizer ao usar almost - corrigido

Figure 13 presents the performance curves for Neural Network on the Eclipse data set. The figure indicates the sensitivity, specificity, and balanced accuracy scores for the algorithm employing different numbers of terms in the text feature vector.

TABLE VI: Parameters for RQ4.2 experiment.

Parameter	Value
Data set	Eclipse
Bug report field	Description
Number of observations	9998
Predictive algorithm	Neural Network
Number of terms	100, 150, 200, 250, 300
Metric to select model	Accuracy
Balancing method	Smote
Sampling method	Repeated CV 5x2
Bug fixing time threshold	365 days

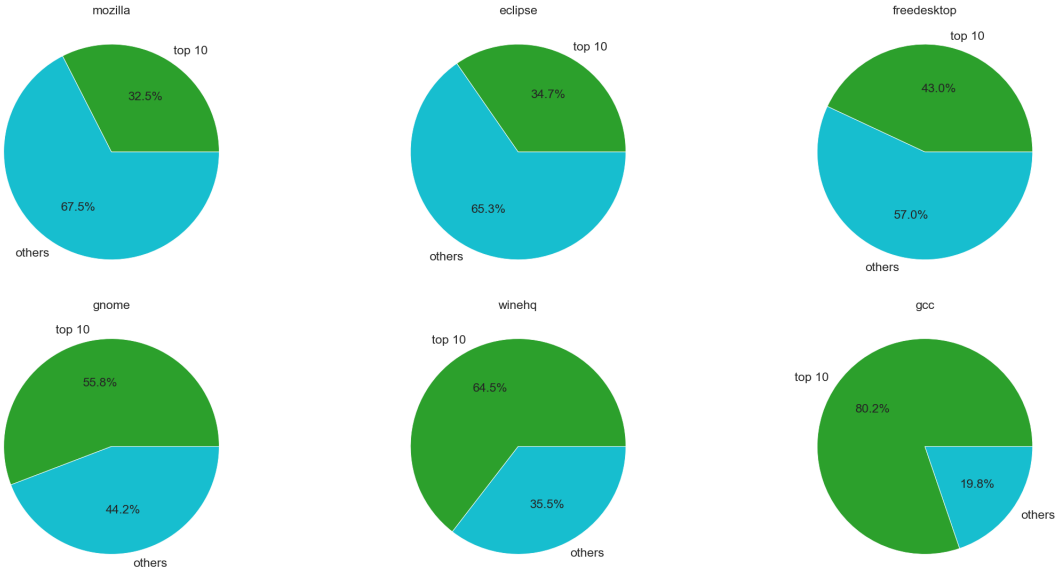


Fig. 9: Percentage of bugs reported related to top ten components.

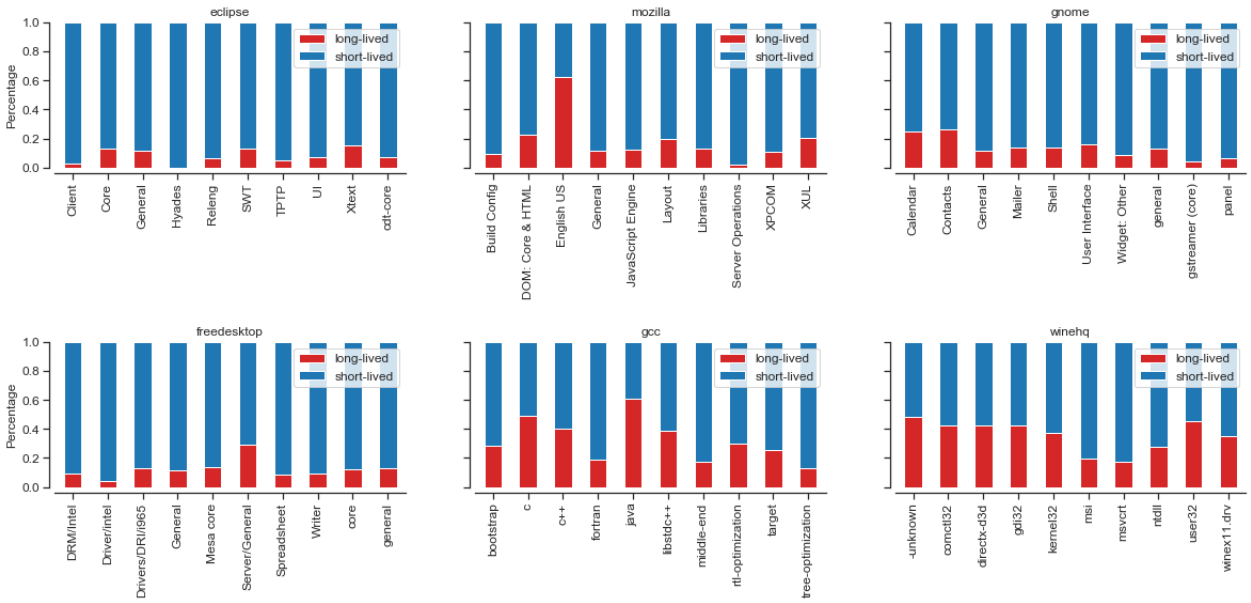


Fig. 10: Percentage of long-lived bugs by component.

Fig. 11: Word clouds based on the summary field of bug reports for all investigated project.

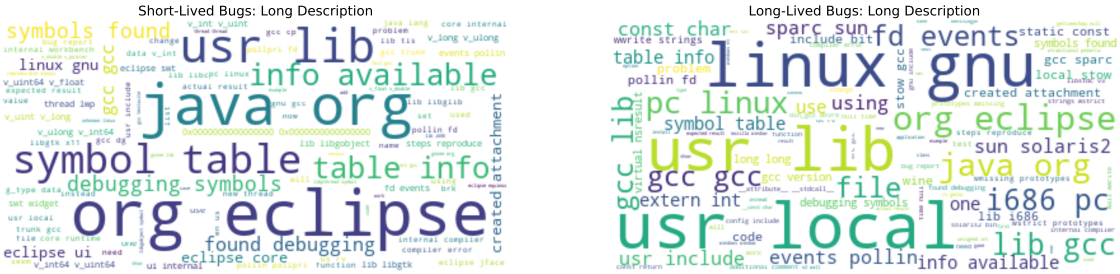


Fig. 12: World clouds based on the description field of bug reports for all investigated project.

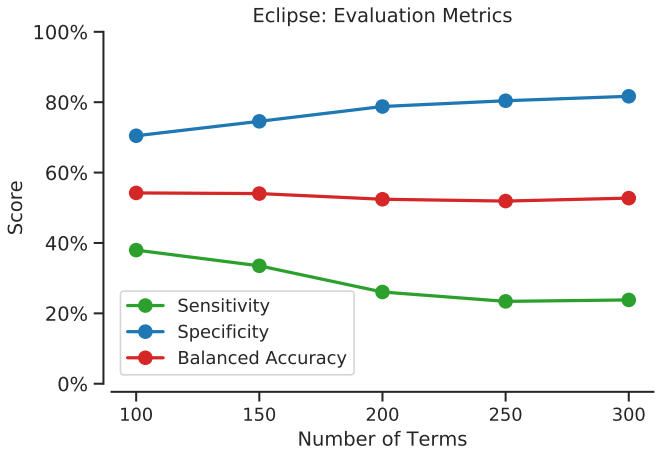


Fig. 13: Sensitivity, Specificity, and Balanced Accuracy scores from Neural Network predicting long-lived bugs on Eclipse data set by the number of terms in the feature vector.

We can observe in the line chart that the balanced accuracy fluctuated slightly, while sensitivity decreased, and specificity increased - both sharply. MARIO: Usar simbolos diferentes para as 3 curvas, além da cor. Tipo, bola vazia, bola cheia, cruz, triangulo
RICARDO: Onde voce definiu a medidas de avaliacao usadas? as medidas serão definidas na seção de terminiologia

The Neural Network algorithm yielded the best-balanced accuracy (≈ 54.21%) when the text mining process extracted **100 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 (a long-lived bug), we might want to take into account the vector with 100 features, given its better sensitivity, as shown in Figure 13.

3) RQ4.3. Evaluating the impact on ML algorithm performance when varying the bug fixing time threshold: : We could think of the median value as the most suitable value for the bug fixing time threshold because it splits up all bugs in 50% of short-lived and 50% long-lived bugs. However, figure 14 shows the bug-fixing time threshold based on the median yielded the worst balanced accuracy and sensibility in investigated data sets. On the other hand, the highest balanced accuracy was generated for the bug fixing time threshold equal to 365 days, which was suggested by Saha et el. [4].

TABLE VII: Parameters for RQ4.3 experiment.

Parameter	Value
Data set	Eclipse
Bug report fields	Description
Number of observations	9998
Predictive algorithms	Neural Network
Number of terms	100
Metric to select model	Accuracy
Balancing method	Smote
Resampling method	Repeated CV 5x2
Bug fix time threshold	8, 63, 108, and 365 days

We can suppose the existence of features vectors that better enabled the predictor to distinguish a long-live from a short-lived bug.RICARDO: why NN? foi o algoritmo que obteve a melhor acurácia balanceada na RQ4.1, and the number of terms in feature vector picked in RQ4.2, which was 100.

To do the before mentioned evaluation, we set the bug fixing time thresholds in 8, 63, 108, and 365 days and performed a new experiment using the parameters, as shown in Table VII. The values of bug fixing thresholds denote respectively the median (as suggested in Giger et al. [15]), the third quartile, the mean of bug-fixing time in Eclipse data set, and the threshold value (as suggested in Saha et al. [4])
RICARDO: sua descricao nao segue uma mesma estrutura; use uma descricao top-down - corrigido

Figure 14 presents the performance curves for Neural Network on the Eclipse data set. The figure indicates the sensitivity, specificity, and balanced accuracy scores for the algorithm employing different bug fixing time threshold. We can observe in the line chart that the balanced accuracy fluctuated slightly, while sensitivity and specificity fluctuated sharply. As shown in the figure, the Neural Network algorithm yielded the best-balanced accuracy (≈ 54.21%) for the 365 bug-fixing time threshold.

4) RQ4.4. Evaluating the performance of the ML algorithm in other FLOSS data sets: : In the previous experiments, we evaluated the performance of ML algorithms when predicting a long-lived bug (RQ4.1), the influence of the variation of the number of terms in the feature vector on the performance of the Neural Network algorithm (RQ4.2), and the impact of the variation of the bug fixing time threshold on the performance of the Neural Network algorithm (RQ4.3).

In this last experiment of RQ4, we investigated the balanced

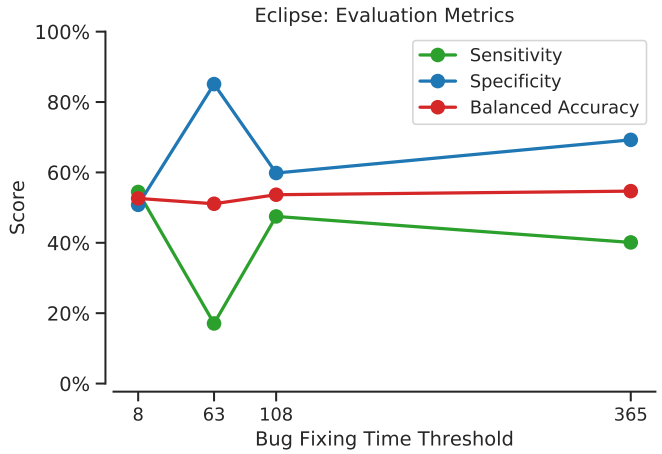


Fig. 14: Sensitivity, Specificity, and Balanced Accuracy scores from Neural Network predicting long-lived bugs on Eclipse data set by the bug fixing time threshold.

accuracy performance of the Neural Network algorithm in other data sets. To do that, as shown in Table VIII, we kept almost the parameters used by the ML algorithm that yield the best results in the RQ4.1, RQ4.2 and RQ4.3 experiments, and we set up the data set parameter as a set of data sets composed by Eclipse, Freedesktop, GCC, Gnome, Mozilla, and WineHQ.

TABLE VIII: Parameters for RQ4.4 experiment.

Parameter	Value
Data sets	Eclipse, Freedesktop, GCC, Gnome, Mozilla, and WineHQ
Bug report fields	Description
Number of observations	9998
Predictive algorithms	Neural Network
Number of terms	100
Metric to select model	Accuracy
Balancing methods	Smote
Resampling method	Repeated CV 5x2
Bug fix time threshold	365

Figure 15 presents the performance curves for Neural Network on Eclipse, Freedesktop, GCC, Gnome, Mozilla, and WineHQ data sets. The figure indicates the sensitivity, specificity, and balanced accuracy scores for the algorithm employing different bug fixing time threshold. We can observe in the char that the Neural Network algorithm yielded the best-balanced accuracy ($\approx$ 70.70%) for the GCC data set. Furthermore, sensibility and specificity scores for this data set were $\approx$ 52.48% and $\approx$ 89.60%, respectively.

E. RQ5: What are the characteristics of bugs that were correctly and incorrectly predicted as long-lived by classifiers?

Finally, in this last research question, we investigated the characteristics of bugs that were predicted as long-lived and were truly long-lived (true positive), and, also, those bugs predicted as not long-lived, but that were long-lived bugs (false negative).

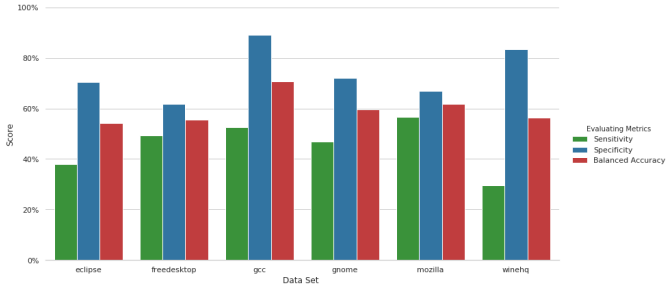


Fig. 15: A comparison of the balanced accuracy, sensitivity, and specificity from Neural Network predicting long-lived bugs on Eclipse, Freedesktop, GCC, Gnome, Mozilla, and WineHQ data sets.

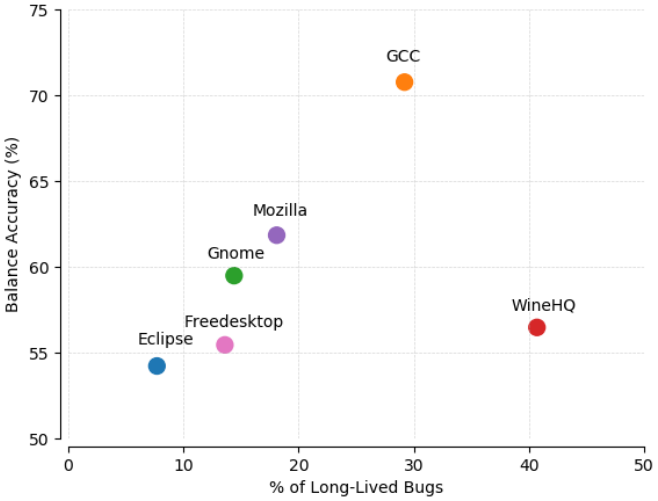


Fig. 16: Balanced Accuracy versus Percentage of Long-Lived Bugs.

1) Reporter Name: Figure 17 shows the number of true positives (charts a and c) and the number of false positives (charts b and d) yielded by Neural Network algorithm when predicting long-lived bugs for Eclipse and GCC test data sets. True positives are long-lived bugs correctly predicted as a long-lived bug. On the other hand, false positives are long-lived bugs incorrectly predicted as a short-lived. The figure grouped the values for true positives and false negatives by bug reporter for each test data set. It is worth to mention that the figure only displays the top ten bug reporter either true or false positive raised in the testing phase of the ML pipeline. Furthermore, the red bar in the charts indicates that the bug reporter is also ranked within the top ten in the whole data set, as shown in Figure 6. We can observe that none of the top 10 ranked reporters of the GCC data set (Figure 17-c) appeared among the top 10 bug reporters considering only true positive bugs.

2) Assignee Name: Figure 18 shows the number of true positives (charts a and c) and the number of false positives (charts b and d) yielded by Neural Network algorithm when predicting long-lived bugs for Eclipse and GCC test data sets. The figure grouped the values for true positives and false

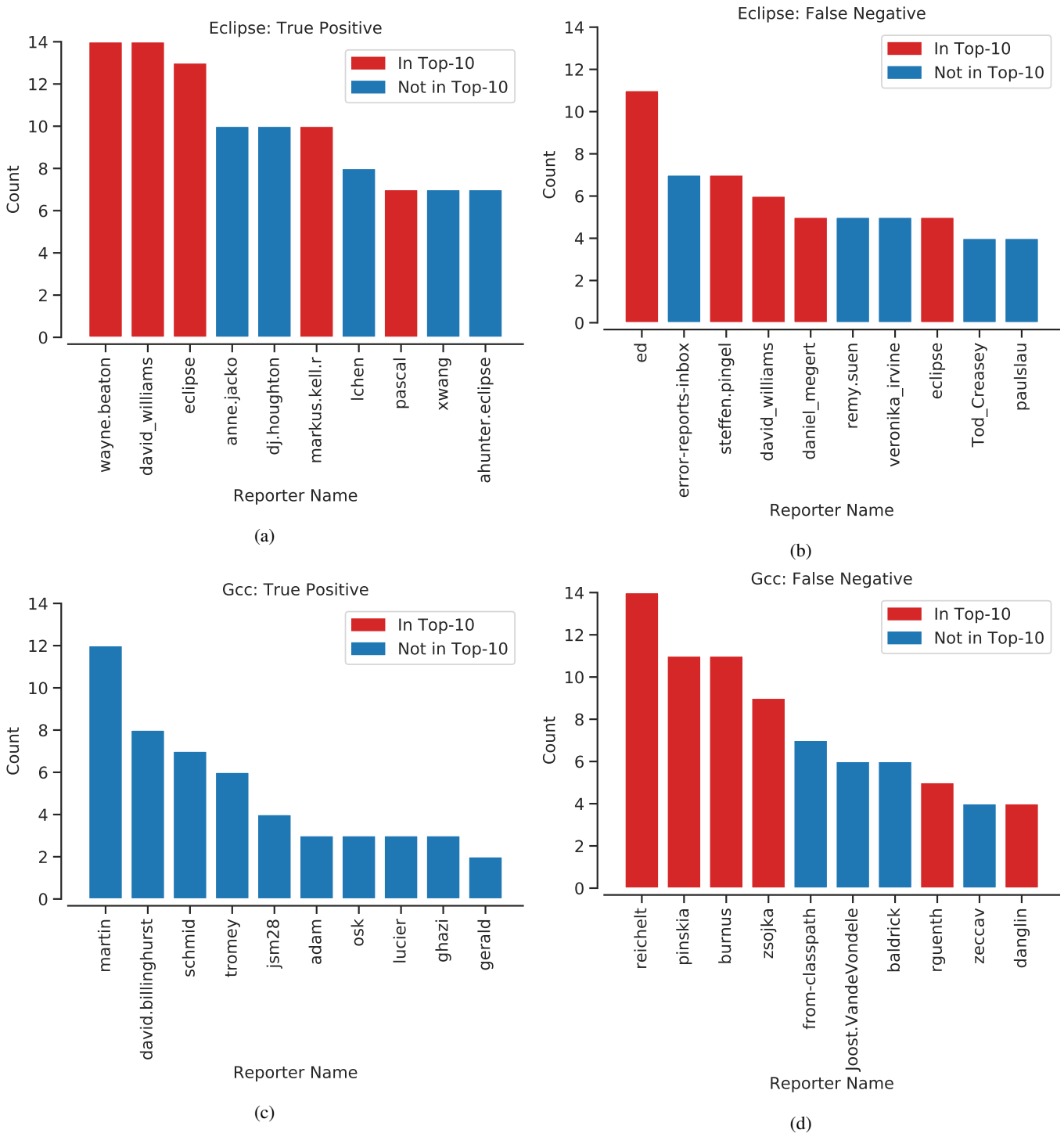


Fig. 17: The bar charts for the top ten bug reporters distribution of bugs used in prediction testing: (a) true positives in Eclipse, (b) false negatives in Eclipse, (c) true positives in GCC, and (d) false negatives in GCC.

negatives by the assignee name for both test data set. It is worth to mention that the figure only displays the top ten assignees for either true or false positive raised in the testing phase of the ML pipeline. Furthermore, the red bar in the charts indicates that the assignee is also ranked within the top ten in the whole data set, as shown in Figure 8. We can note that most true positives and false negatives bugs for GCC had not an assignee (Figure 18 (c) and (d)).

3) Component Name: Figure 19 grouped the values for true positives and false negatives by the component name for Eclipse and GCC test data sets. It is worth to mention that the figure only displays the top ten components for either true or false positive raised in the testing phase of the ML pipeline. Furthermore, the red bar in the charts indicates that the component is also ranked within the top ten in the whole data set, as shown in Figure 10. We can see that most top ten components in Eclipse data set are among the top ten ranked components within true positives bugs (Figure 19 (a)).

4) Severity Level: Figure 20 shows the number of true positives (charts a and c) and the number of false positives (charts b and d) yielded by Neural Network algorithm when predicting long-lived bugs for Eclipse and GCC test data sets. The figure groups the values for true positives and false negatives by severity level for each test data set. Furthermore, this figure indicated that most bugs, either true positive or false negative in both test data sets, had the **normal** severity level.

5) Description: Figure 21 refers to the word clouds of feature vector based on the description field. The leftmost word clouds (a and c) are for true positives and are for false positives (b and d) yielded by Neural Network algorithm when predicting long-lived bugs for Eclipse and GCC test data sets. Colors in clouds denote different values of TF-IDF, while sizes of words denote the scale of TD-IDF. The higher the value of the TD-IDF, the higher the word size will be. We can observe that while true positives bugs have a diversity of words with small and close values of TF-IDF, false negatives bugs have a few words with high and distant values of ITF-IDF.

Figure 22 plots the TF-IDF distribution from feature vector based on the description field. The leftmost histograms (a and c) are for true positives and are for false positives (b and d) yielded by Neural Network algorithm when predicting long-lived bugs for Eclipse and GCC test data sets. We can observe that there is a strong concentration between 2 and 3 of TD-IF values in descriptions of false negative bugs.

VI. Discussions

This section interprets and describes the significance of our findings presented in the previous section. The discussion presented here was organized by the research question, which facilitates the connection with the results section.

A. RQ1: How frequent are the short-lived bugs in FLOSS projects?

Three of six investigated papers [4], [15], [28] defined a long-lived bug in different ways, as shown in Table III. While Saha et al. [4], [28] settled a single value threshold to distinguish a long-lived bug from a short-lived bug based on the release cycle of studied projects, Ginger et al. [15] settled one threshold value based on the median value of bug-fixing time for each FLOSS project. These different visions suggest there is a lack of a shared understanding of what is a long-lived bug. Furthermore, we hypothesize that the definition of the long-live threshold is related to the 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 them.

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

Figure 3(a)-(c) shown that the percentage of long-lived bugs is relative high in investigated FLOSS project.

Saha et al. [**?**] analyzed the Eclipse, GCC, and WineHQ, and conclude that among the total number of bugs that developers fixed, 5%-9% in Java projects (Eclipse) and 14%-37% in C projects (GCC and WineHQ) took more than one year to be fixed. Besides projects analyzed in [4], we investigated Freedesktop, Gnome, and Mozilla projects, and we can observe that our results are following Saha et al. [4].

Another conclusion that we can draw from Figure 3(a)-(c) is that it seems which the percentage of long-lived bugs is higher in projects with less contributors (e.g., WineHQ) than projects with more contributors (e.g., Eclipse). This finding may suggest that the lack of resources affects the bug fixing time negatively in FLOSS projects, as shown in Figure 4.

C. RQ3:

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

1) RQ4.1: Evaluating performance of ML algorithms when predicting long-lived bug.: When comparing the traditional ML algorithm on predicting long-lived bugs using a textual feature vector, we observe that the performance of algorithms is very close with a slight advantage in favor of the Neural Network algorithm (see Table V). However, if we take into account the sensibility rather than balanced accuracy, we will note that Naive Bayes had the best performance.

2) RQ4.2: Evaluating the impact on ML algorithm performance when varying the the number of terms in the feature vector.: Figure 13 shows that increasing the number of terms used in the feature vector decreased the balanced accuracy of Neural Network slightly when predicting a long-lived bug, i.e., the number of terms has not affected the balanced accuracy significantly. The difference between the best-balanced accuracy using terms with the 100 highest weight values and the worst-balanced accuracy using terms with 250 highest weight values, both calculated by the TF-IDF weighting scheme, in the feature vector was only 2%. Therefore, we can consider the former as the group that contains the more significant terms in the textual feature vector needed to distinguish a short-lived from a long-live bug in our current scenario. LUIZ: TODO: discuss repeated terms observed in the word clouds
LUIZ: COLOCAR NA DISCUSSÃO: We aim to find a minimal amount of terms for training without losing too much

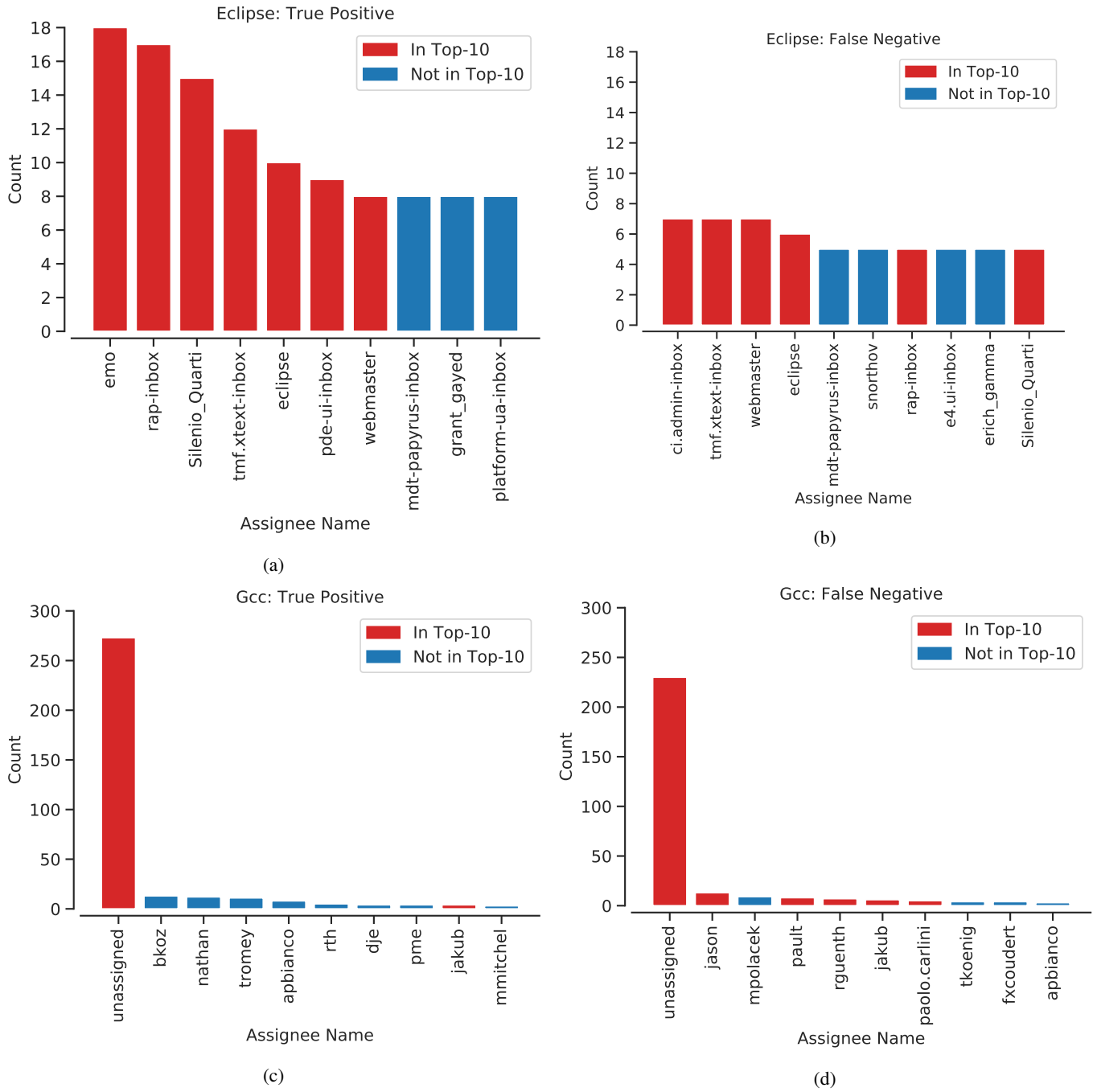


Fig. 18: The bar charts for the top ten assignees distribution of bugs used in prediction testing: (a) true positives in Eclipse, (b) false negatives in Eclipse, (c) true positives in GCC, and (d) false negatives in GCC.

accuracy LUIZ: COLOCAR NA DISCUSSÃO: Notwithstanding this fact, it is recommendable to use the lowest number of terms, in our case 100 terms, in the feature vector without compromising the accuracy by other reasons, such as time processing and memory footprint.

3) RQ4.3: Evaluating the impact on ML algorithm performance when varying the bug fixing time threshold.: The median of bug-fixing time binned bugs reports into short-lived and long-lived equally, i.e., fifty percent of bugs with a bug-fixing time less than or equal to the median are considered short-lived bugs, otherwise, long-lived bugs. Intuitively one would expect that the median to be the most suitable value for the bug fixing time threshold because the binning process will produce a balanced data set. However, figure 14 shows the bug-fixing time threshold based on the median yielded the worst balanced accuracy and sensibility in investigated data sets. On the other hand, the highest balanced accuracy was generated for the bug fixing time threshold equal to 365 days, which was a threshold value suggested by Saha et el. [4] based on the release cycle of Eclipse, GCC and WineHQ projects. Thus, We can suppose the existence of features vectors pattern,

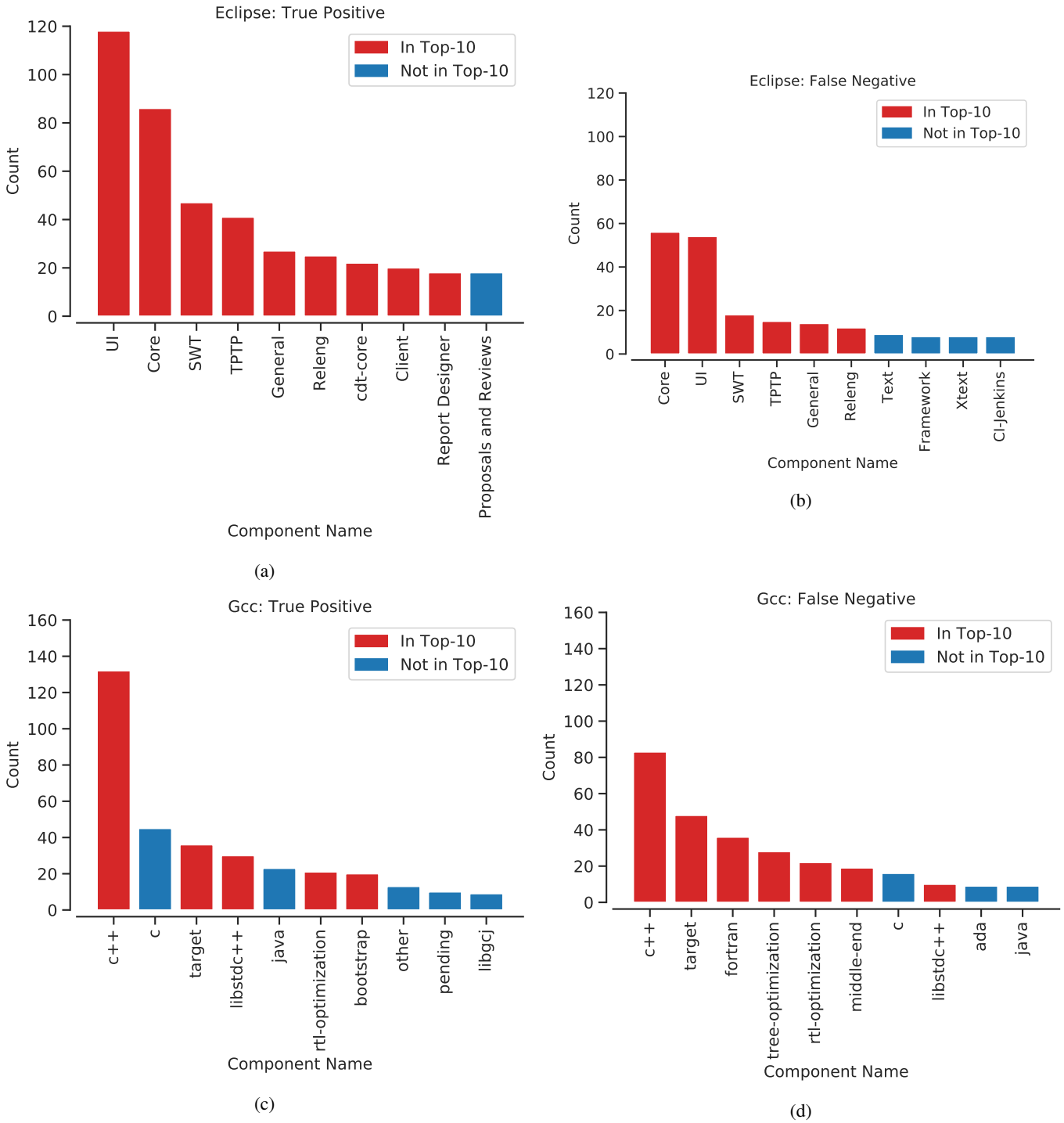


Fig. 19: The bar charts for the top ten components distribution of bugs used in prediction testing: (a) true positives in Eclipse, (b) false negatives in Eclipse, (c) true positives in GCC, and (d) false negatives in GCC.

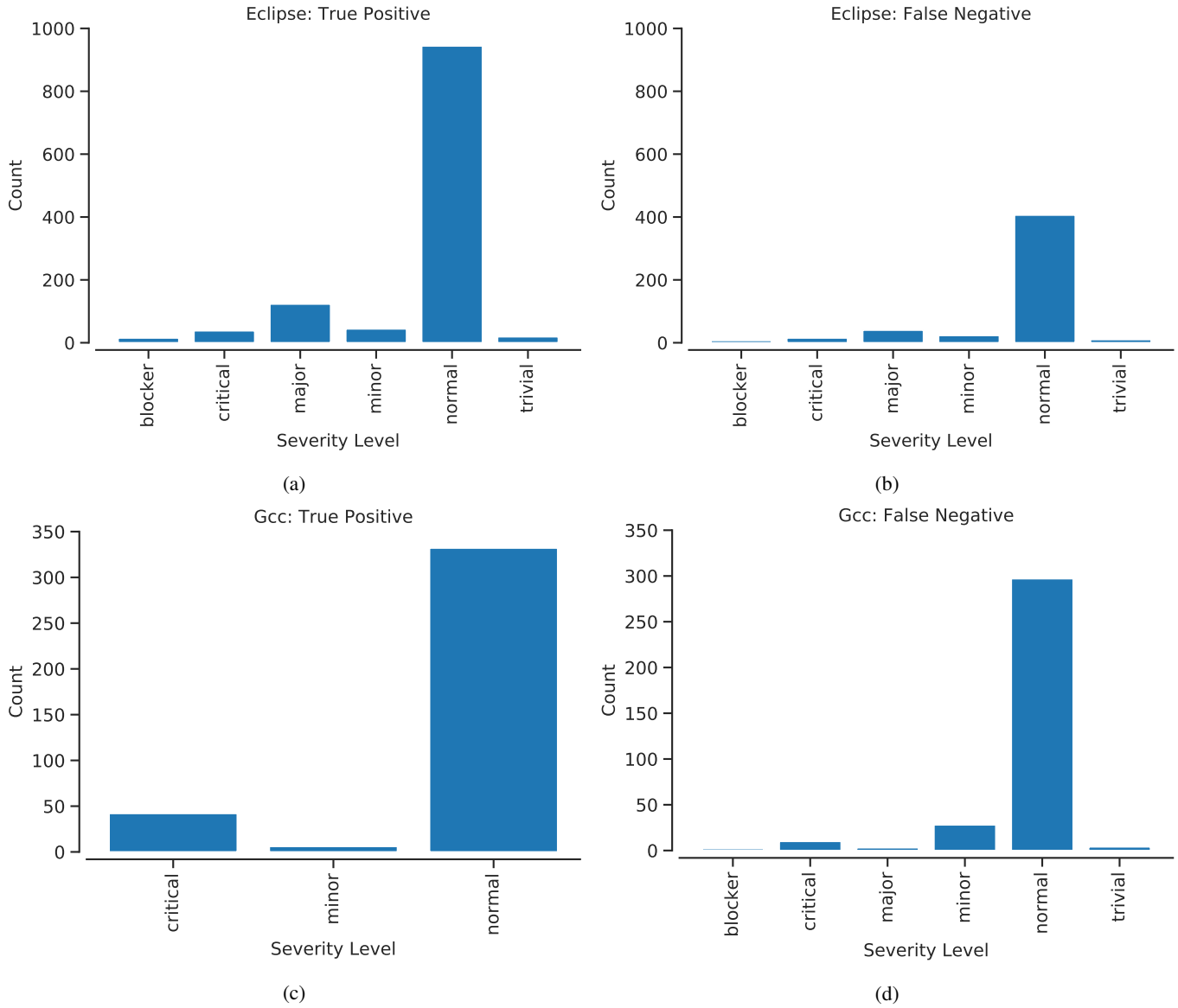


Fig. 20: The bar charts for severity level distribution of bugs used in prediction testing: (a) true positives in Eclipse, (b) false negatives in Eclipse, (c) true positives in GCC, and (d) false negatives in GCC.

which was boosted by the SMOTE method, that better enabled the predictor to distinguish a long-live from a short-lived bug.

4) RQ4.4: Evaluating the performance of ML algorithm in other FLOSS data sets.: Figure 16 shown that

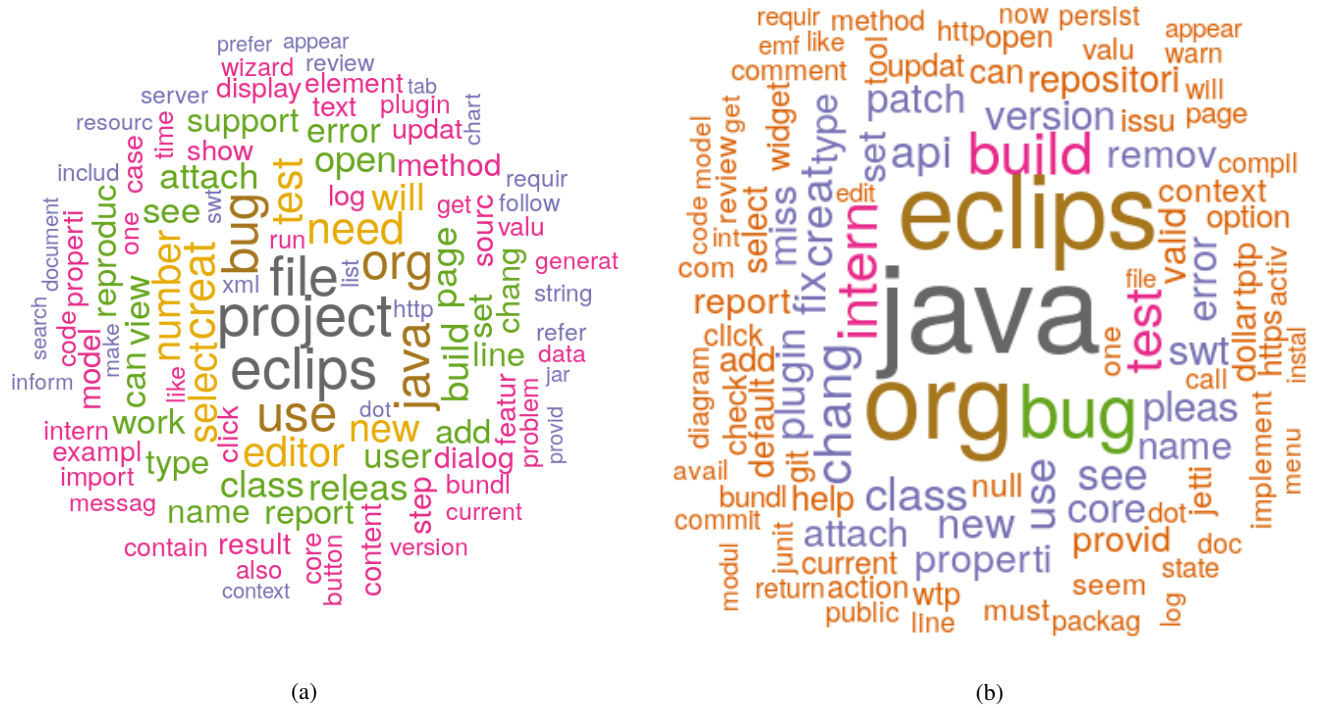
Figure 16 shown that the balanced accuracy performance increased as the number of long-lived bugs increasead, except to WineHQ.

E. RQ5:

VII. Conclusion

In this paper, we have confirmed that most bugs are short-lived. In five of six investigated FLOSS projects, the developer team has fixed 50% of bugs in up to 30 days (RQ_1). Furthermore, we have investigated the performance of four popular ML algorithms to predict long-lived bugs. The results based on bug reports extracted from the Eclipse, Freedesktop, Gnome, Netbeans, Mozilla, and WineHQ FLOSS repositories have shown that Naïve Bayes and k-NN results were better than the other two algorithms to predict long-lived bugs (RQ_2) with good balanced accuracy. Also, we have investigated if the number of terms in the feature vector affects the accuracy of classifiers and we have demonstrated that there is no significant impact in the accuracy of classifiers when predicting long-lived bug (RQ_3). In an additional analysis, we have indicated that 32 days is a bug fixing time threshold that enables the classifier algorithms to yield a good accuracy and is superior to the bug fix time for the most bugs with a short lifecycle (RQ_4).

Validity threats to our research are: (a) We have assumed that bug fix times extracted from repositories are correct and that there is a close relationship between the time for fixing a bug and the short and long descriptions typically found in bug reports (this correlation was also investigated elsewhere [17]);



(a) (b) (c) (d)

Fig. 21: The word cloud for feature vector based on description field of bugs used in prediction testing: (a) true positives in Eclipse, (b) false negatives in Eclipse, (c) true positives in GCC, and (d) false negatives in GCC. The word color denotes a different value for TF-IDF, and the word size indicates the scale of TF-IDF values.

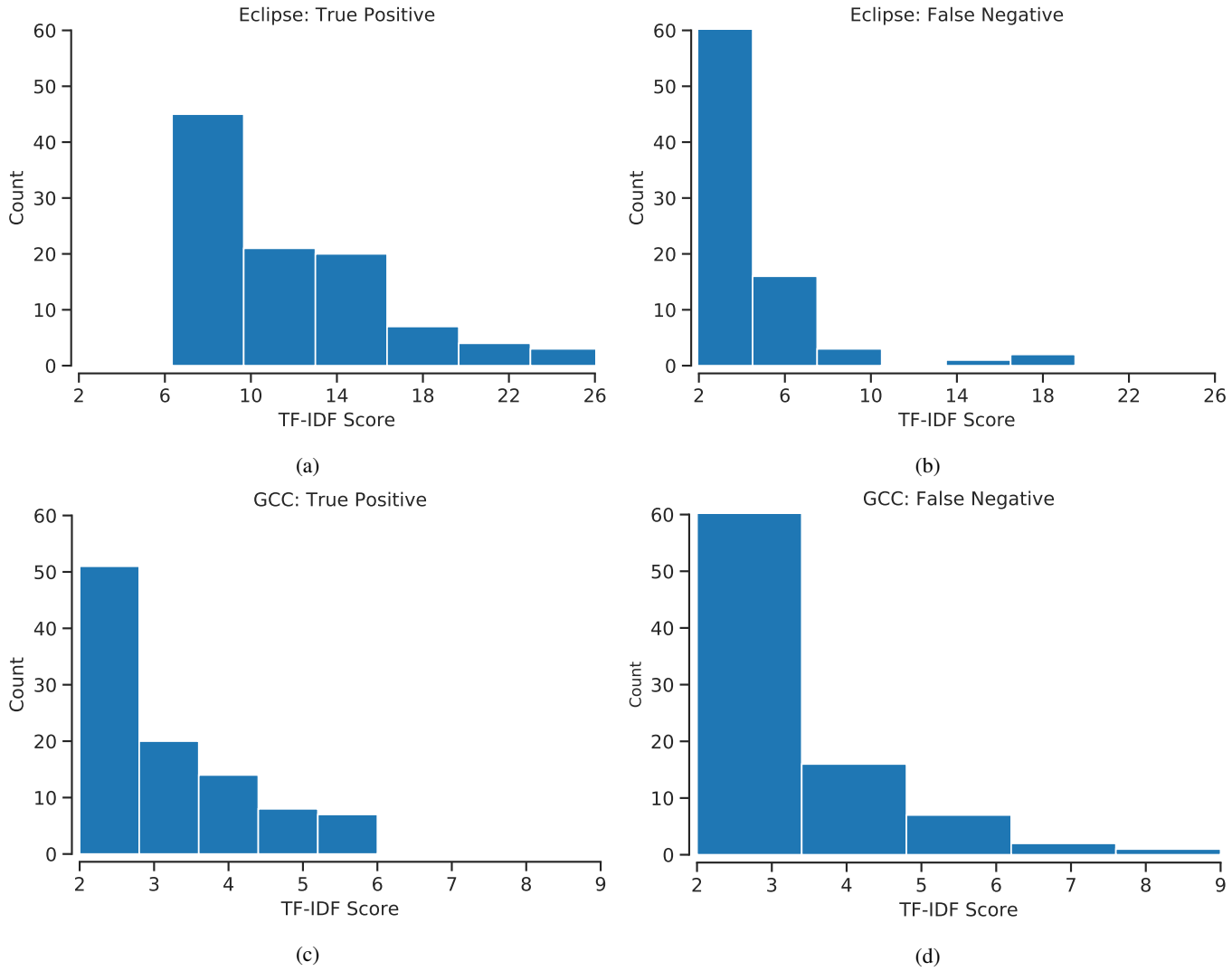


Fig. 22: The distribution histograms of TF-IDF for feature vector based on description field of bugs used in prediction testing: (a) true positives in Eclipse, (b) false negatives in Eclipse, (c) true positives in GCC, and (d) false negatives in GCC.

(b) We have considered six repositories, which lead to a collection composed of 55,490 bug reports. Although we cannot generalize the results to other datasets, the characteristics presented by Eclipse, Freedesktop, Gnome, Netbeans, Mozilla and WineHQ repositories, particularly regarding the balance of the data, are similar to those shown in the repositories studied [4], [15], [16]; (c) another limitation refers to the lack of investigation of the impact of the different parameters of the evaluated classifiers.

As future work, we intend to investigate machine learning algorithms to predict the bug fix time for long-lived bugs. Also, we intend to investigate other repositories and BTS, and develop an approach for representing BTS data generally and uniformly to facilitate the development of a general purpose ML-based long-lived bug prediction assistant.

ACKNOWLEDGMENTS

Authors are grateful to CNPq (grant #307560/2016-3), São Paulo Research Foundation – FAPESP (grants #2014/12236-1, #2015/24494-8, #2016/50250-1, and #2017/20945-0) and the FAPESP-Microsoft Virtual Institute (grants #2013/50155-0 and #2014/50715-9), and The Pontifical Catholic University of Minas Gerais by support to the first author through the Permanent Program For Professor Qualification (PPCD). This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

REFERENCES

[1] Y. C. Cavalcanti, P. A. da Mota Silveira Neto, I. d. C. Machado, T. F. Vale, E. S. de Almeida, and S. R. d. L. Meira, "Challenges and opportunities for software change request repositories: a systematic mapping study," *Journal of Software: Evolution and Process*, vol. 26, no. 7, pp. 620–653, jul 2014.

[2] I. Sommerville, *Software Engineering*, 2010.

[3] G. Yang, S. Baek, J.-W. Lee, and 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*, ser. SAC '17. New York, NY, USA: ACM, 2017, pp. 1280–1287.

[4] R. K. Saha, S. Khurshid, and D. E. Perry, "Understanding the triaging and fixing processes of long lived bugs," *Information and Software Technology*, vol. 65, pp. 114 – 128, 2015.

[5] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals, "Predicting the severity of a reported bug," in *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, May 2010, pp. 1–10.
[6] Y. Tian, D. Lo, and C. Sun, "Information retrieval based nearest neighbor classification for fine-grained bug severity prediction," in *2012 19th Working Conference on Reverse Engineering*, Oct 2012, pp. 215–224.
[7] T. Zhang, G. Yang, B. Lee, and 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*, ser. SAC '15. New York, NY, USA: ACM, 2015, pp. 1553–1558.
[8] P. Flach, *Machine Learning: The Art and Science of Algorithms That Make Sense of Data*. New York, NY, USA: Cambridge University Press, 2012.
[9] S. Marsland, *Machine Learning: An Algorithmic Perspective, Second Edition*, 2nd ed. Chapman & Hall/CRC, 2014.
[10] Y. Tian, N. Ali, D. Lo, and A. E. Hassan, "On the unreliability of bug severity data," *Empirical Softw. Engg.*, vol. 21, no. 6, pp. 2298–2323, dec 2016.
[11] L. Breiman, "Random Forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.
[12] Y. Zhao and Y. Cen, *Data Mining Applications with R*, 1st ed. Academic Press, 2013.
[13] R. Feldman and J. Sanger, *Text Mining Handbook: Advanced Approaches in Analyzing Unstructured Data*. New York, NY, USA: Cambridge University Press, 2006.
[14] G. Williams, *Data Mining with Rattle and R: The Art of Excavating Data for Knowledge Discovery*, 2011.
[15] E. Giger, M. Pinzger, and H. Gall, "Predicting the fix time of bugs," in *Proceedings of the 2Nd International Workshop on Recommendation Systems for Software Engineering*, ser. RSSE '10. New York, NY, USA: ACM, 2010, pp. 52–56.
[16] A. Lamkanfi and S. Demeyer, "Filtering bug reports for fix-time analysis," in *2012 16th European Conference on Software Maintenance and Reengineering*, March 2012, pp. 379–384.
[17] H. Zhang, L. Gong, and S. Versteeg, "Predicting bug-fixing time: An empirical study of commercial software projects," in *2013 35th International Conference on Software Engineering (ICSE)*, May 2013, pp. 1042–1051.
[18] H. Rocha, G. de Oliveira, M. T. Valente, and 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.
[19] M. Habayeb, S. S. Murtaza, A. Miranskyy, and 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*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 700–700.
[20] S. Akbarinasaji, B. Caglayan, and A. Bener, "Predicting bug-fixing time: A replication study using an open source software project," *Journal of Systems and Software*, vol. 136, pp. 173 – 186, 2018.
[21] A. Lamkanfi, S. Demeyer, Q. D. Soetens, and T. Verdonck, "Comparing mining algorithms for predicting the severity of a reported bug," in *2011 15th European Conference on Software Maintenance and Reengineering*, March 2011, pp. 249–258.
[22] H. Valdivia Garcia and E. Shihab, "Characterizing and predicting blocking bugs in open source projects," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: ACM, 2014, pp. 72–81.
[23] E. de Jonge and M. van der Loo, "An introduction to data cleaning with R," *Statistics Netherlands*, p. 53, 2013.
[24] A. Fernandez, S. Garcia, F. Herrera, and N. V. Chawla, "Smote for learning from imbalanced data: Progress and challenges, marking the 15-year anniversary," *Journal of Artificial Intelligence Research*, vol. 61, pp. 863–905, Apr. 2018.
[25] M. Kuhn and K. Johnson, *Applied Predictive Modeling*, ser. SpringerLink : Bücher. Springer New York, 2013.
[26] N. Japkowicz and M. Shah, *Evaluating Learning Algorithms: A Classification Perspective*. New York, NY, USA: Cambridge University Press, 2011.
[27] C. Francalanci and F. Merlo, "Empirical analysis of the bug fixing process in open source projects," in *Open Source Development, Communities and Quality*, B. Russo, E. Damiani, S. Hissam, B. Lundell, and G. Succi, Eds. Boston, MA: Springer US, 2008, pp. 187–196.
[28] R. K. Saha, S. Khurshid, and 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)*, Feb 2014, pp. 144–153.
[29] C. Wang, Y. Li, and B. Xu, "How many versions does a bug live in? an empirical study on text features for bug lifecycle prediction," 2008.

VIII. Questions

- No primeiro experimento da RQ3, utilizar uma tabela (Table V) (completa ou parcial) ou mostrar um gráfico? Como os resultados são muito parecidos a vizualização do gráfico não está muito confortavel.R: *mostrar a tabela de forma parcial e colocar a tabela completa no material completar.*
- Na escolha do melhor algoritmo as métricas Accuracy and Kappa empataram, eu utilizei a Accuracy por ser mais tradicional, porém eu li que a Kappa é mais recomendada para datasets desbalanceados como o nosso. E entre os 10 primeiros resultados a Kappa apareceu 4 vezes, a AUC apareceu 4 vezes e a Accuracy 2 vezes. R: *Utilizar a acurácia por ser mais popular, bem conhecida e mais fácil de entender.*
- Manter a consistência dos nomes no artigo.
- NÃO esquecer de colocar filiação Unicamp, além da PUC MG.
- Trocar feature por bug report field.
- Pensar em subperguntas na RQ1 em tabela.